On Improving the Efficiency of Game Solving for Hybrid System Control

Rüdiger Ehlers[1]

Abstract— **In order to control a system with complex physical dynamics against a temporal specification, we can compute a discrete abstraction of the system dynamics and then solve a game built from the abstraction and the specification. A strategy in this game is then a controller that enforces the satisfaction of the specification. Such games are typically huge, which implies the need to solve them symbolically, i.e., without considering every position in the game separately. Binary decision diagrams (BDDs) are the most commonly used data structure for this purpose, but the BDD of a system's transition relation is typically huge, which limits the scalability of the approach.**

In this paper, we present a new approach to implement the enforceable predecessor operator for games represented in BDD form. Solving a game is typically performed by repetitive application of this operator. Our new approach targets system dynamics for which some dimensions are translation invariant, as common in robotics and vehicle control. We avoid the construction of the overall transition relation in the game and instead base the computation on local substitutions of coordinate values in the translation invariant dimensions, which keeps the intermediate result BDDs small.

We perform a comprehensive experimental comparison of the classical enforceable predecessor implementation and our new operator implementation. The results show that our approach reduces game solving times and hence increases the scalability of controller synthesis when employing a physical system abstraction.

I. INTRODUCTION

Controlling continuous, hybrid, and cyber-physical systems to satisfy complex temporal specifications requires to take all possible environment behaviors into account. Due to the complexity of this problem, automatic controller synthesis has emerged as a field of research to support the engineering process of controllers for such systems [1], [2], [3]. Controller synthesis for non-deterministic environments is typically reduced to solving a game between two players that captures the interaction between the system to be controlled and the environment, including the dynamics of the system and the specification [4]. A winning strategy for the *system player* in the game can then be compiled to a controller that enforces the satisfaction of the specification. Due to the undecidability of solving games of most of the types that we obtain when using a precise description of the continuous/hybrid system dynamics [5], game-based controller synthesis is commonly performed using a *discrete abstraction* of the system dynamics [6]. The resulting games then have a finite number of states and can be solved with standard game solving algorithms. The abstraction *alternatingly simulates* the original system [7], [8], which ensures that whenever we obtain a controller that works correctly on the discrete abstraction, we can translate it to one that also works correctly for the continuous system dynamics from which we took an abstraction.

Using a discrete abstraction of the continuous system dynamics comes at a price, however. For all but the most simple systems, abstractions that are fine enough to permit a winning strategy for the system player in the games built from them have huge state spaces. This makes an explicit representation of their state spaces infeasible. Equally badly, solving games with such huge state spaces is computationally expensive. An alternative are *symbolic* representations of abstractions and the games built from them [4], [9]. In a symbolic representation, state sets are manipulated in an implicit representation. Binary decision diagrams (BDDs) [10] are the most commonly used representation in this context. Unfortunately, the strong dependencies between the dimensions that are observable in many abstractions reduce the efficiency of game solving based on their symbolic representation. However, symbolic data structures such as BDDs have been shown to be instrumental to support complex temporal specification in a controller synthesis process. As the controller synthesis games are products of the specification automata and the abstraction [8], both parts have to be efficiently representable. Hence, rather than abandoning symbolic state space representations for continuous/hybrid system controller synthesis, improving the efficiency of using them appears to be a more reasonable approach.

In this paper, we present a new way to implement the *enforceable predecessor* operator [11] for game solving using binary decision diagrams (BDDs). In contrast to the traditional way of implementing the operator, the new operator implementation does not require building a monolithic BDD for the overall transition relation in the abstraction. Rather, it makes use of the fact that many system dynamics have translation invariant dimensions. For example, the relative position changes of a vehicle are independent of its location, which can be exploited in the game solving process. The new operator implementation is based on substituting state values in such translation invariant dimensions in target state sets by the respective values before a step in the abstraction, which is more efficient than the existential and universal abstraction operations performed in the traditional operator implementation. This keeps the BDDs computed during game solving small, which is the core reason for a higher solving efficiency with our approach.

The new operator implementation supports dynamics in

[1] Rüdiger Ehlers is with the University of Bremen and DFKI GmbH, Bremen, Germany

This work was supported by the Institutional Strategy of the University of Bremen, funded by the German Excellence Initiative.

which only some dimensions are translation invariant. We give a comprehensive experimental evaluation in which we compare it with the classical operator implementation. To make such a comparison meaningful, we performed it in a setting that has as few external influence factors as possible. We used only simple specifications (so that the state spaces of the games considered can be exactly the same as the state spaces of the abstractions). In previous works, the chosen variable order or the used dynamic reordering heuristic has been shown to have a huge influence on the computation times [12], and this influence is hard to predict. To reduce the noise by this effect, we compare the operators on a large number of benchmarks and consider both static and dynamic variable orderings in the evaluation.

II. Preliminaries

a) System Abstractions: Continuous system control can be reduced to controlling a discrete-time abstraction when the concrete systen and its abstraction are in a suitable formal relation. Tabuada [7] described alternating simulation relations as a suitable concept, which was later refined to *feedback refinement relations* by Reissig, Weber, and Rungger [6]. In both cases, a correct controller for the abstraction can be compiled to a correct controller for the concrete system. The details of this process are left unaltered by the approach presented in this paper, and we refer the interested reader to [7], [6] for further information.

A *discrete-time discrete-state system abstraction* is defined as a tuple $\mathcal{A} = (S, A, T, \Sigma, L)$, where S is a finite set of states, A is a finite set of actions, $T \subseteq S \times A \times S$ is a transition relation, Σ is a state label alphabet, and $L : S \to \Sigma$ is a state labeling function. A *decision sequence* of $\mathcal{A}$ is defined as a word $\rho = \rho_0\rho_1\rho_2 \ldots \in A^\omega$, where A^ω represents the set of infinite words over characters from A. We say that an infinite word $\pi = \pi_0\pi_1 \ldots \in S^\omega$ is a *trace* induced by ρ if for every $i \in \mathbb{N}$, we have that $(\pi_i, \rho_i, \pi_{i+1}) \in T$. We say that an *output sequence* $w = w_0w_1 \ldots \in \Sigma^\omega$ is induced by π if for every $i \in \mathbb{N}$, we have $w_i = L(\pi_i)$.

b) Specifications: A specification over some alphabet Σ is a set $\mathcal{L} \subseteq \Sigma^\omega$ that states the allowed output sequences in an abstraction that a controller to be computed should enforce. Enforcing a specification means to select an action sequence such that every output sequence induced by some possible trace in the abstraction for the action sequence satisfies the specification. For every $i \in \mathbb{N}$, the ith element of the action sequence can depend on the first i trace elements.

c) Games: Checking if a specification can be enforced is commonly reduced to solving two-player games. A *game* is a tuple $\mathcal{G} = (V, \Sigma^0, \Sigma^1, E^0, E^1, \delta, \mathcal{L})$ with the set of *vertices* (or *positions*) V, the *action set* Σ^0/Σ^1 of player 0 and 1, respectively, the allowed *action functions* $E^0 : V \to 2^{\Sigma^0}$ and $E^1 : V \times \Sigma^0 \to 2^{\Sigma^1}$, the (partial) *transition function* $\delta : V \times \Sigma^0 \times \Sigma^1 \to V$, and the *winning condition* $\mathcal{L} \subseteq (\Sigma^0 \times \Sigma^1 \times V)^\omega$ of player 0. For all $v \in V$, we require $\delta(v, x^0, x^1)$ to be defined if and only if $x^0 \in E^0(v)$ and $x^1 \in E^1(v, x^0)$.

Two *decision sequences* $\rho^0 = \rho^0_0\rho^0_1 \ldots \in (\Sigma^0)^\omega$ and $\rho^1 = \rho^1_0\rho^1_1 \ldots \in (\Sigma^1)^\omega$ for the two players induce a *play* in $\mathcal{G}$, which can either be finite or infinite. A play $\pi = \pi_0 \ldots \pi_n$ is finite if we have $\pi_{i+1} = E(\pi_i, \rho^0_i, \rho^1_i)$ for all $0 \leq i < n$, and either $\rho^0_n \notin E^0(\pi_i)$ or $\rho^1_n \notin E^1(\pi_i, \rho^0_n)$. A play $\pi = \pi_0 \ldots \pi_n$ is infinite if we have $\pi_{i+1} = \delta(\pi_i, \rho^0_i, \rho^1_i)$ for all $i \in \mathbb{N}$. A finite play is *winning* for player 0 if we have $\rho^1_n \notin E^1(\pi_i, \rho^0_n)$, otherwise it is winning for player 1. An infinite play is winning for player 0 if $(\rho^0_0, \rho^1_0, \pi_0)(\rho^0_1, \rho^1_1, \pi_1) \ldots \in \mathcal{L}$. A *strategy* for player 0 is a function $f : V^* \to \Sigma^0$. The strategy is winning from a vertex $v \in V$ if for all decision sequences ρ^0 and ρ^1 and all plays π induced by ρ^0 and ρ^1, if we have $\rho^0_i = f(\pi_0 \ldots \pi_{i-1})$ for all $i \in \mathbb{N}$ and $\pi_0 = v$, then the play is winning for player 0.

Given an abstraction $\mathcal{A} = (S, A, T, \Sigma, L)$ and a specification $\mathcal{L}$, a game $\mathcal{G} = (V, \Sigma^0, \Sigma^1, E^0, E^1, \delta, \mathcal{L}')$ to check if $\mathcal{L}$ can be enforced in $\mathcal{A}$ can be defined with the following components:

- $V = S$
- $\Sigma^0 = A$
- $\Sigma^1 = V$
- For all $v \in V$, $E^0(v) = \{a \in A \mid \exists v' \in V.(v, a, v') \in T\}$
- For all $v \in V$, $a \in A$, $E^1(v, a) = \{v' \in \Sigma^1 \mid (v, a, v') \in T\}$
- For all $v \in V$, $a \in E^0(v)$, $v' \in E^1(v, a)$, $\delta(v, a, v') = v'$
- $\mathcal{L}' = \{(x^0_0, x^1_0, v_0)(x^0_1, x^1_1, v_1) \ldots \in (\Sigma^0 \times \Sigma^1 \times V)^\omega \mid L(v_0)L(v_1) \ldots \in \mathcal{L}\}$

In such a game, player 0 is the *system player* whereas player 1 is the *environment player*. In every position v of the game, the system player chooses an action $a \in A$ to be executed next, and the environment player then chooses a next position v' such that (v, a, v') is a possible transition in the abstraction from which the game was built. The game implements the idea that the system can choose an action, but it does not know in advance which transition in the abstraction is taken. A winning strategy can take the history of a play into account, which models the fact that a controller can take the evolution of the system's state into account as well.

In this paper, we only build games from abstractions as defined above. However, in more complex applications, the action sets of the two players can also be richer and model additional input and output of the system, such as user interaction and sensor input. More details on how this works can be found in [8, Section 5.1] Restricting our attention to games build from an abstraction allows us to reduce the amount of notation, in particular as the δ function in a game becomes trivial.

d) Game Solving: Solving a game means to partition the set of positions in a game into those from which player 0 has a winning strategy and those from which she does not have a winning strategy. How to solve a game depends on the type of winning condition $\mathcal{L}$. The most simple game type is a *safety game*, where for some set $V_{bad} \subseteq V$ of *bad* *positions*, $\mathcal{L}$ consists of all words $(\rho_0^0, \rho_0^1, \pi_0)(\rho_1^0, \rho_1^1, \pi_1)\ldots$ for which for no $i \in \mathbb{N}$, we have $\pi_i \in V_{bad}$.

In a safety game, the system player tries to avoid reaching a bad vertex during a play. We can find out from which positions the system player can avoid to reach a bad vertex in one step by computing the *enforcable predecessor* for $V' = V \setminus V_{bad}$:

$$\begin{aligned}\mathrm{EnfPre}(V') = \{v \in V \mid \exists x \in E^0(v).\\ \forall y \in E^1(x,y).\delta(v,x,y) \in V'\}\end{aligned}$$

The operator computes the set of positions from which player 0 has a valid action x for which, no matter what action player 1 chooses, the next transition will lead to a position in V'. So for $V' = V \setminus V_{bad}$, we get the set of positions from which player 0 can ensure not to reach a bad position in one step. By continuing this line of reasoning, the following sequence of position sets defines the set of positions from which the system player can avoid to reach a bad position in 1 step in a play, in 1 to 2 steps in a play, in 1 to 3 steps in a play, and so on:

$$W_1 = \mathrm{EnfPre}(V \setminus V_{bad}) \quad (1)$$
$$W_2 = \mathrm{EnfPre}(W_1) \cap W_1 \quad (2)$$
$$W_3 = \mathrm{EnfPre}(W_2) \cap W_2 \quad (3)$$
$$\ldots\ldots \quad (4)$$

By the fact that by definition $W_{i+1} \subseteq W_i$ for every $i \in \mathbb{N}$ and the fact that V is finite, this sequence converges to a position set W_{final} after a finite number of steps. It can be shown that this is the set of positions from which the system player has a strategy to ensure that V_{bad} is never reached, and hence the set of positions from which it has a winning strategy. This winning strategy always picks actions that ensure that a play stays in W_{final}.

For more complicated winning conditions such as a conjunction of a safety winning condition and a *liveness* winning condition, for which some positions must be reached infinitely often in a play, the set of winning positions can be computed in a similar process, i.e., by repetitive applications of the EnfPre operator to positions sets (and merging the results with simple set union, complementation, and intersection operations) [13]. This is also the approach used in the experimental evaluation in Section IV.

e) Binary Decision Diagrams: Symbolic data structures such as *binary decision diagrams* (BDDs) are often employed for solving games with a large number of positions. From a semantic point of view, they represent Boolean formulas $f : 2^{\mathcal{B}} \to \mathbb{B}$ for some set of variables $\mathcal{B}$. For notational simplicity, we use subsets of $\mathcal{B}$ and their characteristic functions interchangeably, i.e., a set $B \subseteq \mathcal{B}$ also represents a function that maps all variables in B to $\mathbf{true}$ and the others to $\mathbf{false}$.

The usual operations on Boolean functions are defined on Binary decision diagrams as well. Representatives for them are disjunction, conjunction, existential and universal abstraction, and complementation. For instance, given two BDDs f and f', the function $f \vee f'$ maps all valuations to $\mathcal{B}$ to $\mathbf{true}$ that either f or f' map to $\mathbf{true}$. As an example for a more complex operation, the existential abstraction $\exists B.f$ for a subset of variables $B \subseteq \mathcal{B}$ is a Boolean function with $(\exists B.f)(B') = \mathbf{true}$ for some $B' \subseteq \mathcal{B}$ if and only if there exists some $B'' \subseteq V$ with $f((B' \setminus B) \cup B'') = \mathbf{true}$.

f) BDDs for Games: A BDD can represent a set of positions in a game and its E^0 and E^1 functions. For this, the set of variables $\mathcal{B}$ needs to contain at least some variables $\{b_0^V, \ldots, b_{n-1}^V, b_0^A, \ldots, b_{m-1}^A, b_0^{A'}, \ldots, b_{m-1}^{A'}\}$ such that $b_0^V, \ldots, b_{n-1}^V$ are plenty enough to encode all positions in V, i.e., we have $n \geq \lceil \log_2(|V|) \rceil$, and $b_0^{A'}, \ldots, b_{n-1}^{A'}$ are plenty enough to encode all actions in A, i.e., we have $m \geq \lceil \log_2(|A|) \rceil$. We assume that some (arbitrary) encoding functions $[\![\cdot]\!]_V : V \to 2^{\{b_0^V, \ldots, b_{n-1}^V\}}$ and $[\![\cdot]\!]_A : V \to 2^{\{b_0^A, \ldots, b_{m-1}^A\}}$, and an encoding function $[\![\cdot]\!]_{V'} : V \to 2^{\{b_0^{V'}, \ldots, b_{n-1}^{V'}\}}$ are given. The latter function uses the same bitwise encoding as $[\![\cdot]\!]_V$ (i.e., for which we have $[\![v]\!]_{V'} = [\![v]\!]_{V'}[b_0^{V'}, b_1^{V'}, \ldots, b_{n-1}^{V'}/b_0^V b_1^V \ldots b_{n-1}^V]$, where $K[a/b]$ represents replacing every occurrence of some element a in some set K by some term b. Whenever there are multiple elements given in this operation, they are replaced simultaneously and while respecting the orders of the variables in the element lists. With these functions, we can encode a set of positions $V' \subseteq V$ by $f_{V'} = \bigvee_{v \in V'} [\![v]\!]$, the allowed action function E^0 as $f_{E^0} = \bigvee_{v \in V, a \in E^0(v)} ([\![v]\!]_V \wedge [\![a]\!]_A)$ and the allowed action function E^1 as $f_{E^1} = \bigvee_{v \in V, a \in E^0(v), v' \in E^1(v,a)} ([\![v]\!]_V \wedge [\![a]\!]_A \wedge [\![v']\!]_{V'})$.

With E^0 and E^1 encoded into BDDs, we can solve games with them. If f is a BDD that represents some set of positions, we can compute the enforceable predecessor operator as follows:

$$\begin{aligned}\mathrm{EnfPre}(f) = \exists b_0^A, \ldots, b_{m-1}^A.(\forall b_0^{V'}, \ldots, b_{n-1}^{V'}.\\ (f[b_0^{V'}, \ldots, b_{n-1}^{V'}/b_0^V, \ldots, b_{n-1}^V] \vee \neg f_{E^1}) \wedge f_{E^0})\end{aligned}$$

With this operator defined over BDDs, the steps for computing the winning positions given in Equation 1 to 4 can be rewritten as follows:

$$\begin{aligned}f_{W_1} &= \mathrm{EnfPre}(\neg V_{bad})\\ f_{W_2} &= \mathrm{EnfPre}(f_{W_1}) \wedge f_{W_1}\\ f_{W_3} &= \mathrm{EnfPre}(f_{W_2}) \wedge f_{W_2}\\ &\ldots\ldots\end{aligned}$$

Since BDDs represent characteristic functions of sets, we replaced the $\cap$ operator by a $\wedge$.

g) BDD Internals: BDDs are internally represented as directed acyclic graphs in which paths from a designated root node to a $\mathbf{true}$ sink node represent variable assignments that are mapped to $\mathbf{true}$ by the BDD. Such details are generally not important for the scope of this paper, but two of their effects will be of relevance:

(1) There is a global order among the variables in a BDD. The number of nodes in a BDD depends on this variable order for many applications, so choosing a good variable order for an application is important, yet difficult. As a

consequence, modern BDD packages often support *dynamic variable reordering*, in which some heuristics are employed that automatically reorder the variables from time to time. Since all operations performed on BDDs assume that all BDDs have the same variable order, BDD libraries enforce a global order for all BDDs. Due to the importance of a good order, dynamic reordering is often crucial for good performance, but at the same time the effect of the heuristics are hard to predict, especially since the computed order depends on the BDDs that a program maintains at the point of time at which reordering is triggered.

(2) Binary decision diagram libraries also employ *caching* of intermediate results. The sizes for caches are however limited, and for different variable orders, cache hit rates can vary substantially. Thus, the computation time of a BDD-based algorithm can vary a lot with minor differences in the input, which asks for using a large number of benchmarks when comparing BDD-based algorithms.

III. Exploiting Translation Invariant Dimensions in Abstractions

The problem addressed in this paper is the improvement of the efficiency of BDD-based game solving for games that are (at least in part) built from the abstraction of a physical system. The continuous state spaces from which the abstractions are built are subsets of a convex set Workspace $\subset \mathbb{R}^d$ for some number of dimensions $d \in \mathbb{N}$ and we assume that we have a way to compute a system abstraction that faithfully models the physical system, i.e., such that a controller for the abstraction can be translated to a correct controller for the physical system. This assumption is reasonable, as there are tools such as `Pessoa` [9] and `SCOTS` [4] available for computing such abstractions.

We want to improve game solving performance by exploiting two properties of such abstractions:

1) For d-dimensional physical systems, the state space of the abstraction is $S = \{lb_1, \ldots, ub_1\} \times \{lb_2, \ldots, ub_2\} \times \ldots \times \{lb_d, \ldots, ub_d\}$ for the lower bound sequence $lb_1, \ldots, lb_d$ and upper bound sequence $ub_1, \ldots, ub_d$.
2) Whenever the behavior of the system along the kth dimension is translation invariant (for some $k \in \mathbb{N}$), i.e., the relative change of the system state does not depend on the concrete state value in the kth dimension, then this is the case for the system abstraction as well.

Translation invariance is often found for the x and y coordinate dimensions in vehicle abstractions, as the dynamics of a vehicle does not depend on where exactly in a workspace it is located. The following formal definition of translation invariance captures the idea at the abstraction level, as we will only deal with the concept on this level in the following.

Definition 1: Let $\mathcal{A} = (S, A, T, \Sigma, L)$ be an abstraction with $S = \{lb_1, \ldots, ub_1\} \times \{lb_2, \ldots, ub_2\} \times \ldots \times \{lb_d, \ldots, ub_d\}$ for some $d \in \mathbb{N}$. We say that dimension $k \in \{1, \ldots, d\}$ is *translation invariant* if for every $(x_1, \ldots, x_d) \in S$, $a \in \Sigma$, and $(x'_1, \ldots, x'_d) \in S$ with $((x_1, \ldots, x_d), a, (x'_1, \ldots, x'_d)) \in T$ and for every $c \in \mathbb{Z}$, either

1) $(x_1, \ldots, x_{k-1}, x_k + c, x_{k+1}, \ldots, x_d), a, (x'_1, \ldots, x_{k-1}, x'_k + c, x'_{k+1}, \ldots, x'_d)) \in T$ (i.e, the same transition shifted by $c \in \mathbb{Z}$ in the kth dimension is contained in T as well),
2) $(x_1, \ldots, x_{k-1}, x_k + c, x_{k+1}, \ldots, x_d) \notin S$ (the source state shifted by $c \in \mathbb{Z}$ in the kth dimension is not part of the abstraction), or
3) for no $s' \in S$, we have $((x_1, \ldots, x_{k-1}, x_k + c, x_{k+1}, \ldots, x_d), a, s') \in T$ (action a is not available from the source state shifted by $c \in \mathbb{Z}$ in the kth dimension).

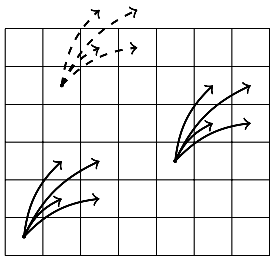

Fig. 1. Explanation figure for translation invariance.

Figure 1 depicts the concept. We assume three-dimensional system dynamics, where only two dimensions are depicted and the predecessor state value for the third dimension and the chosen action is the same for all transitions depicted in the figure. If the dimensions represented by the vertical axis and the horizonal axis in the figure are translation invariant, then this implies that the same transitions shifted up, down, left and right are also present in the abstraction whenever a state/action pair has any successor states in the transition relation. The latter is for example not the case whenever one transition for a state/action pair has a successor state that would be out of the boundaries of the workspace, and hence the action cannot be executed by a controller in the state without risking to leave the designated workspace.

A. Game Representation Compression using Translation Invariance

Translation invariant dimensions $k \in \{1, \ldots, d\}$ enable us to compress the representation of T: rather than storing the complete set T, we can store one representative for every transition $(x_1, \ldots, x_{k-1}, x_k + c, x_{k+1}, \ldots, x_d), a, (x'_1, \ldots, x_{k-1}, x'_k + c, x'_{k+1}, \ldots, x'_d)) \in T$ for some $c \in \mathbb{Z}$. As a consequence, we can reduce the size of the representation of T by a factor of up to $ub_k - lb_k + 1$. If multiple dimensions are translation invariant, we can save even more space. When building games $\mathcal{G} = (V, \Sigma^0, \Sigma^1, E^0, E^1, \delta, \mathcal{L}')$ from abstractions, the compressed representation of T gives rise to a compressed representation for E^1, as this is where T gets encoded into.

Such a compressed representation of E^1 in a game enables us to encode E^1 by a BDD over a smaller set of variables: rather than allocating Boolean variables to encode both the x_k and x'_k components of a transition

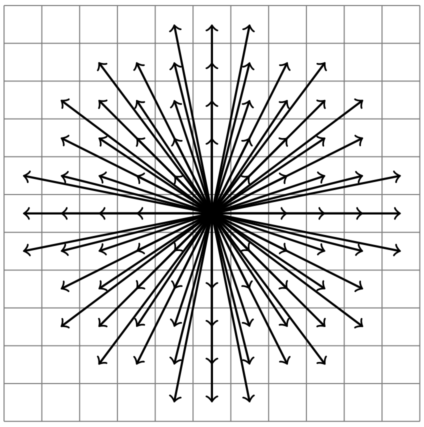

Fig. 2. Possible relative position updates of the single-speed vehicle dynamics used in Section IV.

$((x_1, \ldots, x_d), a, (x'_1, \ldots, x'_d)) \in T$, we only need to encode the relative update, i.e., $x'_k - x_k$.[1] Hence, we can save $\lceil \log_2(ub_k - lb_k + 1) \rceil$ Boolean variables, which reduces the workload of the BDD operations.

B. *An* EnfPre *Operator for Compressed Games*

A compressed representation of components of a game is only useful if we have a way to apply the enforceable predecessor operator to game position sets. As we do not compress the game position sets themselves, but only the representation of E^1, which is only used in this operator during game solving, defining a modified enforcable predecessor operator implementation is sufficient for working with compressed game representations.

The BDD-based EnfPre operator that we present in this paper is based on the observation that in abstractions used for controlling physical systems, the state value changes in the translation invariant dimensions are typically rather small. For instance, Figure 2 shows the possible position updates for a single-speed vehicle dynamics that we use in the experimental evaluation in the next section. The dynamics have three dimensions – apart from the translation invariant x and y coordinate dimensions, the vehicle also has a current heading (which is not shown in the figure). The abstraction was obtained using a fixed time-step. In every such time step, the x or y positions of the vehicle can change by a value of at most 5. For large workspaces, this means that only states with few different (x, y) position changes are reachable by making a single step in the game.

This fact can be exploited in a BDD-based implementation of the EnfPre operator, provided that f_{E^1} is encoded appropriately. Let $\mathcal{D}_{inv} \subseteq \{1, \ldots, d\}$ be the set of translation invariant dimensions, and $\mathcal{B}$ be a set consisting of the following Boolean variables:

[1]This line of reasoning assumes that all state components are encoded into separate sets of Boolean variables when translating E^1 into BDD form. The assumption is justified as both the `Pessoa` and `SCOTS` tools produce BDDs with such encodings, and no systematic approaches to exploit the possibility of not doing so appears to be known in the literature.

- $b_0^{V,i}, \ldots, b_{n_i}^{V,i}$ for every $i \in \{1, \ldots, d\}$ are the variables to encode the state in dimension i, where $n_i = \lceil \log_2(ub_i - lb_i) \rceil$ is the number of bits for the size of the workspace in the ith dimension,
- $b_0^A, \ldots, b_m^A$ are the variables for encoding the actions, where $m = \lceil \log_2(|A|) \rceil$, and
- $b_0^{V',i}, \ldots, b_{n_i}^{V',i}$ for every $i \in \{1, \ldots, d\} \setminus \mathcal{D}_{inv}$ are the variables to encode successor states.

This variable set is the same as in the previous section, except that we split up the variables for every dimension and we do not have two copies of the variables for the translation invariant state dimensions. We denote the encoding of a value in dimension i in the state into the corresponding Boolean values as $[\![\cdot]\!]_{V,i}$ and $[\![\cdot]\!]_{V',i}$, respectively, and let $\mathcal{B}_V$, $\mathcal{B}_A$, and $\mathcal{B}_{V'}$ denote the sets of variables of the three types listed above. We can now encode E^1 as follows:

$$f_{E^1} = \bigvee_{((x_1,\ldots,x_d),a,(x'_1,\ldots,x'_d)) \in T} \Bigg(\Bigg(\bigwedge_{k \in \{1,\ldots,d\} \setminus \mathcal{D}_{inv}} [\![x_k]\!]_{V,k} \wedge [\![x'_k]\!]_{V',k} \wedge [\![a]\!]_A \Bigg) \wedge \bigwedge_{k \in \mathcal{D}_{inv}} [\![x'_k - x_k]\!]_{V,k} \Bigg)$$

Thus, for every non-translation invariant dimension, predecessor and successor state values are stored in the traditional way and as explained in Sect. II. For the translation invariant dimension, we do not store the predecessor and successor values in f_{E^1}, but rather encode the relative differences. Note that for some dimension $k \in \mathcal{D}_{inv}$, the value of $x'_k - x_k$ may not be within $\{lb_k, \ldots, ub_k\}$ – for this reason, we store the value using a two's complement binary encoding and ignore lb_k and ub_k for such dimensions.

Using such an encoding for f_{E^1}, we can now give an alternative EnfPre operator definition over BDDs. For this, we first define the set of possible state component updates in the translation invariant dimensions (which is depicted in Figure 2 for an example dynamics):

$$\mathsf{Disloc} = \{ \bigcirc_{i \in \{1,\ldots,d\}} \left(\begin{cases} x'_i - x_i & \text{if } i \in \mathcal{D}_{inv} \\ 0 & \text{else} \end{cases} \right) \Bigg| \; ((x_1, \ldots, x_d), a, (x'_1, \ldots, x'_d)) \in T \}$$

In this equation, the $\bigcirc$ operator represents building a tuple over the expression right of it for all elements of the index set. For notational simplicity in the following, the elements of Disloc are d-tuples, where values for the dimensions that are not rotation invariant are set to 0. Using this set, we can then define a set of substitution sequences

$$\mathsf{Subst} = \{ \mathsf{Concat}_{i \in \mathcal{D}_{inv}} \mathsf{Subst}_i(x_i) \mid (x_1, \ldots, x_n) \in \mathsf{Disloc} \}$$

for

$$\mathsf{Subst}_i(x_i) = \mathsf{Concat}_{j \in \{0,\ldots,n_i\}} \left(b_j^{V,i}, \bigvee_{\substack{k \in \{\max(lb_i, lb_i + k), \ldots, \\ \min(ub_i, ub_i + k)\}}} [\![k - x_i]\!]_j^{V,i} \right),$$

where Concat refers to the concatenation of sequences.

The idea behind these functions is the following: for all possible changes in the translation invariant dimensions, Subst contains a substitution sequence for the BDD variables in the translation invariant dimensions. For a dislocation vector $(x_1, \ldots, x_d)$, the corresponding substitution sequence modifies the value of every state $(x'_1, \ldots, x'_d)$ to $(x'_1 + x_1, \ldots, x'_d + x_d)$.

Function Subst_i gives a suitable substitution for a single dimension and a single dislocation value for this dimension, while Subst computes substitution sequences for all possible dislocation combinations. The former of these functions iterates over all bits in the binary encoding for dimension d, where $[\![\cdot]\!]_j^{V,i}$ refers to encoding the jth bit of the value passed to the operator using the Boolean variable $b_j^{V,i}$.

Using the set Subst, we can now give the definition of the new EnfPre operator:

$$\mathrm{EnfPre}(f) = \exists \mathcal{B}_A . \forall \mathcal{B}_{V'} . g(f) \wedge f_{E^0},$$

where we have

$$g(f) = \bigwedge_{s_1, s'_1, \ldots, s_l, s'_l \in \mathsf{Subst}} (f[s_1/s'_1, s_2/s'_2, \ldots, s_l/s'_l] \vee \neg f_{E^1}) \, .$$

The final EnfPre operator definition is essentially the same as the classical definition except that combining f_{E^1} and f has been delegated to the function $g(f)$. With the new definition, a position/action combination can only be a model of $g(f)$ if for all possible dislocations that the action can lead to, the resulting state is a model of f or f_{E^1} does not contain the corresponding non-translation invariant position component change.

IV. Experiments

We consider three different system dynamics to evaluate the game solving performance with the classical version of the EnfPre operator and the new version from this paper:

- A slowly rotating single-speed vehicle dynamics with three dimensions (where the X and Y position dimensions are translation invariant, and the current rotation is not). The only control input is the change in rotation.
- An inertia-free vehicle dynamics from [4], which has the same state dimensions as the first dynamics. Control inputs are the change in rotation and the current speed.
- A simple moon lander, which is again translation invariant in the X and Y position dimensions, but not in the other two dimensions that represent the current vertical and horizontal speed values. There are two control inputs to alter the vertical and horizontal speed values.

The details of the first and the third dynamics are given in [14]. We apply these dynamics for control problems in workspaces that, in the X and Y dimensions, have sizes from 16×16 to 64×64 *cells* in the abstraction, where we call the set of states corresponding to one (x, y)-location in the workspace a cell. Table I contains some further information on the abstractions used for this experimental evaluation.

For each of the 480 benchmarks built from these abstractions, a couple of random static obstacles are added to a blank workspace, and target regions for a patrolling task are generated randomly. Thus, the controller has to patrol between the regions while avoiding to collide with obstacles or the workspace boundaries. For the single-speed vehicle dynamics, the random challenge problem generation process lead to 178 challenge problems that permit controllers that ensure that the specification holds, whereas for 32 challenge problems, no such controllers exist. For the inertia-free vehicle, 13 problems permit controllers, while 137 do not. For the moon lander, the corresponding generation process lead to 32 challenge problems that permit a controller, whereas 88 do not.

The implementation of the game solver, all parameters of the physical systems and the abstractions, and the benchmarks can be obtained from `https://progirep.github.io/bestirs`. The game solver uses the `CuDD` binary decision diagram library [15] by Fabio Somenzi. We did not implement a detection scheme for translation-invariant dimensions, but rather considered the list of such dimensions to be an input to the tool.

Figure 3 shows a *cactus plot* for the overall game solving performance on all benchmarks, which states for each algorithm variant how many benchmarks were solved within a given time bound. All experiments were performed on a computer with an AMD Opteron(tm) 2220 SE processor running an x64 version of Linux, where each run had 3 GB of RAM available. Due to the importance of automatic variable reordering in BDDs, we performed all experiments three times: with the default variable order (where predecessor and successor state variables interleave), with automatic variable reordering turned on, and a third time in which the final order from the runs with reordering enabled is used as static variable order.

It can be seen that for all three choices for handling the variable order, the computation times improved substantially with our new operator implementation (note that the time axis is logarithmic). For the most difficult problems, the classical operator implementation is able to catch up. A possible reason for this effect is that the `CuDD` library does not cache intermediate results of substitution operations, whereas all intermediate results can be cached for the classical EnfPre implementation. The more recent BDD library `Sylvan` [16] can also cache substitution operations, but has not been used for this study as it does currently not support

TABLE I
Information on the system dynamics abstractions considered in this paper.

System Dynamics	**Slowly rotating vehicle**	**Inertia-free vehicle dynamics from [4]**	**Moon lander**
Number of states per cell	64	35	11×16
Number of different control inputs	11	7×7	7×8
Number of transitions per cell (encoded into f_{E^1})	16290	21613	246126

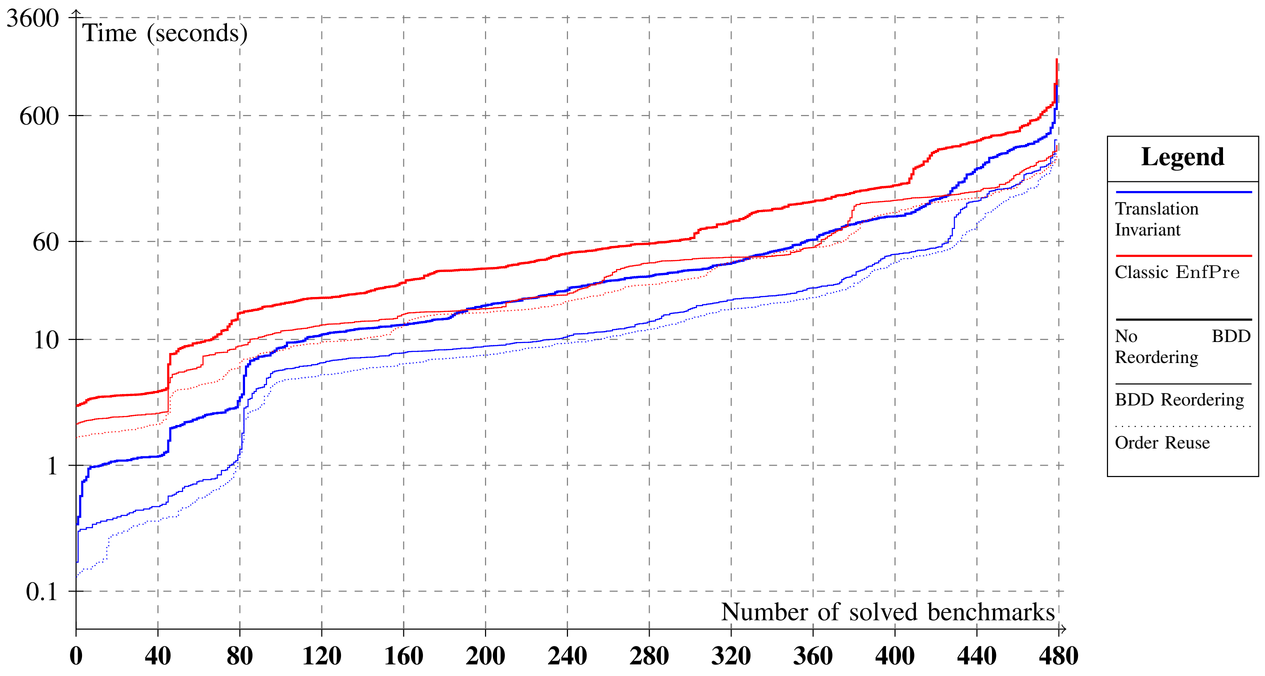

Fig. 3. Cactus plot comparing the different enforceable precedessor operator implementations.

automatic variable reordering. Exploiting better caching of the substitution operation is hence left as future work.

More details on the experiments are also available on `https://progirep.github.io/bestirs`.

V. Conclusion

We have presented a new approach to implementing the enforceable predecessor operator for solving symbolically represented games. We targeted those games that model the controller synthesis problem for a physical system using an abstraction of its dynamics. Our approach exploits the locality of state changes in translation invariant dimensions, which are a common occurrence in control applications.

Our experiments show that the new implementation is substantially faster on problems of moderate complexity, and as good as the traditional operator implementation on more complex control problems. The performance of the new approach can be further improved by computing abstractions that are more *local*, i.e., for which the relative changes in the translation invariant dimensions are small, as this reduces the size of the Disloc set in the operator definition. When computing abstractions of time-continuous systems, this can be achieved by carefully selecting a suitable time step. Optimizing abstractions in this way is left for future work, and since we showed that the new abstraction operator is similarly efficient as the classical one, this approach is promising to further increase game solving efficiency.

References

[1] S. Sadraddini and C. Belta, "Formal methods for adaptive control of dynamical systems," in *56th IEEE Annual Conference on Decision and Control (CDC)*, 2017, pp. 1782–1787.

[2] L. Winterer, S. Junges, R. Wimmer, N. Jansen, U. Topcu, J. Katoen, and B. Becker, "Motion planning under partial observability using game-based abstraction," in *56th IEEE Annual Conference on Decision and Control (CDC)*, 2017, pp. 2201–2208.

[3] M. Benerecetti and M. Faella, "Automatic synthesis of switching controllers for linear hybrid systems: Reachability control," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 4, pp. 104:1–104:27, 2017.

[4] M. Rungger and M. Zamani, "SCOTS: A tool for the synthesis of symbolic controllers," in *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control (HSCC)*, 2016, pp. 99–104.

[5] T. A. Henzinger, B. Horowitz, and R. Majumdar, "Rectangular hybrid games," in *10th International Conference on Concurrency Theory (CONCUR)*, 1999, pp. 320–335.

[6] G. Reissig, A. Weber, and M. Rungger, "Feedback refinement relations for the synthesis of symbolic controllers," *IEEE Trans. Automat. Contr.*, vol. 62, no. 4, pp. 1781–1796, 2017.

[7] P. Tabuada, *Verification and Control of Hybrid Systems - A Symbolic Approach*. Springer, 2009.

[8] C. Belta, B. Yordanov, and E. A. Gol, *Formal Methods for Discrete-Time Dynamical Systems*. Springer, 2017.

[9] P. Roy, P. Tabuada, and R. Majumdar, "Pessoa 2.0: a controller synthesis tool for cyber-physical systems," in *14th ACM International Conference on Hybrid Systems: Computation and Control (HSCC)*, 2011, pp. 315–316.

[10] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[11] R. Mazala, "Infinite games," in *Automata, Logics, and Infinite Games: A Guide to Current Research*, 2001, pp. 23–42.

[12] S. Jacobs, R. Bloem, R. Brenguier, R. Ehlers, T. Hell, R. Könighofer, G. A. Pérez, J. Raskin, L. Ryzhyk, O. Sankur, M. Seidl, L. Tentrup, and A. Walker, "The first reactive synthesis competition (SYNTCOMP 2014)," *STTT*, vol. 19, no. 3, pp. 367–390, 2017.

[13] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," *J. Comput. Syst. Sci.*, vol. 78, no. 3, pp. 911–938, 2012.

[14] F. P. Romero and R. Ehlers, "Steady abstractions for CPS controller synthesis," in *2018 Annual American Control Conference (ACC)*, 2018, pp. 778–785.

[15] F. Somenzi, "CUDD: CU Decision Diagram package release 3.0.0," 2016.

[16] T. van Dijk and J. van de Pol, "Sylvan: multi-core framework for decision diagrams," *STTT*, vol. 19, no. 6, pp. 675–696, 2017.