Abstract Interpretation of Game Properties[⋆,⋆⋆]

Thomas A. Henzinger[1] Rupak Majumdar[1] Freddy Mang[1]
Jean-François Raskin[1,2]

[1]Department of Electrical Engineering and Computer Sciences
University of California at Berkeley, CA 94720-1770, USA
[2]Département d'Informatique, Faculté des Sciences
Université Libre de Bruxelles, Belgium
{tah,rupak,fmang,jfr}@eecs.berkeley.edu

Abstract. We apply the theory of abstract interpretation to the verification of game properties for reactive systems. Unlike properties expressed in standard temporal logics, game properties can distinguish adversarial from collaborative relationships between the processes of a concurrent program, or the components of a parallel system. We consider two-player concurrent games —say, component vs. environment— and specify properties of such games —say, the component has a winning strategy to obtain a resource, no matter how the environment behaves— in the alternating-time μ-calculus ($A\mu$). A sound abstraction of such a game must at the same time restrict the behaviors of the component and increase the behaviors of the environment: if a less powerful component can win against a more powerful environment, then surely the original component can win against the original environment.
We formalize the concrete semantics of a concurrent game in terms of controllable and uncontrollable predecessor predicates, which suffice for model checking all $A\mu$ properties by applying boolean operations and iteration. We then define the abstract semantics of a concurrent game in terms of abstractions for the controllable and uncontrollable predecessor predicates. This allows us to give general characterizations for the soundness and completeness of abstract games with respect to $A\mu$ properties. We also present a simple programming language for multi-process programs, and show how approximations of the maximal abstraction (w.r.t. $A\mu$ properties) can be obtained from the program text. We apply the theory to two practical verification examples, a communication protocol developed at the Berkeley Wireless Research Center, and a protocol converter. In the wireless protocol, both the use of a game property for specification and the use of abstraction for automatic verification were instrumental to uncover a subtle bug.

⋆ A preliminary version of this paper appeared in the *Proceedings of the Seventh International Static Analysis Symposium* (SAS 00), *Lecture Notes in Computer Science* **1824**, Springer-Verlag, 2000, pp. 220–239.

⋆⋆ This research was supported in part by the DARPA (NASA) grant NAG2-1214, the DARPA (Wright-Patterson AFB) grant F33615-C-98-3614, the MARCO grant 98-DT-660, the ARO MURI grant DAAH-04-96-1-0341, and the NSF CAREER award CCR-9501708.

1 Introduction

In compositional verification, one attempts to decompose the task of proving that a system behaves correctly into subtasks which prove that the individual components of the system behave correctly. Often such a proof decomposition cannot proceed blindly, because an individual component may behave correctly only if put into a certain context. Then, some assumptions about the environment of the component are necessary for the subproof to go through. The interaction between a component and its environment is naturally modeled as a two-player infinite game on a state space. If the interaction is synchronous, then the game is *concurrent*: in each round, both players choose their moves simultaneously and independently, and the combination of the two moves determines the next state. (*Turn-based* games for modeling asynchronous or interleaved interaction, where in each round only one of the two players has a choice, can be considered a special case of concurrent games.) If player 1 represents a component, and player 2 represents the environment assumptions, then a typical property of interest is "Does player 1 have a strategy to reach a goal, say, obtain a resource, no matter how player 2 behaves." A rich logic for specifying such game properties formally is the *alternating-time μ-calculus*, denoted $A\mu$ [1], which subsumes several temporal logics for expressing game properties. In [29] the abstract interpretation of the more special case of turn-based games is considered, but the model checking problem of a general class of game properties is not considered.

While there exist algorithms for model checking game properties [1, 16], as usual in model checking, the involved state spaces may be prohibitively large. The common remedy is *abstraction*: the verification engineer attempts to simplify the component model and the environment assumptions as much as possible while still preserving soundness. If the simplifications are sound, and satisfy the desired property, then we can be sure that the actual system also satisfies the property. (By contrast, completeness must often be sacrificed: it may be that the actual system is correct, while the simplified system is not. If an error is found which has no counterpart in the actual system, then some of the simplifying assumptions must be reversed.) For linear-time and branching-time properties, it is well-known how to choose sound simplifications, and how to characterize complete ones. For example, if the objective is to establish a temporal requirement for all traces of a system, then a sound simplification must allow more traces; if the objective is to establish a temporal requirement for some trace, then a sound simplification must allow fewer traces. For game properties, the situation is more complicated. For example, with respect to the property "Does player 1 have a strategy to reach a goal," a simplification is sound if it restricts the power of player 1 and at the same time increases the power of player 2. In this paper, we give a general characterization of soundness and completeness for simplifying games with respect to $A\mu$ properties. This theory is then applied to two practical verification examples, a wireless communication protocol and a protocol converter.

We work in the *abstract-interpretation* framework of [7, 9], which makes precise the notion of "simplification" used informally in the previous paragraph. We first give a set of predicates which are sufficient for model checking all $A\mu$ properties by applying boolean operations and iteration. Following [9], this is called the *collecting semantics* of a game. The essential ingredients of the collecting semantics are the *player-i controllable predecessor* predicate, which relates a state q with a set σ of states if player i can force the game from q into σ in a single round, and its dual, the *player-i uncontrollable predecessor* predicate, which relates q with σ if player i cannot prevent the game from moving from q into σ in a single round. We then define what it means for an abstraction of the collecting semantics and the corresponding abstract model checking algorithm to be *sound* (if an abstract state q^α satisfies an $A\mu$ formula, then so do all states in the concretization of q^α), and *complete* (q^α satisfies an $A\mu$ formula if some state in the concretization of q^α does). The completeness of abstractions will be characterized in terms of the *alternating bisimilarity* relation of [2]. This contrasts with the cases in linear-time and branching-time domains: for linear-time properties, completeness requires trace equivalence; for branching-time properties, bisimilarity. Our results can thus be seen to be systematic generalizations of the abstract-interpretation theory for temporal requirements from (single-player) transition systems [6, 26, 12, 14, 11] to (mutli-player) game structures.

While our development applies to concurrent game structures in general, in practice it is preferable to derive the abstraction of the collecting semantics directly from the text of a program [7, 14]. Such a direct computation (defined by structural induction on the programming language) may lose precision, and we typically obtain only an approximation of the *maximal* abstraction (an abstract state q^α satisfies an $A\mu$ formula if all states in the concretization of q^α do). We introduce a simple programming language for multi-process programs based on guarded commands [18]. We interpret processes as players in a concurrent game, and show how to compute approximations of the maximal abstraction directly from the program text. We present both *domain abstraction*, a nonrelational form of abstraction where each variable is interpreted over an abstract domain, and *predicate abstraction*, a relational abstraction which permits more accuracy by relating the values of different variables via abstract predicates [24].

Abstract interpretation has been used successfully in the automated verification of reactive systems [13, 14, 23, 5, 27]. We illustrate the application of the theory to the automated verification of game properties with two practical examples. The first example originates from the *Two-Chip Intercom* (TCI) project of the Berkeley Wireless Research Center [4]. The TCI network is a wireless local network which allows approximately 40 remotes, one for each user, to transmit voice with point-to-point and broadcast communication. The operation of the network is coordinated by a base station, which assigns channels to the users through a TDMA scheme. Properties specifying the correct operation of each remote can be given in the game logic $A\mu$: each remote must behave correctly in an environment containing the base station and arbitrarily many other remotes. We verified the protocol for a base station and an arbitrary number of remotes. Since the system is infinite state, in order to use our model checkerMocha [3], we needed to abstract it to a finite instance. A bug was found on an abstract version of the protocol. The violated property involves an adversarial behavior of the base station with respect to a remote, and cannot be specified directly in a nongame logic like CTL. Thus, both game properties and abstract interpretation were necessary in the verification process.

The second example concerns the automatic synthesis of a protocol converter between a message sender which speaks the alternating-bit protocol and a receiver which speaks a simple two-phase protocol. We view the problem as a special case of *controller synthesis*, which is in turn a special case of $A\mu$ model checking. We view the composition of the sender and the receiver as the system to be controlled, and the protocol converter as the controller to be synthesized. The requirements of the converter is written in the game logic $A\mu$. Using predicate and domain abstractions, we are able to check for the existence and construct a converter which satisfies the requirements.

2 Structures and Logics for Games

Alternating transition systems. An *alternating transition system* [1] is a tuple $S = \langle \Sigma, Q, \Delta, \Pi, \pi \rangle$ with the following components: (i) Σ is the (finite) set of *players*. (ii) Q is a (possibly infinite) set of *states*. (iii) $\Delta = \{\delta_i : Q \to 2^{2^Q} \mid i \in \Sigma\}$ is a set of *transition functions*, one for each player in Σ, which maps each state to a nonempty set of *choices*, where each choice is a set of possible next states. Whenever the system is in state q, each player $a \in \Sigma$ independently and simultaneously chooses a set $Q_a \in \delta_a(q)$. In this way, a player a ensures that the next state of the system will be in its choice Q_a. However, which state in Q_a will be next depends on the choices made by the other players, because the successor of q must lie in the intersection $\bigcap_{a \in \Sigma} Q_a$ of the choices made by all players. We assume that the transition function is nonblocking and the players together choose a unique next state: if $\Sigma = \{a_1, \ldots, a_n\}$, then for every state $q \in Q$ and every set $Q_1, \ldots, Q_n$ of choices $Q_i \in \delta_{a_i}(q)$, the intersection $Q_1 \cap \ldots \cap Q_n$ is a singleton. Note that we do not lose expressive power by considering only deterministic games, because nondeterminism can be modeled by an additional player. (iv) Π is a set of *propositions*. (v) $\pi : \Pi \to 2^Q$ maps each proposition to a set of states.

From the definition it can be seen that alternating transition systems can model general *concurrent games*, and includes as a special case turn-based games. For two states q and q' and a player $a \in \Sigma$, we say q' is an a-successor of q if there exists a set $Q' \in \delta_a(q)$ such that $q' \in Q'$. For two states q and q', we say q' is a *successor* of q if for all players $a \in \Sigma$, the state q' is an a-successor of q. A *computation* $\eta = q_0 q_1 \ldots$ is a finite or infinite sequence of states such that q_{i+1} is a successor of q_i for all $i \geq 0$. A computation produces a *trace* $\tau = \pi(q_0)\pi(q_1)\ldots$ of sets of propositions. A *strategy* for a player $a \in \Sigma$ is a mapping $f_a : Q^+ \to 2^Q$ such that for $w \in Q^*$ and $q \in Q$, we have $f_a(w \cdot q) \in \delta_a(q)$. Thus, the strategy f_a maps a finite nonempty prefix $w \cdot q$ of a computation to a set in $\delta_a(q)$: this set contains possible extensions of the computation as suggested to player a by the strategy. For fixed strategies $F = \{f_a \mid a \in \Sigma\}$, the computation $\eta = q_0 q_1 \ldots$

is consistent with F if for all $i \geq 0$, $q_{i+1} \in f_a(q_0 q_1 \ldots q_i)$ for all $a \in \Sigma$. For a state $q \in Q$, we define the *outcome* $\mathcal{L}_F(q)$ of F with source q as the set of possible traces produced by the computations which start from state q, and are consistent with the strategies F.

For ease of presentation, we consider in the following only two players, whom we call player 1 and player 2 respectively, i.e., $\Sigma = \{1, 2\}$. The results generalize immediately to multiple players.

Alternating-time μ-calculus. A *game logic* L is a logic whose formulas are interpreted over the states of alternating transition systems; that is, for every L-formula φ and every alternating transition system S, there is a set $[\![\varphi]\!]_S$ of states of S which satisfy φ. The *L model checking problem* for a game logic L and an alternating transition system S asks, given an L-formula φ and a state q of S, whether $q \in [\![\varphi]\!]_S$.

The formulas of the *alternating time μ-calculus* [1] are generated by the grammar

$$\varphi \ ::= \ p \mid \overline{p} \mid x \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \langle\!\langle I \rangle\!\rangle \bigcirc \varphi \mid [\![I]\!] \bigcirc \varphi \mid (\mu x \colon \varphi) \mid (\nu x \colon \varphi),$$

for propositions p in some set Π^L of propositions, variables x in some set X of variables, and teams of players $I = 1, 2, \{1, 2\}$. Let $S = \langle \Sigma, Q, \Delta, \Pi, \pi \rangle$ be an alternating transition system whose propositions include all propositions on which formulas are constructed; that is, $\Pi^L \subseteq \Pi$. Let $\mathcal{E} \colon X \to 2^Q$ be a mapping from the variables to sets of states. We write $\mathcal{E}[x \mapsto \rho]$ for the mapping that agrees with $\mathcal{E}$ on all variables, except that $x \in X$ is mapped to $\rho \subseteq Q$. Given S and $\mathcal{E}$, every formula φ defines a set $[\![\varphi]\!]_{S,\mathcal{E}} \subseteq Q$ of states:

$$\begin{aligned}
[\![p]\!]_{S,\mathcal{E}} &= \pi(p);\\
[\![\overline{p}]\!]_{S,\mathcal{E}} &= Q \backslash \pi(p);\\
[\![x]\!]_{S,\mathcal{E}} &= \mathcal{E}(x);\\
[\![\varphi_1 \left\{ {\vee \atop \wedge} \right\} \varphi_2]\!]_{S,\mathcal{E}} &= [\![\varphi_1]\!]_{S,\mathcal{E}} \left\{ {\cup \atop \cap} \right\} [\![\varphi_2]\!]_{S,\mathcal{E}};\\
[\![\left\{ {\langle\!\langle 1 \rangle\!\rangle \atop [\![1]\!]} \right\} \bigcirc \varphi]\!]_{S,\mathcal{E}} &= \{q \in Q \mid (\left\{ {\exists \sigma \in \delta_1(q). \forall \tau \in \delta_2(q) \atop \forall \sigma \in \delta_1(q). \exists \tau \in \delta_2(q)} \right\} r \in \sigma \cap \tau \colon r \in [\![\varphi]\!]_{S,\mathcal{E}})\};\\
[\![\left\{ {\langle\!\langle 1,2 \rangle\!\rangle \atop [\![1,2]\!]} \right\} \bigcirc \varphi]\!]_{S,\mathcal{E}} &= \{q \in Q \mid (\left\{ {\exists \sigma \in \delta_1(q). \exists \tau \in \delta_2(q) \atop \forall \sigma \in \delta_1(q). \forall \tau \in \delta_2(q)} \right\} r \in \sigma \cap \tau \colon r \in [\![\varphi]\!]_{S,\mathcal{E}})\};\\
[\![\left\{ {\mu \atop \nu} \right\} x \colon \varphi]\!]_{S,\mathcal{E}} &= \left\{ {\bigcap \atop \bigcup} \right\} \{\rho \subseteq Q \mid \rho = [\![\varphi]\!]_{S,\mathcal{E}[x \mapsto \rho]}\}.
\end{aligned}$$

If we restrict ourselves to the closed formulas, then we obtain a game logic, denoted $A\mu$: the state $q \in Q$ *satisfies* the $A\mu$-formula φ if $q \in [\![\varphi]\!]_{S,\mathcal{E}}$ for any variable mapping $\mathcal{E}$; that is, $[\![\varphi]\!]_S = [\![\varphi]\!]_{S,\mathcal{E}}$ for any $\mathcal{E}$.

The logic $A\mu$ is very expressive and embeds the game logics ATL and ATL*[1]. For example, the ATL-formula $\langle\!\langle 1 \rangle\!\rangle \varphi_1 \mathcal{U} \varphi_2$ can be expressed in $A\mu$ as $(\mu x \colon \varphi_2 \vee (\varphi_1 \wedge \langle\!\langle 1 \rangle\!\rangle \bigcirc x))$. Note that the fragment which restricts all game quantifiers $\langle\!\langle I \rangle\!\rangle$ and $[\![I]\!]$ to the team $I = \{1, 2\}$ is the standard μ-calculus. Thus, our results include as a special case the results of [14].

Collecting semantics of alternating transition systems. Alternating transition systems provide an *operational semantics* to our model of systems with interacting components. In addition, we isolate the operators we need to evaluate on an alternating transition system in order to compute the set of states where a formula of the logic $A\mu$ holds. We call these operators, following [7, 8], the *collecting semantics* of the alternating transition system. The collecting semantics of an alternating transition system may be thought of as an instrumented version of the operational semantics in order to gather useful information about temporal properties of a system. Given an alternating transition system $S = \langle \Sigma, Q, \Delta, \Pi, \pi \rangle$, the collecting semantics consists of the following operators.

States satisfying a proposition or the negation of a proposition. For every proposition $p \in \Pi$, and its negation $\bar{p}$, we define $\langle\!| p |\!\rangle = \pi(p)$ and $\langle\!| \bar{p} |\!\rangle = Q \setminus \pi(p)$.

Controllable and uncontrollable predecessors. We define the *player-1 controllable predecessor* relation $CPre_1 : 2^Q \to 2^Q$ as $q \in CPre_1(\sigma)$ iff $\exists \tau \in \delta_1(q).\ \forall \tau' \in \delta_2(q).\ \tau \cap \tau' \subseteq \sigma$ The state q is in the set of controllable predecessors of the set of states σ if player 1 can make a choice such that for all choices of player-2, the successor state of q lies in the set σ. Thus in q, player 1 has the ability to force the next state of the game into σ. We define the *player-1 uncontrollable predecessor* relation $UPre_1 : 2^Q \to 2^Q$ as $q \in UPre_1(\sigma)$ iff $\forall \tau \in \delta_1(q).\ \exists \tau' \in \delta_2(q).\ \tau \cap \tau' \subseteq \sigma$. So the state q is in the set of uncontrollable predecessors of the set of states σ if for each choice of player 1 in q, there exists a choice of player 2 such that the successor state of q is in σ. Thus in q, player 1 cannot force the game outside σ without the cooperation of player 2, or equivalently, player 1 cannot avoid σ. We can similarly define the player 2 controllable and uncontrollable predecessor relations $CPre_2$ and $UPre_2$. The team-$\{1,2\}$ predecessor relations $CPre_{\{1,2\}}$ and $UPre_{\{1,2\}}$ are defined as:

$$q \in CPre_{\{1,2\}}(\sigma) \text{ iff } \exists \tau \in \delta_1(q).\ \exists \tau' \in \delta_2(q).\ \tau \cap \tau' \subseteq \sigma$$
$$q \in UPre_{\{1,2\}}(\sigma) \text{ iff } \forall \tau \in \delta_1(q).\ \forall \tau' \in \delta_2(q).\ \tau \cap \tau' \subseteq \sigma$$

From the definitions above, we can establish the following propositions.

Proposition 1. *The operators $CPre_I$ and $UPre_I$ (for $I = 1, 2, \{1,2\}$) are duals of each other, that is, if $\sigma \subseteq Q$ is a set of states and $\neg\sigma = Q \setminus \sigma$, then $CPre_I(\sigma) = \neg\, UPre_I(\neg\sigma)$.*

Proposition 2. *The operators $CPre_I$ and $UPre_I$ are monotonic, that is, for all sets of states σ_1, σ_2 such that $\sigma_1 \subseteq \sigma_2 \subseteq Q$, we have $CPre_I(\sigma_1) \subseteq CPre_I(\sigma_2)$ and $UPre_I(\sigma_1) \subseteq UPre_I(\sigma_2)$.*

Model checking with game operators The definition of $A\mu$ naturally suggests a model checking method for finite state systems, where fixpoints are computed by successive approximations; note that the operators $CPre_I$ and $UPre_I$ for $I = 1, 2, \{1,2\}$ correspond naturally to the semantics of the logical formulas $\langle\!\langle I \rangle\!\rangle \bigcirc$ and $[\![I]\!] \bigcirc$. For alternating transition systems with a collecting semantics defined by the above operators, one can similarly define a model checking

Semi-algorithm ModelCheck
Input: the collecting semantics of an alternating transition system with operators $\langle\!| \cdot |\!\rangle$, $CPre_I$, $UPre_I$, a formula $\varphi \in A\mu$, and a mapping E with domain X.

Output: $[\varphi]_{S,E} :=$

if $\varphi = p$ **then return** $\langle\!| p |\!\rangle$;
if $\varphi = \overline{p}$ **then return** $\langle\!| \overline{p} |\!\rangle$;
if $\varphi = (\varphi_1 \vee \varphi_2)$ **then return** $[\varphi_1]_{S,E} \cup [\varphi_2]_{S,E}$;
if $\varphi = (\varphi_1 \wedge \varphi_2)$ **then return** $[\varphi_1]_{S,E} \cap [\varphi_2]_{S,E}$;
if $\varphi = \langle\!\langle I \rangle\!\rangle \bigcirc \varphi'$ **then return** $CPre_I([\varphi']_{S,E})$;
if $\varphi = [\![I]\!] \bigcirc \varphi'$ **then return** $UPre_I([\varphi']_{S,E})$;
if $\varphi = (\mu x \colon \varphi')$ **then**
    $T_0 := \emptyset$;
    for $i = 0, 1, 2, \ldots$ **do**
        $T_{i+1} := [\varphi']_{S,E[x \mapsto T_i]}$
        until $T_{i+1} \subseteq T_i$;
    return T_i;
if $\varphi = (\nu x \colon \varphi')$ **then**
    $T_0 := Q$;
    for $i = 0, 1, 2, \ldots$ **do**
        $T_{i+1} := [\varphi']_{S,E[x \mapsto T_i]}$
        until $T_{i+1} \supseteq T_i$;
    return T_i.

Fig. 1. $A\mu$ model checking

procedure that uses boolean operations, as well as the predecessor operations $CPre_I$ and $UPre_I$ [1]. The procedure ModelCheck of Figure 1 takes as input an alternating transition system S described by its collecting semantics, a formula $\varphi \in A\mu$, and an environment E mapping variables to sets of states, and produces a set $[\varphi]_{S,E}$ of states.

Theorem 1. *If the semi-algorithm* ModelCheck *terminates, then* $[\varphi]_{S,E} = [\![\varphi]\!]_S$ *for any closed formula* φ *of* $A\mu$ *and any environment* E.

In general, the fixpoints may not converge in a finite number of steps, and transfinite iteration may be required.

3 Abstractions of Alternating Transition Systems

Let Q^α be a set of *abstract states* and $\gamma : Q^\alpha \to 2^Q$ be a *concretization function* that maps each abstract state q^α to a set of concrete states $\gamma(q^\alpha)$ which q^α represents. We define a *precision order* $\preceq \subseteq Q^\alpha \times Q^\alpha$ on the abstract domain, as $q_1^\alpha \preceq q_2^\alpha$ iff $\gamma(q_1^\alpha) \subseteq \gamma(q_2^\alpha)$. Thus q_1^α is *more precise than* q_2^α if the set of concrete states represented by q_1^α is a subset of the set of concrete states represented by q_2^α. So by definition, γ is monotonic w.r.t. $\preceq$. Let $\hat{\gamma}(\sigma^\alpha) = \bigcup\{\gamma(q^\alpha) \mid q^\alpha \in \sigma^\alpha\}$

denote the set extension of γ. We extend the order $\preceq$ over sets of abstract states, giving $\hat{\preceq} \subseteq 2^{Q^\alpha} \times 2^{Q^\alpha}$, as $\sigma_1^\alpha \hat{\preceq} \sigma_2^\alpha$ iff $\hat{\gamma}(\sigma_1^\alpha) \subseteq \hat{\gamma}(\sigma_2^\alpha)$. A set σ_1^α of abstract states is an *approximation* of a set σ_2^α of abstract states if $\sigma_1^\alpha \hat{\preceq} \sigma_2^\alpha$.

Given an alternating transition system S with state space Q and set of abstract states Q^α with concretization function γ, we want to compute the abstraction of the concrete semantics of any $A\mu$-formula φ, denoted $[\![\varphi]\!]_S^\alpha$. An abstract interpretation is *sound* if properties that we establish with the abstract algorithm are true in the concrete semantics; i.e., $\hat{\gamma}([\![\varphi]\!]_S^\alpha) \subseteq [\![\varphi]\!]_S$ In the sequel, we only consider sound abstract interpretations. Conversely, an abstract interpretation is *complete* if properties that are true on the concrete domain can be established by the abstract interpretation; i.e., $[\![\varphi]\!]_S \subseteq \hat{\gamma}([\![\varphi]\!]_S^\alpha)$. In general, an abstract interpretation is not complete unless strong conditions are fulfilled by the abstract domain and the concretization function. We will give a necessary and sufficient condition on the abstract domain and the concretization function for complete model checking of $A\mu$-properties. In addition to soundness and completeness, one is also interested in the *maximality* of abstract interpretations. The abstract interpretation is maximal if we have for all abstract state $q^\alpha \in Q^\alpha$, if $\gamma(q^\alpha) \subseteq [\![\varphi]\!]_S$ then $q^\alpha \in [\![\varphi]\!]_S^\alpha$. This means that if a property is true in all concrete states represented by an abstract state q^α, then the abstract model checking algorithm is able to establish it. As for completeness, maximality is also lost unless we impose strong conditions on the abstract domain and operations necessary to compute the abstract semantics. We refer the interested reader to [21, 22] for a general treatment of maximality in abstract interpretation.

Abstraction of the collecting semantics. We have shown in the previous section that $[\![\varphi]\!]_S$ can be computed using the collecting semantics of S. We now explain how the collecting semantics can be abstracted. For each component $(\!| \cdot |\!)$, $CPre_I$, and $UPre_I$ of the collecting semantics. We define the abstract counterpart $(\!| \cdot |\!)^\alpha$, $CPre_I^\alpha$, and $UPre_I^\alpha$.

Abstract semantics of propositions. For each proposition $p \in \Pi^L$ we define $(\!|p|\!)^\alpha = \{q^\alpha \in Q^\alpha \mid \gamma(q^\alpha) \subseteq (\!|p|\!)\}$ and $(\!|\bar{p}|\!)^\alpha = \{q^\alpha \in Q^\alpha \mid \gamma(q^\alpha) \subseteq (\!|\bar{p}|\!)\}$. From this it follows that the abstract semantics for propositions is sound. Moreover, for abstract states q^α, r^α with $q^\alpha \preceq r^\alpha$, if $r^\alpha \in (\!|p|\!)^\alpha$ then $q^\alpha \in (\!|p|\!)^\alpha$. Note that given a proposition $p \in \Pi^L$ and an abstract state q^α, we can have $q^\alpha \notin (\!|p|\!)^\alpha$ and $q^\alpha \notin (\!|\bar{p}|\!)^\alpha$. This occurs when the abstract state q^α represents at the same time concrete states where the proposition p evaluates to true and other concrete states where the proposition p evaluates to false.

Abstract controllable and uncontrollable predecessors. Let q^α be an abstract state and σ^α be a set of abstract states, we define the abstract controllable predecessor relation as: $q^\alpha \in CPre_I^\alpha(\sigma^\alpha)$ iff $\forall q \in \gamma(q^\alpha).\ q \in CPre_I(\hat{\gamma}(\sigma^\alpha))$. So an abstract state q^α is included in the abstract controllable predecessors of an abstract region σ^α if all the concrete states represented by q^α are in the controllable predecessors of the set of concrete states represented by the set of abstract states σ^α.

Similarly, the abstraction of the uncontrollable predecessor relation is defined as $q^\alpha \in \mathit{UPre}^\alpha_I(\sigma^\alpha)$ iff $\forall q \in \gamma(q^\alpha).\ q \in \mathit{UPre}_I(\hat{\gamma}(\sigma^\alpha))$. The soundness and the maximality of the abstract controllable and uncontrollable predecessors follow from the definitions.

Lemma 1. Soundness and maximality. *For every set σ^α of abstract states, $\hat{\gamma}(\mathit{CPre}^\alpha_I(\sigma^\alpha)) \subseteq \mathit{CPre}_I(\hat{\gamma}(\sigma^\alpha))$ and $\hat{\gamma}(\mathit{UPre}^\alpha_I(\sigma^\alpha)) \subseteq \mathit{UPre}_I(\hat{\gamma}(\sigma^\alpha))$, expressing soundness. Also, if $q^\alpha \notin \mathit{CPre}^\alpha_I(\sigma^\alpha)$ then $\gamma(q^\alpha) \nsubseteq \mathit{CPre}_I(\hat{\gamma}(\sigma^\alpha))$ and if $q^\alpha \notin \mathit{UPre}^\alpha_I(\sigma^\alpha)$ then $\gamma(q^\alpha) \nsubseteq \mathit{UPre}_I(\hat{\gamma}(\sigma^\alpha))$, expressing maximality.*

Abstract model checking of the alternating-time μ-calculus. An *abstract model checking algorithm* takes as input an abstraction of the collecting semantics of the alternating transition system and an $A\mu$ formula φ , and computes a set of abstract states. This defines the *abstract semantics* of φ. Let AbsModelCheck be the abstract model checking algorithm obtained from ModelCheck by replacing the concrete collecting semantics $\langle\!| \cdot |\!\rangle$, CPre_I, and UPre_I by their respective abstract collecting semantics $\langle\!| \cdot |\!\rangle^\alpha$, CPre^α_I, and UPre^α_I for $I = 1, 2, \{1, 2\}$. The soundness of AbsModelCheck is proved by induction on the structure of formulas, using the soundness of the abstraction of the collecting semantics.

Theorem 2. Soundness of AbsModelCheck. *The abstract model checking algorithm* AbsModelCheck *is sound, i.e., if the algorithm* AbsModelCheck *produces the abstract region $[\varphi]^\alpha_S$ on input formula φ and the abstract collecting semantics of S, then for all abstract states $q^\alpha \in [\varphi]^\alpha_S$, and for all concrete states $q \in \gamma(q^\alpha)$, we have $q \in [\![\varphi]\!]_S$.*

In the proof of soundness, we can replace each of the predicates $\langle\!| \cdot |\!\rangle^\alpha$, CPre^α_I, and UPre^α_I by approximations without losing the soundness of the abstract model checking algorithm. This is because any approximation of the sound abstraction of the collecting semantics remains sound. This statement is made precise in the following lemma.

Lemma 2. Approximation. *The soundness of the abstract model checking algorithm* AbsModelCheck *is preserved if the predicates $\langle\!| \cdot |\!\rangle^\alpha$, CPre^α_I, and UPre^α_I are replaced by approximations $\langle\!| \cdot |\!\rangle^A$, CPre^A_I, and UPre^A_I such that:*

1. *for all $p \in \Pi^L$, we have $\langle\!| p |\!\rangle^A \hat{\preceq} \langle\!| p |\!\rangle^\alpha$, and $\langle\!| \bar{p} |\!\rangle^A \hat{\preceq} \langle\!| \bar{p} |\!\rangle^\alpha$;*
2. *for all $\sigma^\alpha \subseteq Q^\alpha$, we have $\mathit{CPre}^A_I(\sigma^\alpha) \hat{\preceq} \mathit{CPre}^\alpha_I(\sigma^\alpha)$;*
3. *for all $\sigma^\alpha \subseteq Q^\alpha$, we have $\mathit{UPre}^A_I(\sigma^\alpha) \hat{\preceq} \mathit{UPre}^\alpha_I(\sigma^\alpha)$.*

Although the abstract interpretations for the propositions and the controllable and uncontrollable predecessors are maximal, unfortunately the maximality is not preserved by the abstract model checking algorithm AbsModelCheck. This is because the abstract model checking algorithm is defined compositionally. This is a well-known fact [7], and the loss of precision occurs already with boolean connectives. For example, let S be an alternating transition system with four states $Q = \{q_1, q_2, q_3, q_4\}$, and let $Q^\alpha = \{a_1, a_2, a_3\}$ be an abstract domain with three abstract states. Let the concretization function γ be given by $\gamma(a_1) = \{q_1, q_2\}$, $\gamma(a_2) = \{q_2, q_3\}$, and $\gamma(a_3) = \{q_3, q_4\}$. Let p be a proposition, with $[\![p]\!]_S = \{q_1, q_2\}$, and $[\![\bar{p}]\!]_S = \{q_3, q_4\}$. Note that $[p \vee \bar{p}]^{\alpha}_{S,E}$ is not maximal. In fact, even if $\gamma(a_2) \subseteq [\![p \vee \bar{p}]\!]_S$, we have $a_2 \notin [p \vee \bar{p}]^{\alpha}_{S,E}$

Abstract LTL control. The formulas of *linear-time temporal logic* LTL are defined inductively by the grammar

$$\varphi \;::=\; p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \mathcal{U} \varphi_2,$$

for propositions p in some set Π^L of propositions. Formulas are evaluated over *traces* in the standard way [20]. From these formulas, we define the formulas $\Diamond p = true \mathcal{U} p$ and $\Box p = \neg\Diamond\neg p$ as usual. Player 1 can *control the state q of an alternating transition system S for the* LTL *formula* φ if player 1 has a strategy f_1 such that for all strategies f_2 of player 2, every trace $\rho \in \mathcal{L}_{f_1,f_2}(q)$ satisfies the formula φ. The LTL *control problem* asks, given an alternating transition system S and an LTL formula φ, which states of S can be controlled by player 1 for φ. The LTL *controller-synthesis problem* asks, in addition, for the construction of witnessing strategies. The alternating-time μ-calculus can express controllability of LTL formulas [1], that is, for each LTL formula φ, there is an equivalent $A\mu$ formula ψ such that for all alternating transition systems S, player 1 can control a state q of S for φ iff $q \in [\![\psi]\!]_S$. For example, the $A\mu$ formula $\nu X \mu Y.(p \wedge \langle\!\langle 1 \rangle\!\rangle \bigcirc X) \vee \langle\!\langle 1 \rangle\!\rangle \bigcirc Y$ holds at a state if player 1 has a strategy to enforce computations in which the observable p occurs infinitely often, and this is equivalent to the LTL control requirement $\Box\Diamond p$. Thus, an algorithm for model checking $A\mu$ can be used to solve the LTL control problem. In particular, we can use algorithm ModelCheck of Figure 1 to solve the LTL control problem.

Given an LTL requirement φ and the abstraction of the collecting semantics of an alternating transition system S, we can solve the LTL control problem on the abstract system in the following way. We construct an $A\mu$ formula ψ equivalent to φ, and use the abstract model checking algorithm AbsModelCheck to compute the set of abstract states $[\![\psi]\!]^{\alpha}_S$. From the soundness of the abstract model checking algorithm, we can conclude that player 1 can control φ from all concrete states in $\hat{\gamma}([\![\psi]\!]^{\alpha}_S)$. Moreover, from the result of the abstract model checking algorithm, one can derive a controller for player 1 in the concrete system [28, 17].

Completeness of abstract model checking of games. A necessary and sufficient characterization of completeness is provided by considering the *alternating bisimilarity relation* [2] on the state space of an alternating transition system. A binary relation $\cong \subseteq Q \times Q$ on the states of an alternating transition system is an *alternating bisimulation* if $q \cong r$ implies the following three conditions:

(1) $\pi(q) = \pi(r)$.
(2) For every set $T \in \delta_1(q)$ there exists a set $T' \in \delta_1(r)$ such that for every set $R' \in \delta_2(r)$ there exists a set $R \in \delta_1(q)$ such that if $(T \cap R) = \{q'\}$ and $(T' \cap R') = \{r'\}$ then $q' \cong r'$.

(3) For every set $T' \in \delta_1(r)$ there exists a set $T \in \delta_1(q)$ such that for every set $R \in \delta_2(q)$ there exists a set $R' \in \delta_1(r)$ such that if $(T \cap R) = \{q'\}$ and $(T' \cap R') = \{r'\}$ then $q' \cong r'$.

Two states q and r are *alternating bisimilar*, denoted $q \cong^B r$, if there is an alternating bisimulation $\cong$ such that $q \cong r$. Let $Q^{\cong^B}$ denote the set of alternating bisimilarity classes of states, and let $q_1^{\cong^B}, q_2^{\cong^B}, \ldots$ refer to classes in $Q^{\cong^B}$. In [2], it is shown that two states on an alternating transition system satisfy the same alternating-time μ-calculus formulas iff they are alternating bisimilar. Using this characterization, we can show that if the abstract model checking algorithm is complete, then for each alternating bisimilarity class $q^{\cong^B} \in Q^{\cong^B}$ there is a set of abstract states whose concretization is exactly the class $q^{\cong^B}$. The proof is by contradiction; if not, we can either find an $A\mu$ formula that can distinguish two concrete states that are in the concretization of the same abstract state, or show that the concretization of all the abstract states is strictly included in the set of concrete states implying that the abstract interpretation cannot be complete. This shows that the abstract domain and the concretization function must refine the alternating bisimilarity relation for the abstract model checking algorithm to be complete. Moreover, by induction on the structure of formulas we can prove the converse: if the set of abstract states refine the alternating bisimilarity classes then the abstract model checking algorithm is complete.

Theorem 3. Completeness of AbsModelCheck. *The abstract model checking algorithm* AbsModelCheck *is (sound and) complete on an alternating transition system S with state space Q if and only if the abstract domain Q^α and the concretization function γ satisfy that for every alternating bisimilar class $q^{\cong^B} \in Q^{\cong^B}$, there exists $\sigma^\alpha \subseteq Q^\alpha$ such that $\hat{\gamma}(\sigma^\alpha) = q^{\cong^B}$.*

Thus, to achieve completeness, each alternating bisimilarity class should be the image of a set of abstract states under the concretization function. In general, the abstract model checking algorithm is sound, but not necessarily complete. This means that if an $A\mu$ property fails to hold on the abstract system, we cannot conclude that it does not hold in the concrete system. In order to disprove a property, we have to check if the negation of the property holds on the abstract system. Of course, neither the property nor its negation may hold at a state, in which case we have to refine our abstraction. Moreover, the abstract interpretation can be used to produce both under and overapproximations of a region satisfying a formula (to construct an overapproximation of a formula, we compute an underapproximation of the negation of the formula by concretizing the abstract region returned by the abstract model checking algorithm, and take the complement of the resulting set). Using techniques in [19, 10], this can give more precise approximations of a region where the formula holds.

4 Multi-process Programs: Concrete and Collecting Semantics

While the theory in the previous sections provides ways to model check an abstract system via its abstract collecting semantics, the abstract collecting semantics is derived from the (concrete) collecting semantics. In practice it is often preferable to be able to compute the abstract collecting semantics directly from the syntax of a program [7, 14, 25]. We introduce a simple programming language that is able to model concurrent games and show how to construct an abstraction of the collecting semantics directly from the program text.

Multi-process programs. We consider a simple programming formalisms based on Dijkstra's guarded command language [18]. Let X be a set of variables interpreted over some possibly infinite domains. We denote by $X' = \{x' \mid x \in X\}$ the set of variables obtained by priming each variable in X. A valuation v of the variables X is a function which maps each of the variables in X into a value in its domain. Denote by $\mathcal{V}_X$ the set of all possible valuations of the variables X. For any valuation v, we write v' the valuation obtained by priming each domain variable of v. For any predicate φ over the variables X and a valuation $v \in \mathcal{V}_X$, denote by $\varphi[\![v]\!]$ the truth value of the predicate with all the free variables interpreted according to the valuation v.

Given a set of program variables X, a Y*-action* ξ (guarded command) has the form $[\!]$ *guard* $\rightarrow$ *update*, where $Y \subseteq X$, the guard *guard* is a boolean predicate over X, and the update relation *update* is a boolean predicate over $X \cup Y'$. We also write $guard_\xi$ and $update_\xi$ to represent the guard and update relation of ξ respectively. Given a valuation $v \in \mathcal{V}_X$, the action ξ is said to be *enabled* if the guard $guard_\xi[\![v]\!]$ evaluates to true. We assume that for every Y-action, the update relation is functional, that is, for all valuation functions $v \in \mathcal{V}_X$, there is exactly one valuation function $u \in \mathcal{V}_Y$ such that $update_\xi[\![v \cup u']\!]$ holds. A Y*-process* Φ is a finite set of Y-actions. We say that the set of variables Y is *the controlled variables* of Φ. We require that for any valuation of the program variables, at least one action in the process is enabled.

To model two-player games, we partition the program variables X into two disjoint sets X_1, X_2 , i.e., $X = X_1 \uplus X_2$. Intuitively, X_i contains the variables updated by player i. A *program* $P = (\Phi_1, \Phi_2)$ over X is a pair of processes such that Φ_i is an X_i-process. Each process of the program can be defined by composing smaller processes, each of which controls a subset of the variables. For two processes Φ_1 and Φ_2 with disjoint controlled variables, their *composition* $\Phi_1 \| \Phi_2$ is the process $\Phi = \{[\!]\ guard_\xi \wedge guard_\eta \rightarrow update_\xi \wedge update_\eta \mid \xi \in \Phi_1 \wedge \eta \in \Phi_2\}$.

Example 1. Consider the program in Figure 3, depicting a protocol converter that operates between a lossy sender implementing an alternating-bit protocol, and a lossless receiver. The lossy sender *Sender* communicates with the converter through a pair of two-phase handshake lines, namely *sendreq* and *recvreq*. It signals the sending of a new message by inverting the variable *sendreq*, and

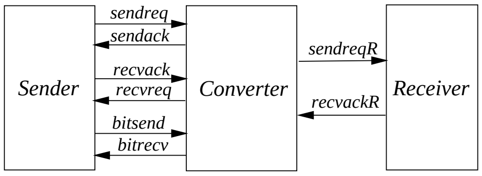

Fig. 2. An lossy sender speaking the alternating bit protocol communicating with a two-phase receiver through a protocol converter. The converter is to be synthesized.

expects an acknowledgement from the converter at a later time. It also sends a bit *bitsend* to the converter together with the message it sent. (The actual message is not modeled.) The sender is lossy in the sense that it may not signal the converter that a new message has been sent (by not inverting the variable *sendreq*), and it may just drop the acknowledgement sent by the converter (by inverting the variable *recvack*). If there is a loss, the variable *drop* will be set to *true*. The *Receiver* simply checks if there is a new message sent by the converter, and returns an acknowledgement after it has consumed the message. The *Monitor* process counts the number of messages received and sent by the converter. In this program Φ_2 is the composition *Sender*‖*Receiver*‖*Monitor*. The process Φ_1 is the *most* nondeterministic process which set the variables *sendack*, *sendreq*R, *recvreq*, and *bitrecv* to arbitrary values.

Concrete semantics of programs. The *concrete interpretation* of a program $P = \{\Phi_1, \Phi_2\}$ over the program variables X is the alternating transition system $S_P = \langle \Sigma, Q, \Delta, \Pi, \pi \rangle$, where (i) $\Sigma = \{1, 2\}$ (the two players of the program P). (ii) $Q = \mathcal{V}_X$, i.e., a state of S_P is a valuation of the program variables. (iii) $\Delta = (\delta_1, \delta_2)$ with $\delta_i : \mathcal{V}_X \to 2^{2^{\mathcal{V}_X}}$ is the transition function of player i and is defined as $\delta_i(v) = \left\{u \mid \xi \in \Phi_i \wedge guard_\xi[\![v]\!] \wedge update_\xi[\![v \cup u']\!]\right\}$. (iv) Π is a set of boolean predicates over the program variables X. (v) π maps an element p of Π to the set of states that satisfy p. Intuitively, for $i \in \{1, 2\}$, player i controls the actions in Φ_i. A program is run in steps. At each step, player 1 chooses an enabled action $\xi \in \Phi_1$, and updates the values of the variables in X_1 according to the predicate $update_\xi$. Independently and simultaneously, player 2 chooses an enabled action $\eta \in \Phi_2$ and updates the values of the variables in X_2 according to $update_\eta$.

Collecting semantics of programs. Given a program P over the set of variables X, let $\langle\!| \cdot |\!\rangle$, $CPre_I$, $UPre_I$ (for $I = 1, 2, \{1, 2\}$) be the collecting semantics of S_P. We construct predicate formulas that represent the collecting semantics of the program P. Let R be a predicate over the variables X. We say R *represents* a state v of S_P if $R[\![v]\!]$ is true. We use predicates on X to denote the set of states they represent, and naturally extend operations defined on sets of states to predicates. Given a predicate R representing a set of states of S_P, we define

$$
\begin{array}{lll}
\| \ pc = s_0 & \rightarrow & \begin{array}{l} drop' := \mathit{false}; pc' := s_1; \\ bitsend' := \neg bitsend; \end{array} \\
\| \ pc = s_1 \wedge (recvack = recvreq) & \rightarrow & \begin{array}{l} drop' := \mathit{false}; pc' := s_1; \\ sendreq' := \neg sendreq; \end{array} \\
\| \ pc = s_1 \wedge (recvack = recvreq) & \rightarrow & \begin{array}{l} drop' := \mathit{true}; pc' := s_1; \\ sendreq' := sendreq; \end{array} \\
\| \ pc = s_1 \wedge (recvack \neq recvreq) \wedge (bitsend = bitrecv) & \rightarrow & \begin{array}{l} drop' := \mathit{false}; pc' := s_0; \\ recvack' := \neg recvack; \end{array} \\
\| \ pc = s_1 \wedge (recvack \neq recvreq) \wedge (bitsend \neq bitrecv) & \rightarrow & \begin{array}{l} drop' := \mathit{false}; pc' := s_1; \\ recvack' := \neg recvack; \end{array} \\
\| \ pc = s_1 \wedge (recvack \neq recvreq) & \rightarrow & \begin{array}{l} drop' := \mathit{true}; pc' := s_1; \\ recvack' := \neg recvack; \end{array}
\end{array}
$$

Sender

$$
\begin{array}{l}
\| \ (sendackR \neq sendreqR) \rightarrow sendackR' := \neg sendackR; \\
\| \ (sendackR = sendreqR) \rightarrow sendackR' := sendackR;
\end{array}
$$

Receiver

$$
\begin{array}{lll}
\| \ sendreq \neq sendack & \rightarrow & count' := count + 1; \\
\| \ sendreqR \neq sendackR & \rightarrow & count' := count - 1; \\
\| \ (sendreqR = sendackR) \wedge (sendreq = sendack) & \rightarrow & count' := count;
\end{array}
$$

Monitor

Fig. 3. An alternating-bit sender, a simple receiver and a monitor

the player-1 controllable predecessor predicate $\Psi_P^{CPre_1}(R)$ as:

$$\Psi_P^{CPre_1}(R) \equiv \bigvee_{\xi \in \Phi_1} \left(guard_\xi \wedge \bigwedge_{\eta \in \Phi_2} \left(guard_\eta \rightarrow \forall X' \,.(update_\xi \wedge update_\eta \rightarrow R') \right) \right)$$

where R' is the predicate obtained by substituting each free variable in the predicate R by its primed counterpart. Similarly, we define the player-1 uncontrollable predecessor predicate $\Psi_P^{UPre_1}(R)$ as:

$$\Psi_P^{UPre_1}(R) \equiv \bigwedge_{\xi \in \Phi_1} \left(guard_\xi \rightarrow \bigvee_{\eta \in \Phi_2} \left(guard_\eta \wedge \exists X' \,.(update_\xi \wedge update_\eta \wedge R') \right) \right)$$

The other predicates, $\Psi_P^{CPre_I}(R)$ and $\Psi_P^{UPre_I}(R)$ for $I = 2, \{1,2\}$, can be defined similarly. The following proposition states that the predicates constructed above exactly coincide with the collecting semantics of the program P.

Proposition 3. *The computed formulas $\langle \Psi_P^{CPre_I}(R), \Psi_P^{UPre_I}(R) \rangle$ from the program P are equivalent to the collecting semantics of the alternating transition system S_P of the program P, that is, for every state v of S_P, and every predicate R representing the set of states σ, we have $v \in CPre_I(\sigma)$ iff $\Psi_P^{CPre_I}(R)[\![v]\!]$, and $v \in UPre_I(\sigma)$ iff $\Psi_P^{UPre_I}(R)[\![v]\!]$.*

5 Abstract Interpretation of Multi-process Programs with respect to Game Properties

In this section we show two methods of computing an approximation of the abstraction of the collecting semantics directly from the program text.

Abstract interpretation via domain abstraction. In abstract interpretation via domain abstraction, we are given for each program variable $x \in X$ a fixed abstract domain over which the variable is interpreted. Let $\mathcal{V}_X^\alpha$ be the set of abstract valuations of X, i.e., valuations of the variables over their abstract domains. Let $\gamma : \mathcal{V}_X^\alpha \to 2^{\mathcal{V}_X}$ be a concretization function mapping an abstract valuation (an abstract state) to the set of concrete valuations (the set of concrete states). To derive sound abstractions of the collecting semantics from the program text, we introduce, for $\xi \in \Phi_i$ and $i \in \{1,2\}$, predicates $guard_\xi^C$, $guard_\xi^F$ over the variables X, and predicates $update_\xi^C$, $update_\xi^F$ over the variables $X \cup X'$. These predicates represent over (or *free*) and under (or *constrained*) approximations for the guards and update relations of the program. We define formally the free and constrained versions of the guard and update relations as follows.

- The predicate $guard_\xi^F$ is a *free abstract interpretation of* $guard_\xi$ iff for all $u \in \mathcal{V}_X^\alpha$, $guard_\xi^F[\![u]\!]$ is true if there exists a concrete valuation v in the concretization of u on which $guard_\xi[\![v]\!]$ evaluates to true; i.e., $\forall u \in \mathcal{V}_X^\alpha.\ \exists v \in \gamma(u).\ (guard_\xi[\![v]\!] \to guard_\xi^F[\![u]\!])$
- The predicate $update_\xi^F$ is a *free abstract interpretation of* $update_\xi$ iff for all $u_1, u_2 \in \mathcal{V}_X^\alpha$, $update_\xi^F[\![u_1, u_2']\!]$ evaluates to true if there exists a pair of concrete valuations (v_1, v_2) represented by (u_1, u_2) on which $update_\xi$ evaluates to true; i.e., $\forall u_1, u_2 \in \mathcal{V}_X^\alpha.\ \exists v_1 \in \gamma(u_1), v_2 \in \gamma(u_2).\ (update_\xi[\![v_1 \cup v_2']\!] \to update_\xi^F[\![u_1 \cup u_2']\!])$.
- The predicate $guard_\xi^C$ is a *constrained abstract interpretation of* $guard_\xi$ iff $guard_\xi^C$ evaluates to true on u if all concrete valuations v in the concretization of u make $guard_\xi$ evaluates to true; i.e., $\forall u \in \mathcal{V}_X^\alpha.\ \forall v \in \gamma(u).\ (guard_\xi^C[\![u]\!] \to guard_\xi[\![v]\!])$
- The predicate $update_\xi^C$ is a *constrained abstract interpretation of* $update_\xi$ iff for all $u_1, u_2 \in \mathcal{V}_X^\alpha$, $update_\xi^C[\![u_1, u_2']\!]$ evaluates to true if all pairs of concrete valuations (v_1, v_2) represented by (u_1, u_2) make $update_\xi$ true; i.e., $\forall u_1, u_2 \in \mathcal{V}_X^\alpha.\ \forall v_1 \in \gamma(u_1), v_2 \in \gamma(u_2).\ (update_\xi^C[\![u_1 \cup u_2']\!] \to update_\xi[\![v_1 \cup v_2']\!])$.

We now show how to approximate the abstraction of the collecting semantics directly from the program text by nonstandard interpretations of the guard and update predicates of the program P. For $I = 1, 2, \{1,2\}$ and a set of abstract states represented by the predicate R^α, we construct the parameterized formulas $\Psi_P^{CPre_I^\alpha}(R^\alpha)$ and $\Psi_P^{UPre_I^\alpha}(R^\alpha)$ from $\Psi_P^{CPre_I}(R)$ and $\Psi_P^{UPre_I}(R)$ as follows: we replace every predicate that appears positively by its constrained version and every predicate that appears negatively by its free version. For example, the formula $\Psi_P^{CPre_1^\alpha}(R^\alpha)$ is as follows:

$$\bigvee_{\xi\in\Phi_1}\left(guard^C_\xi \wedge \bigwedge_{\eta\in\Phi_2}\left(guard^F_\eta \rightarrow \forall X'.\,(update^F_\xi \wedge update^F_\eta \rightarrow R^{\alpha\prime})\right)\right)$$

We similarly obtain abstractions of the other formulas. Since we are always approximating conservatively, the following proposition holds.

Proposition 4. *Domain-based abstract interpretation produces approximations; i.e., for every predicate R^α representing the set of abstract states σ^α, we have $\Psi_P^{CPre_1^\alpha}(R^\alpha) \mathrel{\hat{\preceq}} CPre_1^\alpha(\sigma^\alpha)$, and $\Psi_P^{UPre_1^\alpha}(R^\alpha) \mathrel{\hat{\preceq}} UPre_1^\alpha(\sigma^\alpha)$.*

Abstract interpretation via predicate abstraction. Domain-based abstraction computes abstractions from the program text compositionally, and may often produce crude approximations. An alternative method of constructing abstractions of the collecting semantics from the program text is *predicate abstraction*. In predicate abstraction, the abstract state is represented by a set of propositional predicates (called *abstraction predicates*) [24, 15] over the variables X of a program. An abstract state assigns truth values to each abstraction predicate. The concretization function γ maps an abstract state to the set of concrete states that satisfy the predicates.

The abstraction of the collecting semantics under predicate abstraction may be approximated directly from the program text. Whereas we can still construct the abstract predicates compositionally by substituting for each concrete predicate (*guard* or *update*) a conjunction of the abstraction predicates that implies (or is implied by) the concrete predicate, very often this leads to an overly crude abstraction. Therefore we sacrifice compositionality in order to obtain a more precise approximation of the abstraction of the collecting semantics. The approximation of the abstraction of the collecting semantics is derived as follows (we show the computation explicitly for $\Psi_P^{CPre_1^\alpha}$, the other operators can be constructed similarly). For each pair $\xi \in \Phi_1$, $\eta \in \Phi_2$ of moves of player 1 and player 2, we compute the formula

$$\chi_{\xi\eta}(R^\alpha) \;=\; guard_\xi \wedge \left(guard_\eta \rightarrow \forall X'.\,(update_\xi \wedge update_\eta \rightarrow \hat{\gamma}(R^\alpha)')\right)$$

Thus, the predicate $\chi_{\xi\eta}(R^\alpha)$ holds at a concrete state v if we reach $\hat{\gamma}(R^\alpha)$ from v when player 1 plays move (action) ξ and player 2 plays move η. We replace each predicate $\chi_{\xi\eta}(R^\alpha)$ by a boolean combination of abstraction predicates $\chi^\alpha_{\xi\eta}(R^\alpha)$ which is implied by $\chi_{\xi\eta}(R^\alpha)$ to obtain a sound approximation of the abstraction of $\chi_{\xi\eta}(R^\alpha)$. Finally, $\Psi_P^{CPre_1^\alpha}(R^\alpha)$ is obtained by existentially quantifying over the moves of player 1 and universally quantifying over the moves of player 2 from the predicates $\chi^\alpha_{\xi\eta}$; formally, $\Psi_P^{CPre_1^\alpha}(R^\alpha) \;=\; \bigvee_{\xi\in\Phi_1}\bigwedge_{\eta\in\Phi_2}\chi^\alpha_{\xi\eta}$. Thus $\Psi_P^{CPre_1^\alpha}(R^\alpha)$ is true at an abstract state if there is a move that player 1 can make, such that for all moves that player 2 makes, the game ends up in R^α. The other

$$
\begin{array}{lll}
\| \neg reset\_rt \wedge pc = RESET \wedge conn & \rightarrow & pc' := WAITC; req'_{ID} := ConnReq; \\
\| \neg reset\_rt \wedge pc = WAITC \wedge ack = (ID,1) & \rightarrow & pc' := CONN; req'_{ID} := NoReq; \\
\| \neg reset\_rt \wedge pc = WAITC \wedge ack = (ID,0) & \rightarrow & pc' := RESET; req'_{ID} := NoReq; \\
\| \neg reset\_rt \wedge pc = CONN \wedge disc & \rightarrow & pc' := WAITD; req'_{ID} := DiscReq; \\
\| \neg reset\_rt \wedge pc = WAITD \wedge ack = (ID,1) & \rightarrow & pc' := RESET; req'_{ID} := NoReq; \\
\| \neg reset\_rt \wedge pc = WAITD \wedge ack = (ID,0) & \rightarrow & pc' := RESET; req'_{ID} := DiscReq; \\
\| \ reset\_rt & \rightarrow & pc' := RESET; req'_{ID} := NoReq;
\end{array}
$$

Fig. 4. A remote whose id is ID.

$$
\begin{array}{lll}
\| \neg reset\_bs \wedge req_{id} = ConnReq \wedge \neg register[id] & \rightarrow & register'[id] := true; ack'_{id} := 1; \\
\| \neg reset\_bs \wedge req_{id} = ConnReq \wedge register[id] & \rightarrow & ack'_{id} := 0; \\
\| \neg reset\_bs \wedge req_{id} = DiscReq \wedge \neg register[id] & \rightarrow & ack'_{id} := 0; \\
\| \neg reset\_bs \wedge req_{id} = DiscReq \wedge register[id] & \rightarrow & register'[id] := false; ack'_{id} := 1; \\
\| \ reset\_bs \wedge register[id] & \rightarrow & register'[i] := false; ack'_{id} := 0
\end{array}
$$

Fig. 5. A process of the base station. The complete base station is the composition of the above processes for each remote id.

formulas are obtained similarly. We call this computation the *predicate-based abstract interpretation* of programs. The following proposition holds because in the construction of the operators we have always taken sound approximations.

Proposition 5. *Predicate-based abstract interpretation produces approximations; i.e., for every predicate* R^α *representing the set of abstract states* σ^α*, we have* $\Psi_P^{CPre_1^\alpha}(R^\alpha) \hat{\preceq} CPre_1^\alpha(\sigma^\alpha)$*, and* $\Psi_P^{UPre_1^\alpha}(R^\alpha) \hat{\preceq} UPre_1^\alpha(\sigma^\alpha)$.

6 Two Examples

We illustrate the methods introduced in the previous sections through two practical verification examples.

A wireless communication protocol. This example is taken from the Two-Chip Intercom (TCI) project at the Berkeley Wireless Research Center [4]. The TCI network is a wireless local network which allows approximately 40 *remotes*, one for each user, to transmit voice with point-to-point and broadcast communication. The operation of the network is coordinated by a central unit called the base station which assigns channels to the users through the Time Division Multiplexing Access scheme.

We briefly describe a simplified model of the actual protocol used in TCI. The protocol operates as follows. Before any remote is operational, it has to register at the base station. A remote (Figure 4) has a state variable *pc*, which can be *RESET* or *CONN*. If the remote is in the *RESET* state, it can accept a connection request *conn* from the user. The remote in turn sends a connection request to the base station and waits for an acknowledgement. It moves to the *CONN* state if it receives a positive acknowledgement, or to the *RESET* state otherwise. Once the remote is in the *CONN* state, it can be disconnected by accepting a *disc* request from the user.

A base station (Figure 5) keeps track of the states of the remotes in the database *register*. If it receives a connection request *ConnReq* from a remote, it checks if the remote is already registered. If not, it registers the remote, and sends back a positive acknowledgement. If the remote is already registered, a negative acknowledgement is sent back. A similar process occurs if the remote wishes to disconnect. Both the remote and the base station has an external reset signal (*reset_rt* and *reset_bs*) that can reset the units to the reset state.

We consider this protocol for a system with a base station and an arbitrary number of remotes. A natural property that such a system should have is the following: for any remote, say $remote_1$, no matter what the base station and the other remotes do, there should be a way to connect to the base station. However, the original protocol contained a bug, which we found in the course of the verification. We found the bug by proving the opposite, i.e., no matter what this remote does, the base station and the other remotes are able to keep this remote out of the network. This can be written as (assuming the two players in our system are the remote $remote_1$, and the base station together with all other remotes, denoted by *Env*): $\varphi = \langle\!\langle Env, remote_1 \rangle\!\rangle \Diamond [\![remote_1]\!] \Box (pc_1 \neq CONN)$, where pc_1 is the state variable in $remote_1$.

We automatically detected the bug using automated model checking. Since the system was infinite state, we could not apply model checking directly, and we needed to use abstraction to reduce the problem to a finite instance. By symmetry, it suffices to show the bug for the first remote, $remote_1$. We prove the property as follows: for every $i \neq 1$ we abstract the variable pc_i of $remote_i$ into one value, say $\perp$. To check the property, we need to constrain the behavior of the remotes. We choose the simplest form of constrained predicates, namely *false*. In other words, these remotes simply deadlock and therefore they can be removed from our consideration. For variables of $remote_1$ and the base station, we use the trivial (identity) abstraction. The property is proved by model checking on this abstract system in our model checker MOCHA [3].

To understand why this bug occurs, consider the following scenario. Suppose the remote has already been connected to the base station. The user *resets* the remote and it returns to the reset state immediately. Now suppose the user instructs the remote to send a connection request to the base station. The base station, however, still has the remote being registered and therefore it simply refuses to grant any further connection permission. Note also that the property violated is a true game property: it cannot be expressed directly in a nongame logic like CTL. Thus, both abstraction and game properties were necessary in the automatic detection of this bug.

Protocol converter synthesis. As an example of predicate abstraction, we consider the automatic synthesis of a protocol converter operates between the lossy sender *Sender* and a the lossless receiver *Receiver* in Example 1. We require our protocol converter *cv* to satisfy the following property (note that the converter is synthesized from the the process Φ_1):

$$\langle\langle cv \rangle\rangle \Box((count = 0 \vee count = 1) \wedge (\Box\Diamond\neg drop \Rightarrow \Box\Diamond(sendreqR \neq sendackR)))$$

This formula specifies that the converter has a strategy that ensures the following two properties. First, the difference between the number of messages received and sent by the converter has to be either 0 or 1. Second, if the lossy sender is fair, there should be an infinite number of messages received by the receiver.

The abstract predicates used are $pc = s_0$, $pc = s_1$, $(sendreq = sendack)$, $(recvreq = recvack)$, $(bitsend = bitrecv)$, $(sendreqR = sendackR)$ and $drop = true$. Moreover, we abstract the domain of the variable $count$ to $\{0, 1, \perp\}$ where the abstract values 0 and 1 represent the values 0 and 1 respectively, and the abstract value $\perp$ represents all other values. Using these predicate and domain abstractions, we are able to check that a converter that meets the requirement exists. Using methods in [28, 17], the actual converter can be synthesized.

References

1. R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 100–109. IEEE Computer Society Press, 1997.
2. R. Alur, T.A. Henzinger, O. Kupferman, and M.Y. Vardi. Alternating refinement relations. In D. Sangiorgi and R. de Simone, editors, *CONCUR 97: Concurrency Theory*, Lecture Notes in Computer Science 1466, pages 163–178. Springer-Verlag, 1998.
3. R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. MOCHA: modularity in model checking. In *CAV 98: Computer-aided Verification*, Lecture Notes in Computer Science 1427, pages 521–525. Springer-Verlag, 1998.
4. *Berkeley Wireless Research Center*. http://bwrc.eecs.berkeley.edu.
5. E.M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks using abstraction and regular languages. In *CONCUR 95: Concurrency Theory*, Lecture Notes in Computer Science 962, pages 395–407. Springer-Verlag, 1995.
6. E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages*, pages 343–354. ACM Press, 1992.
7. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual Symposium on Principles of Programming Languages*. ACM Press, 1977.
8. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2/3):103–179, 1992.
9. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
10. P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6(1):69–95, 1999.
11. P. Cousot and R. Cousot. Temporal abstract interpretation. In *Proceedings of the 27th Annual Symposium on Principles of Programming Languages*, pages 12–25. ACM Press, 2000.

12. D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking.* PhD thesis, Eindhoven University of Technology, The Netherlands, 1996.
13. D.R. Dams, R. Gerth, G. Döhmen, R. Herrmann, P. Kelb, and H. Pargmann. Model checking using adaptive state and data abstraction. In *CAV 94: Computer-Aided Verification*, Lecture Notes in Computer Science 818, pages 455–467. Springer-Verlag, 1994.
14. D.R. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.
15. S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *CAV 99: Computer-aided Verification*, Lecture Notes in Computer Science 1633, pages 160–171. Springer-Verlag, 1999.
16. L. de Alfaro, T.A. Henzinger, and O. Kupferman. Concurrent reachability games. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, pages 564–575. IEEE Computer Society Press, 1998.
17. L. de Alfaro, T.A. Henzinger, and R. Majumdar. Symbolic algorithms for infinite-state games. Technical report, University of California, Berkeley, 2000.
18. E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.
19. D.L. Dill and H. Wong-Toi. Verification of real-time systems by successive over-and underapproximation. In *CAV 95: Computer-aided Verification*, Lecture Notes in Computer Science 939, pages 409–422. Springer-Verlag, 1995.
20. E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier Science Publishers, 1990.
21. R. Giacobazzi, F. Ranzato, and F. Scozzari. Complete abstract interpretations made constructive. In *MFCS 98: Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 1450, pages 366–377. Springer-Verlag, 1998.
22. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretation complete. *Journal of the ACM*, 2000. To appear.
23. S. Graf. Verification of a distributed cache memory by using abstractions. In *CAV 94: Computer-aided Verification*, Lecture Notes in Computer Science 818, pages 207–219. Springer-Verlag, 1994.
24. S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In *CAV 97: Computer-aided Verification*, Lecture Notes in Computer Science 1254, pages 72–83. Springer-Verlag, 1997.
25. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:11–44, 1995.
26. D.E. Long. *Model checking, abstraction, and compositional verification.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1993.
27. K.L. McMillan. Verification of infinite state systems by compositional model checking. In *CHARME 99: Correct Hardware Design and Verification Methods*, Lecture Notes in Computer Science 1703, pages 219–233. Springer-Verlag, 1999.
28. P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete-event processes. *SIAM Journal of Control and Optimization*, 25(1):206–230, 1987.
29. P. Stevens. Abstract interpretation of games. In *Proceedings of the 2nd International Workshop on Verification, Model Checking and Abstract Interpretation*, 1998.