NICTA Technical Report NRL-8281

Predicate Abstraction for Reactive Synthesis

Adam Walker[§] Leonid Ryzhyk[§¶]
[§] NICTA and UNSW, Sydney, Australia
[¶] University of Toronto

***Abstract*—We present a predicate-based abstraction refinement algorithm for solving reactive games. We develop solutions to the key problems involved in implementing efficient predicate abstraction, which previously have not been addressed in game settings: (1) keeping abstractions concise by identifying relevant predicates only, (2) solving abstract games efficiently, and (3) computing and solving abstractions symbolically. We implemented the algorithm as part of an automatic device driver synthesis toolkit and evaluated it by synthesising drivers for several real-world I/O devices. This involved solving game instances that could not be feasibly solved without using abstraction or using simpler forms of abstraction.**

I. Introduction

Two-player games are a useful formalism for synthesis of reactive systems [17]. Many problems in electronic design automation [3], industrial automation [5], device driver development [19], etc., can be formalised as games. The resulting games often have very large state spaces and can not be efficiently solved using existing techniques.

Abstraction offers an effective approach to mitigating the state explosion. For example, in the model checking domain abstraction proved instrumental in enabling automatic verification of complex hardware and software systems [6], [7], [15]. The reactive synthesis community has also identified the key role of abstraction in tackling real-world synthesis problems; however most research in this area has so far been of theoretical nature [10], [14].

In this paper we present the first practical abstraction-refinement algorithm for solving games. Our algorithm is based on *predicate abstraction*, which proved to be particularly successful in model checking [13]. Predicate abstraction partitions the state space of the game based on a set of predicates, which capture essential properties of the system. States inside a partition are indistinguishable to the abstraction, which limits the maximal precision of solving the game achievable within the given abstraction. The abstraction is iteratively refined by introducing new predicates.

The key difficulty in applying predicate abstraction to games is to efficiently solve the abstract game arising at every iteration of the abstraction refinement loop. This requires computing the abstract *controllable predecessor* operator, which maps a set of abstract states, winning for one of the players, into the set of states from which the player can force the game into the winning set in one round of the game. This involves enumerating concrete moves available to both players in each abstract state, which can be prohibitively expensive.

We address the problem by further approximating the expensive controllable predecessor computation and refining the approximation when necessary. To this end, we introduce additional predicates that partition the set of actions available to the players into *abstract actions*. The controllable predecessor computation then consists of two steps: (1) computing abstract actions available in each abstract state, and (2) evaluating controllable predecessor over abstract states and actions.

The first step involves potentially expensive analysis of concrete transitions of the system and is therefore computed approximately. More specifically, solving the abstract game requires overapproximating moves available to one of the players, while underapproximating moves available to the other [14]. The former is achieved by allowing an abstract action in an abstract state if it is available in at least one corresponding concrete state, the latter allows an action only if it is available in all corresponding concrete states. We compute the overapproximation by initially allowing all actions in all states and gradually refining the abstraction by eliminating spurious actions. Conversely, we start with an empty underapproximation and add available actions as necessary.

We incorporated our predicate abstraction algorithm in the three-valued abstraction refinement framework of de Alfaro and Roy [10]. However, it can be readily adapted for use with other abstraction refinement methods, such as the counterexample-guided framework of Henzinger et al [14].

This paper makes three contributions:

1) We propose the first practical predicate-based abstraction refinement algorithm for two-player games.
2) We introduce a new type of refinement, which increases the precision of controllable predecessor computation without refining the abstract state space of the game. This approach avoids costly operations involved in solving the abstract game, approximating them with a sequence of light-weight operations performed on demand, leading to dramatically improved scalability.
3) We evaluate the algorithm by implementing it as part of the Termite driver synthesis toolkit [19] and using it to synthesise drivers for complex real-world devices. Our algorithm efficiently solves games with very large state spaces, which is impossible without using abstraction or using simpler forms of abstraction.

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

This research is supported by a grant from Intel Corporation.

II. Related work

Predicate abstraction has been extensively explored in automatic verification [13], including hardware [6] and software [7], [15] verification. In verification, given a set of abstract error states, we would like to overapproximate the set of predecessor states, from which the system may transition to one of the error states. To this end, one constructs an overapproximation of the abstract transition relation of the system, which relates a pair of abstract states if there exists a matching concrete transition between these two states [1]. De Alfaro et al. [9] pointed out that similar approach is not applicable to solving abstract games. In game settings, given a set of abstract goal states, we would like to compute its abstract controllable predecessor, i.e., the set of abstract states from which one of the players can force the game into the goal in one round. This fundamentally cannot be encoded as a relation over pairs of abstract states as, although the player may not be able to force the game into an individual abstract state, it may be able to force it into a subset of goal states. Therefore, instead of approximating the abstract transition relation of the game, we approximate its abstract controllable predecessor operator.

The three-valued abstraction refinement technique was first proposed as a method for CTL model checking [20] and was later adapted to games [9]. It was further developed by de Alfaro and Roy [10] into a form amenable to fully symbolic implementation. They present an instantiation of their method for a particular type of abstraction—*variable abstraction*. In the present paper, we combine their method with the more flexible predicate abstraction.

Counterexample-guided abstraction refinement (CEGAR) [8] is another method of constructing abstractions automatically. Henzinger et al. [14] present an adaptation of CEGAR to games. Similar to the three-valued abstraction framework of de Alfaro and Roy, their technique can be instantiated for different forms of abstraction. Dimitrova and Finkbeiner present an instantiation based on predicate abstraction [11], [12]. They focus on partial information and timed games, as opposed to perfect-information games with large state spaces, as we do in the present work. They report solving games with up to 2000 abstract states, whereas our case studies reported in Section VII required abstractions with up to 2^{33} abstract states.

III. Preliminary definitions

A two-player *game structure* $G = \langle S, L, I, \tau_1, \tau_2, \delta \rangle$ consists of a set of states S, a set of transition labels L, a set $I \subseteq 2^S$ of initial states, a partitioning of S into player-1 states τ_1 and player-2 states τ_2 ($\tau_1 \cap \tau_2 = \emptyset$, $\tau_1 \cup \tau_2 = S$), a transition function $\delta : (S, L) \to S$ associating with a state $s \in S$ and a label $l \in L$ a successor state $\delta(s, l)$. We refer to the opponent of player i as $\overline{i}$ ($\overline{1} = 2$, $\overline{2} = 1$).

The game proceeds in an infinite sequence of rounds, starting from an initial state. In each round, in state $s \in \tau_i$, player i chooses a label l and the game transitions to state $s' = \delta(s, l)$. We do not require the game to be strictly alternating, i.e., $s' \in \tau_{\overline{i}}$ is not generally true. The infinite sequence of states visited $(s_0, s_1, \ldots) \in S^\omega$ is called a *path*.

An *objective* $\Phi \subseteq S^\omega$ is a subset of state sequences of G. In this paper we are concerned with ω-regular objectives, i.e., objectives characterised by ω-regular languages. Two special cases of ω-regular objectives are *reachability objectives* that consist of all paths $s_0, s_1, \ldots$ that visit a target set T at least once: $\exists i.s_i \in T$ and *safety objectives* that consist of paths that stay in a safe set T forever: $\forall i.s_i \in T$.

A *strategy* for player i is a function $\pi_i : S^* \times \tau_i \to L$ that, in any player i state, associates the history of the game with a label to play. The set of initial states I and a player i strategy π_i determines a set $Outcomes_i(I, \pi_i)$ of paths $s_0, s_1, s_2, ...$ such that $s_0 \in I$ and $s_{k+1} = \delta(s_k, \pi_i(s_0, ..., s_k))$ when $s_k \in \tau_i$ and $s_{k+1} = \delta(s_k, l)$ for some l when $s_k \in \tau_{\overline{i}}$. Given an objective $\Phi \in S^\omega$ we say that state $s \in S$ is winning for player i if there is a strategy π_i such that $Outcomes_i(s, \pi_i) \subseteq \Phi$.

A. Symbolic games

We deal with *symbolic games* defined over a finite set of state variables X and a finite set of label variables Y in some theory. Each state $s \in S$ represents a valuation of variables X, each label $l \in L$ represents a valuation of variables Y. For a set Z of variables, we denote by $\mathcal{F}(Z)$ the set of propositional formulas in the underlying theory constructed from the variables in Z. Sets of states and transition relations of a symbolic game are represented by their characteristic formulas. In particular I, τ_1, τ_2 are given as formulas in $\mathcal{F}(X)$. The transition relation is specified as $\delta \in \mathcal{F}(X \cup Y \cup X')$, where $X' = \{x' \mid x \in X\}$ is the set of next-state variables. We refer to sets and their characteristic formulas interchangeably.

Example. We introduce our running example, where we aim to synthesise a software driver for an artificially trivial I/O device. The device contains 32 bits of non-volatile memory, which can be accessed from software via the data register. The task of the driver is to transfer a data value from the main memory to the device memory.

We set up a game between the driver (player 1) and the device (player 2). Device and driver internal state is modelled using state variables (Figure 1a). The player who makes the next move is determined by the value of the `bsy` flag inside the device. When the flag is set to `0`, the device remains idle and the driver performs a write to the data register. The argument of the write is modelled by the `val` label variable. The write operation flips the `bsy` flag to `1`. This triggers a device transition at the next round of the game, which copies the value in the data register to memory. The objective of the game on behalf of player 1 is to reach the target set $\mathtt{T} = (\mathtt{req} = \mathtt{mem})$, i.e., the device memory must store the requested value `req` (Figure 1c). We require that the game is winnable from any initial state, hence $\mathtt{I} = \top$. The winning strategy for player 1 in this example is to write the value of `req` in the first transition (by setting $\mathtt{val} = \mathtt{req}$), thus forcing the device to copy this value to memory at the second transition.

Figure 1b specifies the transition relation δ of the game in the form of variable update functions $\mathtt{x}' = \mathtt{t_x}(\mathtt{X}, \mathtt{Y})$, one for

var	type	description
state vars (X)		
mem	int32	device memory
dat	int32	data register
bsy	bool	device busy bit
req	int32	value to write to mem
label vars (Y)		
val	int32	value to write to dat

(a) Game variables

$\tau_1 = (\texttt{bsy} = \texttt{false})$ $\tau_2 = (\texttt{bsy} = \texttt{true})$ $\texttt{I} = \top$ $\texttt{T} = (\texttt{req} = \texttt{mem})$

(b) Turn functions, initial and target sets

$$\texttt{dat}' = \begin{cases} \texttt{val}, & \text{if } \neg\texttt{bsy} \\ \texttt{dat}, & \text{otherwise} \end{cases}$$

$$\texttt{bsy}' = \begin{cases} \texttt{true}, & \text{if } \neg\texttt{bsy} \\ \texttt{false}, & \text{if bsy} \end{cases}$$

$$\texttt{mem}' = \begin{cases} \texttt{dat}, & \text{if bsy} \\ \texttt{mem}, & \text{otherwise} \end{cases}$$

$$\texttt{req}' = \texttt{req}$$

(c) Variable update functions

a.var	predicate
state predicates	
σ_1	req = dat
σ_2	req = mem
untracked predicates	
ω_1	bsy = false
ω_2	req = 5
label predicates	
λ_1	val = req
λ_2	val = 5

(d) Abstract variables and corresponding predicates

Fig. 1: A driver synthesis problem encoded as a game

each variable $\texttt{x} \in \texttt{X}$. Consider the update function for bsy as an example. The variable switches between values true and false on each transition, thus enabling player 1 and player 2 in a round robin fashion. □

B. Controllable predecessor

Omega-regular games are often solved using the *controllable predecessor* operator. Player i controllable predecessor of set $\phi \subseteq S$ consists of all states from which i can force the game into ϕ in one transition. It is a union of player i states where there exists a winning transition to ϕ and player $\bar{i}$ states where all outgoing transitions terminate in ϕ.

$$\begin{aligned} Cpre_i(\phi) = &\tau_i \wedge \exists Y, X'.\ \delta \wedge \phi' \ \vee \\ &\tau_{\bar{i}} \wedge \forall Y, X'.\ \delta \rightarrow \phi' \end{aligned} \tag{1}$$

where ϕ' denotes the formula obtained from ϕ by replacing every $x \in X$ with x'.

We can compute the winning set of a reachability game by iterating the controllable predecessor operator starting from the target set T of the game. Using fix-point notation: $\textsc{Reach}(T, Cpre) = \mu X.Cpre(X) \vee T$, We pass the controllable predecessor operator as an argument to the REACH function, so that it can be used with multiple different versions of $Cpre$ introduced below.

C. Abstraction

An abstraction of a game structure G is a tuple $\langle V, \downarrow \rangle$, where V is a finite set of abstract states and $\downarrow : V \rightarrow 2^S$ is the *concretisation function*, which takes an abstract state and returns the possibly empty set of concrete states that the abstract state corresponds to. We require that $\bigcup_{v \in V} v\downarrow = S$ and $v_1\downarrow \cap v_2\downarrow = \emptyset$ for any v_1 and v_2, $v_1 \neq v_2$. In the case when $v\downarrow = \emptyset$ the abstract state v is said to be *inconsistent*. We extend the $\downarrow$ operator to sets of abstract states. For $U \subseteq V$: $U\downarrow = \bigcup_{u \in U} u\downarrow$.

Algorithm 1 Three-valued abstraction refinement for games.

Input: A game structure $G = \langle S, L, I, \tau_1, \tau_2, \delta \rangle$, a set of target states $T \subseteq S$, and an initial abstraction $\alpha = \langle V, \downarrow, Cpre_1^{m+}, Cpre_1^{M-} \rangle$ that is precise for T, I, and τ_i.
Output: *Yes* if $I \subseteq \textsc{Reach}(T, Cpre_1)$, and *No* otherwise.

```
1:  loop
2:      W^M ← Reach(T↑^M, Cpre_1^{M-})
3:      W^m ← Reach(T↑^m, Cpre_1^{m+})
4:      if I↑^M ⊆ W^M  return Yes
5:      else if I↑^M ⊈ W^m  return No
6:      else
7:          refined ← refineCpre(W^M)
8:          if (¬refined) refineAbstraction(W^M) endif
9:      end if
10: end loop
```

IV. Three-valued abstraction refinement

In this section we present a modified version of the three-valued abstraction refinement technique of de Alfaro and Roy [10]. To simplify the presentation, we focus on solving reachability games. De Alfaro and Roy present an extension of their method to arbitrary ω-regular games. This extension is directly applicable to the version of the algorithm presented here.

We start with defining two versions of the abstraction operator: the *may-abstraction* $\uparrow^m$ and the *must-abstraction* $\uparrow^M$. For a set of concrete states $T \subseteq S$: $T\uparrow^m = \{v \in V \mid v\downarrow \cap T \neq \emptyset\}$, $T\uparrow^M = \{v \in V \mid v\downarrow \subseteq T\}$. We say that abstraction is *precise* for a set $T \subseteq S$ if $(T\uparrow^m)\downarrow = (T\uparrow^M)\downarrow$.

Next, we define may and must versions of the abstract controllable predecessor operator:

$$Cpre_i^m(U) = Cpre_i(U\downarrow)\uparrow^m, \quad Cpre_i^M(U) = Cpre_i(U\downarrow)\uparrow^M \tag{2}$$

These operators have the property: $Cpre_i^M(U)\downarrow \subseteq Cpre_i(U\downarrow) \subseteq Cpre_i^m(U)\downarrow$, and hence $\textsc{Reach}(T\uparrow^M, Cpre_i^M)\downarrow \subseteq \textsc{Reach}(T, Cpre_i) \subseteq \textsc{Reach}(T\uparrow^m, Cpre_i^m)\downarrow$.

The abstract $Cpre_i^m$ and $Cpre_i^M$ operators are defined in terms of the concrete controllable predecessor $Cpre$. As these may not be possible to compute efficiently in practice, we introduce approximate versions, $Cpre_i^{m+}$ and $Cpre_i^{M-}$, such that for all $U \subseteq V$: $Cpre_i^m(U)\downarrow \subseteq Cpre_i^{m+}(U)\downarrow$ and $Cpre_i^{M-}(U)\downarrow \subseteq Cpre_i^M(U)\downarrow$. The definition of $Cpre_i^{m+}$ and $Cpre_i^{M-}$ is determined by each particular instantiation of the abstraction refinement scheme. We present our version of these operators in Section V-A.

Figure 2 illustrates the main idea of our approach, which is presented in algorithm 1. At every iteration, the algorithm computes the must-winning set W^M that underapproximates, and the may-winning set W^m that overapproximates the true winning set (lines 2–3). The algorithm terminates if the must-winning set contains the entire initial set or the may-winning set has shrunk beyond the initial set (lines 4–5). Otherwise, the algorithm refines the abstraction in a way that expands the must-winning set.

The key observation behind the refinement procedure is that candidate winning states can be found at the *may-must boundary* of the game, i.e., the set $Cpre_1^{m+}(W^M)\backslash W^M$, of all may-predecessors of the must-winning set. The boundary consists

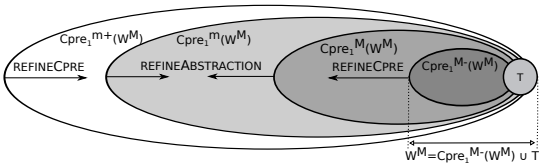

Fig. 2: Refining the may-must boundary. Arrows indicate how the two refinement functions change the boundary region.

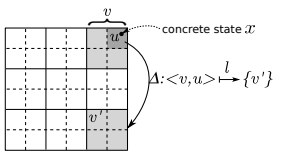

Fig. 3: Concrete state space partitioned into abstract states (solid lines) and untracked sub-states (dashed lines).

of three regions shown in Figure 2: (1) $Cpre_1^M(W_M) \setminus W_M$, (2) $Cpre_1^m(W^M) \setminus Cpre_1^M(W^M)$, and (3) $Cpre_1^{m+}(W^M) \setminus Cpre_1^m(W^M)$. The first and the third regions can be shrunk by increasing the precision of the $Cpre^{M-}$ and $Cpre^{m+}$ operators respectively. The second region can only be shrunk by refining the abstraction itself, i.e., partitioning abstract states into smaller regions.

These two types of refinement are performed in lines 7 and 8 of the algorithm. The REFINECPRE function computes a more precise version of the controllable predecessor operators. It returns $false$ iff no such refinement is possible, i.e., $Cpre^M(W_M) = Cpre^{M-}(W_M)$ and $Cpre^{m+}(W^M) = Cpre^m(W^M)$. The REFINEABSTRACTION function refines the abstract state space in a way that expands the set $Cpre^M(W^M)$ with at least one new abstract state.

Algorithm 1 differs from [10] in that it uses an additional type of refinement which refines the controllable predecessor operators without changing the abstract state space.

V. PREDICATE ABSTRACTION

We instantiate the three-valued abstraction refinement scheme for predicate abstraction. Consider a symbolic game $G = \langle S, L, I, \tau_1, \tau_2, \delta \rangle$ defined over state variables X and label variables Y. Let $\Sigma \subseteq \mathcal{F}(X)$ be a finite set of boolean predicates over state variables. We refer to Σ as *state predicates*. We introduce boolean variables $\vec{\sigma} = (\sigma_1 \dots \sigma_n)$ to represent values of predicates Σ. Given a boolean variable σ, $\|\sigma\|$ denotes its corresponding state or label predicate. $\|\vec{\sigma}\|$ denotes the vector of all state predicates in Σ.

The state space V of the abstract game is defined as $V = \mathbb{B}^n$, where each abstract boolean state vector $v \in V$ represents a truth assignment of variables $\vec{\sigma}$. The concretisation function $\downarrow$ from Section III-C can be expressed as: $v\downarrow = (\bigwedge_{i=1..n} \|\sigma_i\| = v_i)$, which maps an abstract state v into the set of concrete states such that each predicate in Σ evaluates to true or false depending on the value of the corresponding element of v.

Example. Consider an abstraction of the running example game induced by abstract variables σ_1, σ_2 and corresponding predicates: $\|\sigma_1\| = (\texttt{req} = \texttt{dat})$, $\|\sigma_2\| = (\texttt{req} = \texttt{mem})$. Consider an abstract state $\texttt{v} = (\texttt{true}, \texttt{false})$. We compute $\texttt{v}\downarrow = ((\texttt{req} = \texttt{dat}) = \texttt{true} \wedge (\texttt{req} = \texttt{mem}) = \texttt{false})$ or equivalently $\texttt{v}\downarrow = (\texttt{req} = \texttt{dat} \wedge \texttt{req} \neq \texttt{mem})$. Hence $\texttt{v}$ represents the set of all concrete states where conditions $(\texttt{req} = \texttt{dat})$ and $(\texttt{req} \neq \texttt{mem})$ hold for concrete state variables $\texttt{mem}$, $\texttt{req}$, and $\texttt{dat}$. □

We obtain the initial abstraction by extracting atomic predicates from expressions T, I, and τ_i, which guarantees that the abstraction is precise for T, I, and τ_i. While this property is not essential for our approach, we will rely on it to simplify the presentation of the algorithm.

A. Abstract controllable predecessors

Following the three-valued algorithm presented in Section IV, we would like to find an efficient way to compute over- and under-approximations $Cpre^{m+}$ and $Cpre^{M-}$ of the abstract controllable predecessor operators. Recall that computing $Cpre^m$ and $Cpre^M$ precisely is expensive, as it requires applying the controllable predecessor operator to the concrete transition relation δ. We approximate this costly computation by computing the controllable predecessor over the *abstract transition relation* instead. The abstract transition relation of the game is defined over boolean predicate variables and therefore can be manipulated much more efficiently than the concrete one.

We construct the abstract transition relation via efficient syntactic analysis of the concrete transition relation δ. We present the construction assuming that δ is given in the variable update form, as in Figure 1c. A similar construction is possible for specifications written in real-world hardware and software description languages.

For each state predicate in Σ, we compute the update function by replacing concrete variables in the predicate with their corresponding update functions. We then transform the resulting formula into a boolean combination of atomic predicates over concrete state and label variables.

Example. Let us compute the update function for abstract variable σ_1 (Figure 1d). Using update functions for $\texttt{req}$ and $\texttt{dat}$ variables (Figure 1c), we obtain: $\sigma_1' = (\texttt{req}' = \texttt{dat}') = \big(\neg(\texttt{bsy} = \texttt{false}) \wedge (\texttt{req} = \texttt{dat}) \vee (\texttt{bsy} = \texttt{false}) \wedge (\texttt{val} = \texttt{req})\big)$. This equation contains three atomic predicates: in addition to the existing predicate $\sigma_1 \leftrightarrow (\texttt{req} = \texttt{dat})$, it introduces new predicates $(\texttt{bsy} = \texttt{false})$ and $(\texttt{val} = \texttt{req})$. □

In the general case, the syntactically computed update function for a predicate may depend on existing state predicates in Σ as well as new predicates that are not yet part of the abstraction. The new predicates are partitioned into *untracked predicates* defined over concrete state variables (e.g., $\texttt{bsy} = \texttt{false}$ in the above example) and *label predicates* that involve at least one concrete label variable (e.g., $\texttt{val} = \texttt{req}$). The term "untracked predicate" indicates that these predicates are not part of the abstract state space of the game. Untracked predicates can be seen as partitioning abstract states in V into smaller *untracked sub-states*, as illustrated in Figure 3.

By substituting untracked and label predicates with fresh

boolean variables, $\vec{\omega}$ and $\vec{\lambda}$ respectively, we obtain the abstract transition relation Δ in the form:

$$\vec{\sigma}' = \Delta(\vec{\sigma}, \vec{\omega}, \vec{\lambda})$$

This syntactically computed transition relation contains two sources of imprecision. First, untracked variables $\vec{\omega}$ are not part of the abstract state space Σ and are therefore treated as external inputs. Second, not all abstract labels are available in all abstract states and hence not all transitions in Δ correspond to a feasible concrete transition. For example, given the set of predicates shown in Figure 1d, the abstract label $\lambda_1 = true, \lambda_2 = true$ is only available in concrete states that satisfy the condition $req = 5$. In general, given a state-untracked-label tuple $\langle v, u, l\rangle$, the abstract label l may be available in all, some, or none of the concrete states consistent with v and u.

We formalise this by introducing *consistency relations* C^m and C^M that over- and under-approximate available abstract labels. A state-untracked-label tuple $\langle v, u, l\rangle$ is *may-consistent* if the abstract label l is available in *at least one* concrete state consistent with v and u:

$$C^m(v, u, l) = \exists X, Y. \|\vec{\sigma}\| = v \wedge \|\vec{\omega}\| = u \wedge \|\vec{\lambda}\| = l. \quad (3)$$

The tuple $\langle v, u, l\rangle$ is *must-consistent* if l is available in *any* concrete state consistent with v and u:

$$C^M(v, u, l) = \forall X. ((\|\vec{\sigma}\| = v \wedge \|\vec{\omega}\| = u) \rightarrow \exists Y. \|\vec{\lambda}\| = l) \quad (4)$$

Computing C^m and C^M can be prohibitively expensive. Therefore we use approximations C^{m+} and C^{M-} such that $C^m \subseteq C^{m+}$ and $C^{M-} \subseteq C^M$. Initially we assign $C^{m+} = \top$ and $C^{M-} = \bot$. Approximations are refined lazily as part of the abstraction refinement process, as explained below.

We compute over- and under-approximations of the controllable predecessor operator by resolving the two sources of imprecision in favour of one of the players. In particular, we compute $Cpre_i^{m+}$ by (1) allowing player i to pick assignments to untracked predicates, (2) over-approximating consistent labels available to i, and (3) under-approximating consistent labels available to the opponent player $\bar{i}$:

$$\begin{aligned} Cpre_i^{m+}(\phi) = \exists \vec{\omega}.\ & \tau_i\uparrow^M \wedge \exists \vec{\lambda}, \vec{\sigma}'.((C^{m+} \wedge \Delta) \wedge \phi') \ \vee \\ & \tau_{\bar{i}}\uparrow^M \wedge \forall \vec{\lambda}, \vec{\sigma}'.((C^{M-} \wedge \Delta) \rightarrow \phi') \end{aligned} \quad (5)$$

This formula has a similar structure to the definition of the concrete controllable predecessor operator (1). It replaces the concrete transition relation δ with the abstract transition relation Δ restricted with consistency relations (C^{m+} and C^{M-}). In addition, it existentially quantifies untracked variables $\vec{\omega}$, i.e., an abstract state v is a may-predecessor of ϕ if at least one of its untracked sub-states is a may-predecessor of ϕ.

Dually, we compute $Cpre_i^{M-}$ by (1) allowing the opponent player $\bar{i}$ to pick values of untracked predicates, (2) under-approximating labels available to i and (3) over-approximating labels available to $\bar{i}$:

$$\begin{aligned} Cpre_i^{M-}(\phi) = \forall \vec{\omega}.\ & \tau_i\uparrow^M \wedge \exists \vec{\lambda}, \vec{\sigma}'.((C^{M-} \wedge \Delta) \wedge \phi') \ \vee \\ & \tau_{\bar{i}}\uparrow^M \wedge \forall \vec{\lambda}, \vec{\sigma}'.((C^{m+} \wedge \Delta) \rightarrow \phi') \end{aligned} \quad (6)$$

Equations (5) and (6) suggest two possible abstraction refinement tactics, which correspond to the two types of refinement used in Algorithm 1. First, we can refine C^{m+} and C^{M-} by removing spurious transitions from C^{m+} or adding new consistent transitions to C^{M-}. Such a refinement increases the precision of controllable predecessor computation without introducing new state predicates, which corresponds to the REFINECPRE operation in the algorithm. Second, we can add some of the untracked predicates to the set of state predicates Σ, thus reducing the imprecision introduced by treating them as external inputs. This refinement increases the precision of the abstraction, which corresponds to the REFINEABSTRACTION function in the algorithm.

In summary, we solve the abstract game by decomposing potentially expensive computations into three types of light-weight operations performed on demand, as required to improve the precision of the abstraction:

- Computing the abstract transition relation Δ via light-weight syntactic analysis of the concrete game
- Computing consistency relations C^{m+} and C^{M-} by iteratively identifying spurious and consistent transitions
- Solving the abstract game using abstract controllable predecessor operators (5) and (6)

The computational bottleneck in this method can arise either from having to perform an excessive number of refinements or if abstractions generated by the algorithm are too complex. Our refinement procedures, described below, are designed to avoid such situations by heuristically picking refinements that are likely to speed up the convergence of the algorithm.

B. REFINECPRE

Figure 4 illustrates the main idea of the consistency refinement algorithm. It shows an abstract state v (Figure 4a) at the may-must boundary whose untracked substates u_1, u_2, and u_3 have C^{m+}-consistent transitions to the must-winning set W^M, but none of these transitions is consistent with C^{M-}. The REFINECPRE algorithm attempts to precisely categorise these substates as must-winning or must-losing. In Figure 4b, the algorithm identifies the abstract transition $\langle v, u_1, l_1\rangle$ as spurious and eliminates it from C^{m+}, thus making the u_1 sub-state must-losing. Alternatively, it may detect that abstract transition $\langle v, u_2, l_2\rangle$ is available in all concrete states in u_2 and thus add this transition to C^{M-}, making the u_2 sub-state must-winning (Figure 4c). Finally, it may determine that abstract transition $\langle v, u_3, l_3\rangle$ is available in some, but not all, concrete states in u_3, i.e., $\langle v, u_3, l_3\rangle \in C^m \setminus C^M$. It then partitions u_3 into two or more subsets, exactly one of which has a C^{M-}-consistent transition to W^M, by introducing new untracked predicates (Figure 4d).

Algorithm 2 shows the pseudocode of REFINECPRE. Lines 3–6 compute the set of candidate tuples $\langle v, u, l\rangle \in C^m \setminus C^M$. Note that for player i states we consider may-consistent transition to W^M, whereas for player $\bar{i}$ states we consider spoiling transitions to $V \setminus W^M$. Line 9 picks a single refinement candidate $\langle v, u, l\rangle$ from the set. By construction we

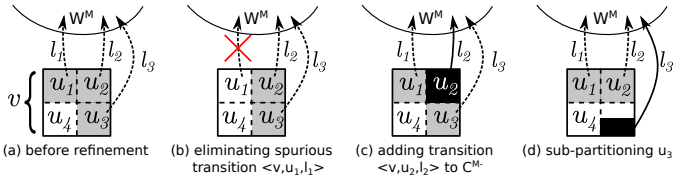

Fig. 4: Different types of consistency refinements. White, grey, and black background is used to mark respectively must-losing, may-winning, and must-winning untracked substates. Dashed and solid arrows show C^{m+} and C^{M-}-consistent abstract transitions.

Algorithm 2 Pseudocode of the REFINECPRE function

```
1: function RefineCpre(W^M)
2:     ▷ player i may-winning transitions
3:     T_i ← τ_i↑^M ∧ C^{m+} ∧ ¬C^{M-} ∧ ∀σ'.(Δ → (W^M)')
4:     ▷ player ī may-spoiling transitions
5:     T_ī ← τ_ī↑^M ∧ C^{m+} ∧ ¬C^{M-} ∧ ∃σ'.(Δ ∧ ¬(W^M)')
6:     T ← T_i ∨ T_ī
7:     if T = ⊥ then return false ▷ no refinement is possible
8:     else
9:         choose ⟨v, u, l⟩ ∈ T
10:        F ← (‖σ⃗‖ = v ∧ ‖ω⃗‖ = u ∧ ‖λ⃗‖ = l)
11:        if SatisfiableF) then
12:            A ← EliminateQuantifiers(∃Y.‖λ⃗‖ = l)
13:            Â ← replace atomic predicates in A with boolean
                   vars, introducing fresh vars when necessary
14:            C^{M-} ← C^{M-} ∨ (Â ∧ λ⃗ = l)
15:        else
16:            C^{m+} ← C^{m+} ∧ ¬UnsatCore(F)
17:        end if
18:        return true
19:    end if
20: end function
```

know that $\langle v, u, l\rangle \in C^{m+}$. Since C^{m+} is an overapproximation of C^m, we check whether $\langle v, u, l\rangle \in C^m$, i.e., whether v, u, and l satisfy equation (3). To this end, in line 11 we invoke a decision procedure for the underlying theory to check satisfiability of the formula: $(\|\vec{\sigma}\| = v \wedge \|\vec{\omega}\| = u \wedge \|\vec{\lambda}\| = l)$. If the formula is unsatisfiable, then $\langle v, u, l\rangle$ is a spurious transition that must be eliminated from C^{m+}. Furthermore, by extracting an unsatisfiable core of the formula, we obtain an inconsistent subset of its conjuncts $(\bigwedge \|\alpha_i\| = c_i)$, $\alpha_i \in \vec{\sigma} \cup \vec{\omega} \cup \vec{\lambda}$, which represents a potentially large set of similar spurious transitions. We eliminate all of these transitions from C^{m+} in line 16.

If, on the other hand, the formula is satisfiable, then there exists a concrete state-label pair consistent with $\langle v, u, l\rangle$. In this case we want to precisely characterise the set of states where label l is available, so that we can either add $\langle v, u, l\rangle$ to C^{M-} (as in Figure 4c) or refine it with additional untracked predicates (as in Figure 4d).

Line 12 computes the set of concrete states where abstract label l is available by performing quantifier elimination from formula $(\exists Y.\|\vec{\lambda}\| = l)$, resulting in a quantifier-free formula A over concrete state variables X. We assume that the underlying theory supports quantifier elimination, which is the case for many practically relevant theories, including the theory of fixed-size bit vectors supported by our tool. In line 13, the resulting formula A is decomposed into atomic predicates possibly introducing new untracked and label predicates. By replacing all atomic predicates in A with corresponding boolean variables, we obtain a formula $\hat{A}$ that describes the set of all state-untracked pairs must-consistent with the abstract label l. Line 14 refines C^{M-} with the set of newly discovered must-consistent transitions.

Example. Assume that in line 9 the algorithm picks a tuple $\langle \mathtt{v}, \mathtt{u}, \mathtt{l}\rangle$ where $\mathtt{l} = (\mathtt{true}, \mathtt{true})$. Line 12 performs quantifier elimination from the formula $\exists \mathtt{val}.(\|\lambda_1\| = \mathtt{true} \wedge \|\lambda_2\| = \mathtt{true}) = \exists \mathtt{val}.(\mathtt{val} = \mathtt{req} \wedge \mathtt{val} = 5) = (\mathtt{req} = 5)$. We have discovered a new predicate $\mathtt{req} = 5$ that must hold in states where abstract label $\mathtt{l}$ is available. We introduce a new untracked variable ω_2, $\|\omega_2\| = (\mathtt{req} = 5)$ and refine $\mathtt{C}^{\mathtt{M}-}$ with a new consistent transition: $\mathtt{C}^{\mathtt{M}-} \leftarrow \mathtt{C}^{\mathtt{M}-} \vee (\omega_2 \wedge \lambda_1 \wedge \lambda_2)$. □

Appendix A presents an important optimisation of the REFINECPRE function.

Algorithm 3 Pseudocode of REFINEABSTRACTION

```
1: function RefineAbstraction(W^M)
2:     U^M ← CpreU_1^{M-}(W^M) ∧ ¬W^M
3:     toPromote ← ω⃗ ∩ Support(ShortPrime(U^M))
4:     Promote(toPromote)
5: end function
```

C. REFINEABSTRACTION

The REFINEABSTRACTION function is invoked by the abstraction refinement algorithm when no further consistency refinements are possible. At this point, every untracked substate of the boundary region is either must-winning or must-losing, i.e., can be coloured white or black using notation of Figure 4. REFINEABSTRACTION promotes a subset of untracked predicates making sure that the winning region W^M expands after re-solving the game in line 2 of Algorithm 1.

Algorithm 3 shows the pseudocode of REFINEABSTRACTION. Line 2 computes all untracked boundary substates that are must-predecessors of W^M. Here, $CpreU^{M-}$ is the same as $Cpre^{M-}$ (Equation (6)), but without untracked variable quantification:

$$\begin{aligned} CpreU_i^{M-}(\phi) = &\tau_i{\uparrow}^M \wedge \exists \vec{\lambda}, \vec{\sigma'}.((C^{M-} \wedge \Delta) \wedge \phi') \ \vee \\ &\tau_{\bar{i}}{\uparrow}^M \wedge \forall \vec{\lambda}, \vec{\sigma'}.((C^{m+} \wedge \Delta) \rightarrow \phi') \end{aligned}$$

We aim to grow W^M by promoting as few untracked predicates as possible. To this end, we extract a short prime implicant from U^M and promote the untracked variables in the support of the prime implicant (line 3). This has the effect of adding a large cube over state and untracked predicates to W^M. The PROMOTE function invoked in line 4 moves the selected untracked predicates to the set of state predicates Σ and recomputes the abstraction transition relation Δ for the new state predicates. This can lead to the introduction of new untracked and label predicates, which can serve as refinement candidates in the future.

D. Correctness

The correctness and termination theorems of [10] hold for Algorithm 1 with REFINECPRE and REFINEABSTRACTION functions defined above.

Theorem 1. *If Algorithm 1 terminates, it returns the correct answer.*

Proof. By construction, $Cpre_i^{m+}$ and $Cpre_i^{M-}$ over- and under-approximate abstract controllable predecessor operators, i.e., $Cpre_i^m(\phi)\downarrow \subseteq Cpre_i^{m+}(\phi)\downarrow$ and $Cpre_i^{M-}(\phi)\downarrow \subseteq Cpre_i^M(\phi)\downarrow$, for any set ϕ. Hence, winning sets $W^m = \textsc{Reach}(T\uparrow^m, Cpre_1^{m+})$ and $W^M = \textsc{Reach}(T\uparrow^M, Cpre_1^{M-})$ computed using these operators over- and under-approximate the winning set W of the concrete game: $W^M\downarrow \subseteq W \subseteq W^m\downarrow$.

If the algorithm returns *Yes* then the initial set of the game is a subset of the must-winning region ($I \subseteq W^M\downarrow$) and hence $I \subseteq W$. Likewise, if the algorithm returns *No* then $I \not\subseteq W^m\downarrow$ and hence $I \not\subseteq W$. In both cases the answer produced by the algorithm is correct. □

Theorem 2. *If there exists a finite region algebra $\mathcal{A}$ such that all abstractions $\langle V, \downarrow\rangle$ produced by Algorithm 1 are contained in $\mathcal{A}$ then the algorithm terminates.*

Proof outline. Let W^M and $\hat{W}^M$ be must-winning sets computed at two subsequent iterations of Algorithm 1.

We first show that refinement procedures REFINECPRE and REFINEABSTRACTION guarantee that the must-winning set computed at every iteration of the refinement loop grows monotonically, i.e., $W^M\downarrow \subseteq \hat{W}^M\downarrow$. This follows from the soundness of the refinement procedures, which improve the precision of $Cpre_i^{M-}$ at every iteration.

Next we show that the algorithm is guaranteed to make forward progress, i.e., after a finite number of refinements it either terminates or discovers new must-winning states ($W^M\downarrow \subset \hat{W}^M\downarrow$). Consider the consistency refinement procedure REFINECPRE first. Every invocation of this procedure classifies some of the untracked substates at the may/must boundary as either must-winning or must-losing (see Figure 4). Eventually, it will either classify all boundary states as must-losing, in which case $W^m\downarrow = W = W^M\downarrow$, and the algorithm terminates, or find at least one must-winning sub-state (as in Figures 4c and 4d). In the latter case, a subsequent invocation of the abstraction refinement procedure REFINEABSTRACTION is guaranteed to partition one of the boundary states so that one of the resulting abstract states is must-winning. This state will be discovered at the next run of the reachability algorithm, thus expanding the must-winning set.

Since, by the assumption of the theorem, all must-winning sets W^M generated by the algorithm belong to a finite region algebra, the algorithm is guaranteed to terminate after a finite number of iterations. □

The theory of fixed-size bit vectors supported by our current implementation satisfies the premise of Theorem 2, which guarantees the termination of the algorithm.

VI. Implementation

We implemented our abstraction refinement algorithm in the Termite [19] driver synthesis toolkit. Termite takes a model of an I/O device and a specification of the service that the driver must provide to the operating system, and synthesises a driver implementation in C. Termite provides powerful debugging facilities such as tools for analysis of synthesis failures based on counterexample strategies, interactive exploration of synthesised strategies and user-guided interactive code generation.

Our current implementation handles games with Generalised Reactivity-1 (GR(1)) [16] objectives. GR(1) games are sufficiently expressive to formalise many real-world problems, including the driver synthesis problem. They strike a balance between expressiveness and computational difficulty. We extended our abstraction refinement algorithm to handle GR(1) games as outlined by de Alfaro and Roy [10].

Termite currently supports input specifications over the concrete domain of fixed-size bit vectors and arrays. We use the Z3 SMT solver to check satisfiability and retrieve unsatisfiable cores of formulas over concrete variables (lines 11 and 16 of Algorithm 2). Quantifier elimination (line 12) over bit vector formulas is performed using our custom implementation of the decision procedure for bit vectors by Barrett et al. [2]. Termite interacts with the theory solver through a well-defined interface and hence can be readily extended with additional theories. All computations over the abstract domain are performed symbolically using the CUDD BDD package.

In addition to the techniques described in the paper we implemented a number of performance optimisations. First, we relax the requirement of Algorithm 1 that the initial abstraction must be precise for initial set I and instead overapproximate it and refine the approximation lazily whenever the algorithm discovers a spurious losing initial state. Second, we take advantage of the natural conjunctive partitioning of the transition relation and perform early quantification [4] when computing the controllable predecessor. Third, we avoid re-solving the game from scratch by reusing results of previous computations. For example, when computing the must-winning set W^M in Algorithm 1, we use the must-winning set W^M from the previous abstraction-refinement iteration as the starting value of the fixed point computation. Finally, we use BDD-specific optimisations supported by CUDD, including dynamic variable reordering [18] and variable grouping.

In addition to the predicate-based abstraction refinement algorithm, we implemented the original algorithm by de Alfaro and Roy, based on variable abstraction, which enables direct comparison of the two techniques.

Termite consists of 30,000 lines of Haskell code, with the core abstraction refinement algorithm accounting for 1,800 lines, and took approximately 10 person-years to develop.

VII. Evaluation

We evaluate Termite by synthesising drivers for several real-world I/O devices, including an IDE hard disk, a real-time clock, two versions of UART serial controller, two versions of I2C bus controller, an SPI bus controller, and a UVC webcam. We developed corresponding device and OS models using the Termite Specification Language (TSL) by following the common methodology used by hardware developers in building high-level device models. We refer the reader to

	Statistic	Case study									
		IDE	RTC	UART-1	UART-2	I2C-1	I2C-2	SPI	UVC	simple SPI	simple I2C
1	concrete state vars (bits)	83 (952)	64 (624)	61 (335)	65(896)	64 (458)	50(222)	66(644)	95 (75908)	7 (46)	11 (64)
2	concrete label vars (bits)	27 (389)	24 (199)	20 (86)	15(289)	25 (199)	15(81)	24(384)	33 (49657)	9 (58)	14 (42)
3	consistency refinements	11	9	42	4	12	4	6	22	0	23
4	state refinements	18	16	18	50	15	17	26	25	11	9
5	state predicates	31	25	33	58	24	24	31	30	14	17
6	label predicates	57	41	40	53	36	32	28	130	19	36
7	untracked predicates	7	4	35	2	5	1	6	32	0	0
8	run time (s)	71	74	309	603	39	43	14	190	1	10
9	peak BDD size	864612	515088	907536	1142596	440482	688828	324996	785918	87892	242214
	Performance of the de Alfaro and Roy algorithm [10]										
10	run time (s)	∞	∞	∞	∞	∞	∞	∞	∞	865	1151
11	peak BDD size	-	-	-	-	-	-	-	-	400624	4088000

TABLE I: Summary of experimental case studies.

an accompanying paper for a more detailed description of the TSL language and the modelling methodology [19]. The source code of the case studies is available as part of the Termite distribution [21].

Table I summarises our experiments. The first two rows characterise the complexity of the input models in terms of the number of variables and the total number of bits used in the concrete specification of the game. Concrete state variables model internal device state, as well as the state of the driver-OS interface; label variables model commands and responses exchanged by the driver, the device, and the OS.

Rows 3 and 4 show the number of iterations of the abstraction refinement loop required to solve the game. Rows 5 through 7 show the size of the abstract game at the final iteration, when a winning strategy for the driver was obtained, in terms of the number of state, label, and untracked predicates. These results demonstrate the dramatic reduction of the problem dimension achieved by our abstraction refinement method. The resulting abstract games are still too complex to solve using explicit state enumeration, hence the use of symbolic techniques is essential. In all case studies, Termite was able to find the winning strategy within 11 minutes running on a 2.9GHz Intel Core i7 laptop (row 8), with peak BDD size under one million nodes (row 9).

The two final rows show the performance of the original three-valued abstraction refinement algorithm of de Alfaro and Roy on our benchmarks. As expected, the algorithm does not terminate on any of the real-world driver benchmarks within a two-hour time limit. We therefore developed simplified versions of two of the benchmarks (SPI and I2C-2) with significantly reduced state spaces. As shown in the last two columns of the table, the de Alfaro and Roy algorithm terminates on these benchmarks; however it takes several orders of magnitude longer than our new algorithm, which uses predicate abstraction. These results show that predicate abstraction is essential to solving complex real-world games.

VIII. Conclusion

We presented and evaluated a practical predicate-based abstraction refinement algorithm for solving games. To the best of our knowledge, this is the first such algorithm described in the literature. We addressed key performance bottlenecks involved in applying predicate abstraction in game settings and demonstrated that our algorithm performs well on real-world reactive synthesis benchmarks.

References

[1] T. Ball, B. Cook, S. Das, and S. K. Rajamani. Refining approximations in software predicate abstraction. In *TACAS*, pages 388–403, Barcelona, Spain, Mar. 2004.

[2] C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for bit-vector arithmetic. In *DAC*, pages 522–527, San Francisco, California, USA, June 1998.

[3] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware from PSL. *ENTCS*, 190(4):3–16, Nov. 2007.

[4] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. pages 49–58. North-Holland, 1991.

[5] F. Cassez, J. J. Jessen, K. G. Larsen, J.-F. Raskin, and P.-A. Reynier. Automatic synthesis of robust and optimal controllers - an industrial case study. In *HSCC*, pages 90–104, San Francisco, CA, USA, Apr. 2009.

[6] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, Chicago, IL, USA, July 2000.

[7] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.

[8] S. Das and D. Dill. Successive approximation of abstract transition relations. In *LICS*, pages 51–58, Boston, MA, USA, June 2001.

[9] L. de Alfaro, P. Godefroid, and R. Jagadeesan. Three-valued abstractions of games: uncertainty, but with precision. In *LICS*, pages 170–179, Turku, Finland, July 2004.

[10] L. de Alfaro and P. Roy. Solving games via three-valued abstraction refinement. In *CONCUR*, pages 74–89, Lisboa, Portugal, Sept. 2007.

[11] R. Dimitrova and B. Finkbeiner. Abstraction refinement for games with incomplete information. In *FSTTCS*, Bangalore, India, Dec. 2008.

[12] R. Dimitrova and B. Finkbeiner. Counterexample-guided synthesis of observation predicates. In *FORMATS*, pages 107–122, London, UK, Sept. 2012.

[13] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, Haifa, Israel, June 1997.

[14] T. A. Henzinger, R. Jhala, and R. Majumdar. Counterexample-guided control. In *ICALP*, pages 886–902, Eindhoven, The Netherlands, July 2003.

[15] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, Portland, Oregon, Jan. 2002.

[16] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of Reactive(1) designs. pages 364–380, Charleston, SC, USA, Jan. 2006.

[17] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190, Austin, Texas, USA, Jan. 1989.

[18] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD*, pages 42–47, Santa Clara, CA, USA, 1993.

[19] L. Ryzhyk, A. Walker, J. Keys, A. Legg, A. Raghunath, M. Stumm, and M. Vij. User-guided device driver synthesis. In *OSDI*, Broomfield, CO, USA, Oct. 2014.

[20] S. Shoham and O. Grumberg. A game-based framework for CTL counterexamples and 3-valued abstraction-refinement. In *CAV*, pages 275–287, Boulder, Colorado, USA, July 2003.

[21] Termite 2 driver synthesis tool. http://www.termite2.org.

APPENDIX

Algorithm 2 has an important performance issue: in lines 10–16 it analyses and adds to C^{M-} a single abstract label l. The set of all abstract labels is exponential in size in the number of label predicates and can be very large in practice, making explicit enumeration infeasible. We therefore modify the algorithm to handle a set of abstract labels at every iteration. To this end, in line 7, instead of choosing a complete assignment to state, untracked, and label predicates, we compute a *prime implicant* of set T, i.e., an assignment to a subset of variables in $\vec{\sigma} \cup \vec{\omega} \cup \vec{\lambda}$ such that any extension of this assignment to the remaining variables satisfies T:

$$\langle v, u, l \rangle \leftarrow \text{PRIMEIMPLICANT}(T),$$

where v, u, and l are partial valuations of abstract variables, which compactly represent a potentially large set of abstract transitions. In practice, we typically discover prime implicants that only constrain few of the abstract variables, meaning that other predicates are irrelevant for the outcome of the transition.

Given this modification, the C^{m+} refinement case of the algorithm (lines 18–19) is still correct and does not require any changes. However, changes are needed in the C^{M-} refinement logic. The set A computed in lines 10–1 contains all concrete states where *at least one* abstract label from the set characterised by the partial assignment l of label predicates is available. It does not guarantee the availability of any particular label from this set. To model this constraint, we would have to change C^{M-} to be a relation over sets of labels. Every element of the relation would describe a set of labels, one of which is guaranteed to be available for the given state and untracked predicate assignment.

This introduces a new form of imprecision to the consistency relation: rather than categorising each abstract label as available or unavailable in the given state, we record a set of labels, one of which is available. However, such a relation would be hard to represent and manipulate efficiently in the symbolic form. Therefore, we propose a different approach that allows symbolic implementation.

We transform the game (without changing its winning set) in order to solve it more efficiently. The idea of our solution is to model the new form of imprecision as non-determinism in the abstract transition relation Δ. For each abstract label variable λ_i, we introduce an auxiliary *enabling variable* ε_i and transform the transition relation Δ as follows:

$$\Delta \leftarrow (\varepsilon_i \wedge \Delta) \vee (\overline{\varepsilon_i} \wedge \exists \lambda_i . \Delta).$$

When $\varepsilon_i = true$, Δ behaves exactly as before. In case $\varepsilon_i = false$, the value of λ_i chosen by the player is ignored and the environment non-deterministically selects next-state variables assignment that is consistent with *some* assignment of λ_i.

We can now modify line 16 of Algorithm 2 as follows:

$$C^{M-} \leftarrow C^{M-} \vee (\hat{A} \wedge \bigwedge_{i \in j_1 \ldots j_p} \lambda_i = l_i \wedge \bigwedge_{i \notin j_1 \ldots j_p} \varepsilon_i = false),$$

where $j_1 \ldots j_p$ are indices of variables assigned by l. The above statement refines C^{M-} by allowing the player to assign a subset of label variables in accorance with l and disabling other label variables not constrained by l. The environment will non-deterministically pick arbitrary values for these variables. In this way we precisely capture what we currently know about consistent predicate assignments in a a symbolic form, at the cost of introducing extra variables ε_i.