Nonuniform abstractions, refinement and controller synthesis with novel BDD encodings ⋆

Oscar Lindvall Bulancea * **Petter Nilsson** ** **Necmiye Ozay** ***

* *KTH Royal Institute of Technology, 100 44, Stockholm, Sweden (e-mail: oscarlb@kth.se)*
** *California Institute of Technology, Pasadena, CA 91125, USA (e-mail: pettni@caltech.edu)*
*** *University of Michigan, Ann Arbor, MI 48109, USA (e-mail: necmiye@umich.edu)*

Abstract: This paper presents a control synthesis algorithm for dynamical systems to satisfy specifications given in a fragment of linear temporal logic. It is based on an abstraction-refinement scheme with nonuniform partitions of the state space. A novel encoding of the resulting transition system is proposed that uses binary decision diagrams for efficiency. We discuss several factors affecting scalability and present some benchmark results demonstrating the effectiveness of the new encodings. These ideas are also being implemented on a publicly available prototype tool, ARCS, that we briefly introduce in the paper.

Keywords: Control synthesis, abstraction-refinement

1. INTRODUCTION

Automatic synthesis of embedded control software that meets its specifications by construction provides a rigorous means for the design of cyber-physical control systems. Abstraction-based techniques, where one creates a finite transition system (FTS) corresponding to the continuous or hybrid system to be controlled, and solves a discrete control synthesis problem, has attracted considerable attention in the past decade (Tabuada, 2009; Belta et al., 2017).

Various software tools have been developed for correct-by-construction control synthesis. These tools differ by the class of systems (e.g., discrete-time vs. continuous-time; linear, piecewise affine or nonlinear) or specifications that they can handle (e.g., simple safety or reachability (Mazo Jr et al., 2010; Rungger and Zamani, 2016); expressive fragments of linear temporal logic (Wongpiromsarn et al., 2011; Filippidis et al., 2016)), the abstraction techniques that are used (e.g., uniform grid-based (Rungger and Zamani, 2016; Mazo Jr et al., 2010), multi-scale (Mouelhi et al., 2013), or partition-based (Filippidis et al., 2016)), the way they represent the FTSs internally (symbolic or explicit) and the synthesis techniques implemented.

The key challenge in control synthesis is scalability. Factors affecting scalability include the number of discrete states in the FTS, efficiency of computation of transitions between the discrete states based on continuous dynamics, the representation of the FTS, the complexity of the specification, and the complexity of the resulting controller. For instance, structural properties of dynamics such as linearity or monotonicity (Coogan and Arcak, 2015) are shown to make the computation of transitions easy. Sparsity of the dynamics has recently been exploited together with binary decision diagrams (BDDs), a compact (i.e., memory-efficient) representation of FTSs, to obtain abstractions efficiently (Gruber et al., 2017). However, it is unclear how optimizing different factors individually would affect the efficiency of solving the end-to-end synthesis problem.

This paper builds on the abstraction-refinement based incremental synthesis approach by Nilsson et al. (2017) that handles a slightly more general class of specifications than most of the earlier tools listed above. In particular, the class includes safety, recurrence and persistence components, while allowing augmented finite transition systems as the discrete model, thereby handling fairness-like assumptions. To mitigate the state-explosion problem, a nonuniform partition of the continuous state space is used. The main contribution of the present paper is a novel BDD encoding of the states that takes into account the topology of the partition and that makes it convenient to add new states in the refinement process while preserving structure. The effectiveness of the new encoding is demonstrated with examples. A prototype tool, ARCS, that implements some of these ideas is also introduced.

2. OVERVIEW

In this section we formally state the control synthesis problem and give an overview of the solution methodology.

The first ingredient of the synthesis problem is a dynamical system model

$$x^+ = f(x, u, d), \tag{1}$$

⋆ This work is supported in part by DARPA grant N66001-14-1-4045, and NSF grants CNS1446298 and ECCS-1553873. For the extended version, see Lindvall Bulancea et al. (2018).

where $x \in \mathcal{X}$ is the state, $u \in \mathcal{U}$ is the control input, and $d \in \mathcal{D}$ is the disturbance. The notation x^+ either stands for the value of x in the next time step in the discrete-time setting or the derivative in the continuous-time setting. The second ingredient is the specification, which in this paper is restricted to the following fragment of linear temporal logic:

$$\varphi = \Box A \wedge \Diamond\Box B \wedge \left(\bigwedge_{i \in I} \Box\Diamond G^i \right), \tag{2}$$

where A, B, and G^i's are propositions that indicate the membership of the state in a certain subset of the state space $\mathcal{X}$. The specification φ roughly mandates that the state trajectory of (1) should never leave the states indicated by A; that it should eventually reach the states indicated by B and remain there indefinitely; and that it should visit the states indicated by each G^i infinitely often. For details of the semantics of LTL, we refer the reader to Baier and Katoen (2008). Given these ingredients, the control synthesis problem can be stated as follows.

Problem 1. Given a system of the form (1) and a specification of the form (2), find a **control policy** μ, possibly dependent on the state history, and a set of initial conditions $\mathcal{X}_0 \subset \mathcal{X}$ called the **winning set**, such that all closed-loop trajectories starting in $\mathcal{X}_0$ satisfy φ.

Ideally, we want to solve for the largest winning set $\mathcal{X}_0$ but since this is in general hard, we opt for trying to incrementally expand it until it is large enough, or a certificate for its maximality is obtained.

Our solution methodology consists of the following steps:

(1) Construction of an abstraction in the form of a (augmented) FTS: This step requires partitioning the state space into cells based on propositions, finding transitions among cells, and, in case of an augmented FTS, finding transient cells for progress groups.
(2) Representing the abstraction using BDDs: This step requires deciding on an encoding of the states and representing transitions and progress groups as a BDD as they are generated in step (1).
(3) Discrete synthesis on the abstraction via fixed-point algorithms that generate a winning set.
(4) Refining the abstraction: If the winning set is not satisfactory, additional cells are added to the partition using the information from the fixed points in step (3). The BDD representation of the abstraction is updated accordingly with new states, transitions, and possibly progress groups.
(5) Extracting a controller from the resulting fixed point.

For the first step, we follow the optimization-based procedures in Nilsson et al. (2017). Our tool ARCS currently supports polynomial f for the dynamics, a finite set $\mathcal{U}$ for the inputs, and rectangular sets for $\mathcal{X}$ and $\mathcal{D}$. It can however be extended to any setting where computing or over-approximating reachable sets (required for encoding transitions in the FTS) and certifying transience properties (required for progress groups in an augmented FTS) are possible. This paper is primarily concerned with steps (2) through (5) above; in the rest of the paper we introduce our novel ideas and demonstrate resulting computational gains.

3. REPRESENTING THE ABSTRACT SYSTEM

As mentioned in earlier sections the continuous system (1) is abstracted to an FTS. Formally, an FTS is a tuple $\mathcal{T} = (\mathcal{Q}, \mathcal{U}, \rightarrow_{\mathcal{T}}, \mathcal{L})$, where $\mathcal{Q}$ is a finite number of states, $\mathcal{U}$ is a finite number of inputs, $\rightarrow_{\mathcal{T}} \subset \mathcal{Q} \times \mathcal{U} \times \mathcal{Q}$ is a transition relation, and $\mathcal{L}$ is a labeling function mapping each state in $\mathcal{Q}$ to a subset of propositions appearing in the formula (2). In order to find a control policy $\mu_{\mathcal{T}}$ for the FTS $\mathcal{T}$, we need to represent it with a data structure suitable for both storage (memory efficiency) and processing (time efficiency). In what follows, we discuss different representations available in ARCS, their advantages and disadvantages, with a particular focus on a novel BDD encoding.

3.1 List Representation

The perhaps most obvious way to represent an FTS is by encoding states and actions as integers, and transitions as an array

$$\mathcal{L} = \left[(q_0, u_0, q_0'), (q_1, u_1, q_1'), \ldots, (q_{|\mathcal{T}|}, u_{|\mathcal{T}|}, q_{|\mathcal{T}|}') \right]. \tag{3}$$

Evidently, this choice requires $\mathcal{O}(|\mathcal{T}|)$ memory and standard array operations such as access, insertion, search, and deletion can be done in at most $\mathcal{O}(|\mathcal{T}|)$ time, where $|\mathcal{T}|$ is the number of transitions in the system.

Remark 1. Eq. (3) can be viewed as a representation of a sparse matrix in coordinate (COO) format, that has non-zero entries q_k' at positions (q_k, u_k). There are several other methods for sparse matrix representations that have different benefits. For instance, the compressed sparse row (CSR) format allows for efficient matrix-vector product computation. ■

This representation is very simple and thus easy to implement, and scales linearly in both space and time. Although this might seem like an acceptable complexity, the size of a grid-based abstraction $\mathcal{T}$ scales at least exponentially with the dimension of the concrete system. To illustrate the potential for improvement consider an n-dimensional linear system $\dot{x} = Ax$: it requires n^2 numbers (the entries of A) to represent in its canonical ODE form, whereas the size of a finite abstraction based on a list scales exponentially with n. This results from the fact that the semantics of an ODE encodes more side information than the semantics of a transition system, thus allowing the former to be more succinct. The idea of this paper is to explore whether information such as geometrical relationships embedded in an ODE can be stored as part of the encoding of $\mathcal{T}$ by working with more sophisticated representations.

3.2 Binary Decision Diagrams

In this section, we present an alternative representation of transition systems based on Binary Decision Diagrams (BDDs). We briefly overview how certain operations on transition systems can be performed with BDDs. A key design choice for this type of representations is how to encode the state and action sets as binary variables. As the main contribution of the paper, we present a novel choice for the encodings that attempts to capture underlying geometrical relationships.

A BDD is a data structure for representing *boolean functions*

$$B : \{0,1\}^n \to \{0,1\}, \tag{4}$$

taking binary variables, $z_1, \dots, z_n$, defined with an order of evaluation $z_1 < \cdots < z_n$. To represent a finite set C with a BDD, one needs an *encoding* $E : C \to \{0,1\}^n$, an injective map from elements of C to truth assignments of the variables $\{z_i\}$, following the order of evaluation. The boolean function B_C is said to represent the set C if such an encoding E is defined on all possible elements c and

$$C = \{c : B_C(E(c)) = 1\}, \tag{5}$$

i.e. the BDD forms the characteristic function of the set (Bryant, 1992). For a given encoding such a function can easily be obtained for singletons: if the encoding of q_k is the binary array $E(q_k) = (b_{k,1}, b_{k,2}, \dots, b_{k,n})$, the boolean function for that element can be constructed as

$$B_{q_k}(z_1, z_2, \dots, z_n) = \bigwedge_{i=1}^{n} \left\{ \begin{matrix} z_i & b_{k,i} = 1 \\ \overline{z}_i & b_{k,i} = 0 \end{matrix} \right\}, \tag{6}$$

where $\overline{z}_i$ denotes negation of the variable z_i. Then a boolean function for the whole set C can be formed as $B_C = \bigvee_{k=1}^{n} B_{q_k}$.

To construct BDDs for the elements of a transition mapping $\to_{\mathcal{T}}$, an encoding has to be chosen to represent elements $(q_k, u_k, q'_k) \in \mathcal{Q} \times \mathcal{U} \times \mathcal{Q}$. To construct singleton BDDs according to (6), one needs to separate the logical variables for the different parts of the elements, while also separating those used for the initial and final transition states q_k and q'_k. Therefore $2n+m$ variables are defined: $z_{q,1}, \dots, z_{q,n}$ to represent the set initial states $\mathcal{Q}$, $z_{u,1}, \dots, z_{u,m}$ for the action set, and $z_{q',1}, \dots, z_{q',n}$ for the final state set $\mathcal{Q}'$. Having defined encodings $E_{\mathcal{Q}}$ and $E_{\mathcal{U}}$ for the set of states $\mathcal{Q}$ and set of actions $\mathcal{U}$, an encoding for the transition can be chosen as

$$E_{\mathcal{T}}(q_k, u_k, q'_k) = (E_{\mathcal{Q}}(q_k), E_{\mathcal{U}}(u_k), E_{\mathcal{Q}}(q'_k)). \tag{7}$$

With such an encoding, the BDD for one transition $t = (q_k, u_k, q'_k)$ can be constructed as in (6):

$$B_t = B_{q_k} \wedge B_{u_k} \wedge B_{q'_k}. \tag{8}$$

A disjunction among such transitions gives the BDD representing the entire set $\to_{\mathcal{T}}$.

Using BDDs is analogous to working with the sets themselves, as the basic set operations $\{\cup, \cap, \setminus\}$ have as counterparts logical operations on the corresponding BDDs. It follows from (5) that for sets C and D represented by BDDs B_C and B_D, we have $B_{C\cup D} = B_C \vee B_D$, $B_{C\cap D} = B_C \wedge B_D$, and $B_{C\setminus D} = B_C \wedge \neg B_D$. Similar to logical operators, set descriptions involving existential and universal quantifiers can be represented with BDDs as:

$$\exists_{z_1} B(z_1, z_2) = B(0, z_2) \vee B(1, z_2), \tag{9}$$

$$\forall_{z_1} B(z_1, z_2) = B(0, z_2) \wedge B(1, z_2). \tag{10}$$

The BDD equivalent of quantification over elements of a set is to use logical quantification over all variables used in describing the corresponding set BDD. We therefore define the quantifiers $\exists_C = \exists_{z_1}\exists_{z_2}\dots\exists_{z_n}$, and $\forall_C = \forall_{z_1}\forall_{z_2}\dots\forall_{z_n}$, where the variables $\boldsymbol{z} = (z_1, \dots, z_n)$ are used to describe the elements in a set C.

For the operations above, the complexity as functions of the size of BDDs involved is as follows: Conjunction/disjunction of two BDDs B_1 and B_2 requires $\mathcal{O}(|B_1||B_2|)$ time, producing a BDD with the same bound in size (Bryant, 1986). Negation and assignment of a number of variables in a BDD B requires $\mathcal{O}(|B|)$ time. Negation does not change the BDD size, but the size after variable assignment is bounded by the original BDD size $|B|$ (Bryant, 1992). Regarding (9) and (10), quantification of a single variable on a BDD B, can be achieved in $\mathcal{O}(|B|^2)$ and results in a size bounded by $\mathcal{O}(|B|^2)$.

Each operation only takes time, and produces a BDD of size, that is polynomial in the input sizes, but successive applications of these operations are required in BDD manipulation. For instance, the quantifications $\exists_C$ and $\forall_C$ have a worst-case complexity that is exponential in the input size. However, these worst-case complexities are seldom encountered in practice.

The BDD data structure is based on a reduced binary tree whose size, i.e. number of nodes, varies not only with the number of elements it represents but also with the encodings and the evaluation order defined for the variables. Choices of evaluation order and encodings are vital, as time and memory used by the logical operations are dependent on the size of the BDD structures involved (Bryant, 1986).

As for the choice of variable ordering, an optimal choice can result in a BDD of size linear in the number of binary variables, and a bad choice can give a size exponential in the number of variables (Bryant, 1986). Finding the optimal variable ordering is a computationally hard problem (Bollig and Wegener, 1996) and cannot be solved exactly for any large number of variables in reasonable time, although several heuristics exist (e.g. Bollig et al. (1995)).

The choice of element encoding E involves two aspects: The number of variables used in the encoding, and how each element is mapped. We investigate two kinds of encodings for the states in the abstracted system. One memory-efficient encoding that minimizes the number of variables used, and one encoding that attempts to capture the structure of the partition after iterated refinement.

State Encodings: The first type of encoding, which we refer to as the *log encoding*, assumes a numbering of the states from 1 to some number $|\mathcal{Q}|$ and uses these to define the mapping in the form

$$E_{\log} : \begin{cases} \mathcal{Q} \to \{0,1\}^n, \\ q \mapsto \mathrm{Bin}(q-1), \end{cases} \tag{11}$$

where $\mathrm{Bin}(q)$ is the binary representation of the number q. When the state space $\mathcal{Q}$ is expanded at refinement, a state with number k is split into two new states. One of these is numbered by k and the other by $|\mathcal{Q}|+1$, after which they are encoded according to (11). The number of variables is also incremented if $|\mathcal{Q}| = 2^n$ before refinement, i.e. all encodings for n variables are used by the present states. With this encoding, the absolute minimum of $\lceil \log_2(|\mathcal{Q}|) \rceil$ variables are used to encode the states. As it simply uses the least amount of variables, it is the encoding to prefer when nothing obvious can be stated about the structure of the problem. This is the standard encoding used in some tools (Filippidis et al., 2016; Gruber et al., 2017).

The novel encoding we propose in this paper—denoted the *split encoding*—is based on the splitting procedure during refinement. As the partition grows increasingly non-uniform, with a possibly small area becoming increasingly fine in contrast to others, we believe that an encoding that reflects this structure can lead to computational gains.

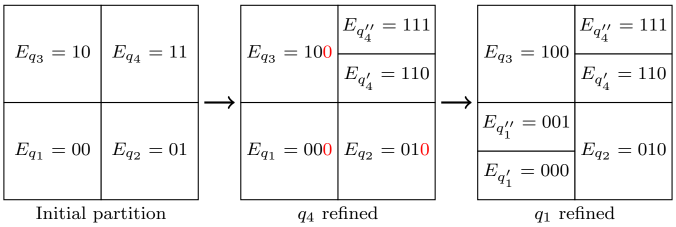

Fig. 1. Example of change in split encoding after refinement of initial partition using $k = 2$ variables. Refinement of q_4 also reaches new largest refinement depth. Bits appended during expansion, but not later assigned, are shown in red.

Starting with a coarse initial abstraction and an encoding E for the states using k variables, new states created from the splitting procedure have their encoding chosen based on that of its predecessor and refinement depth. The refinement depth is a measure of how many refinements have been performed on the domain the state contains. Every cell resulting from splitting a cell with depth d has a depth of $d+1$, and cells in the initial partition are defined to have depth zero. When a state q with depth d is split into two others, q', q'', the new states keep the encoding of their predecessor, with a modified bit at position $k+d+1$. This bit is set to 0 for q' and to 1 for q''. In the event that the partition reaches a new largest splitting depth, a new variable first has to be created to describe the new states, effectively expanding all encodings by one bit. The default value of the appended bit is chosen as 0. An example of both cases is shown in Fig. 1. If the states of the initial partition are given unique initial encodings before refinement, this procedure also results in a unique assignment of state encodings.

To summarize, having a partition with an encoding E and largest refinement depth D; the state q with a refinement depth d and encoding $E(q) = (b_{q,1}, \dots, b_{q,k}, \dots, b_{q,k+D})$, is split into two states labeled q' and q''. The encodings of the new states are chosen as

$$E(q') = E(q)\big|_{b_{k+d+1}=0}, \tag{12}$$

$$E(q'') = E(q)\big|_{b_{k+d+1}=1}. \tag{13}$$

But if $d = D$, then all encodings are first expanded by a 0 bit, i.e.

$$E(q) = (E(q), 0) \quad \forall q \in \mathcal{Q}, \tag{14}$$

before applying (12) and (13) with the newly expanded encodings.

With this choice of encoding, neighboring cells have similar encodings and our hypothesis is that such similarity admits a compact BDD representation by capturing the geometry of the underlying vector fields. Smaller BDDs typically result in computational gains in the synthesis step, an effect we see in simulations. However, the number of binary variables is generally larger than with the log encoding so the theoretical worst-case complexity is higher.

4. SOLVING SYNTHESIS PROBLEMS

The general way of solving a finite LTL synthesis problem involves translating the LTL specification to a Rabin automaton and computing fixed points on the product of the transition system and the Rabin automaton. Certain LTL fragments do however admit winning sets defined as fixed points on the transition system, which avoids the potentially expensive construction of the product system. This is the case for the GR(1) fragment, as well as the fragment (2) considered in this paper. In the following, we present the fixed-point mappings associated with (2) and how they can be evaluated symbolically when sets are represented as BDDs.

The fundamental component of the fixed-point mappings is the backwards controlled reachability operator $\mathrm{Pre}_{\sharp_1,\sharp_2}$ defined as follows:

$$\mathrm{Pre}_{\sharp_1,\sharp_2}(X) = \{q : (\sharp_1 u \in \mathcal{U}), (\sharp_2 (q,u,q') \in \to_{\mathcal{T}}), q' \in X\}. \tag{15}$$

Here, $\sharp_1$ and $\sharp_2$ is either $\exists$ or $\forall$ and reflects the controllability assumptions: $\mathrm{Pre}_{\exists,\forall}$ corresponds to u being controllable and nondeterminism uncontrollable, whereas $\mathrm{Pre}_{\forall,\exists}$ corresponds to uncontrollable u but controllable nondeterminism.

Computing Pre *with list representation:* For synthesis algorithms the fundamental operator is the $\mathrm{Pre}_{\exists,\forall}(X)$ operator, which can be computed as follows:

(1) Find the set $C = \mathrm{Pre}_{\exists,\exists}(X)$ of all q such that there exists $(q,u,q') \in \to_{\mathcal{T}}$ for $q' \in X$.
(2) For each $q \in C$, for each action u, find the set $C_{q,u} = \{q' : \exists (q,u,q') \in \to_{\mathcal{T}}\}$.
(3) Now for all q, $q \in \mathrm{Pre}_{\exists,\forall}(X)$ if and only if $q \in C$ and for some u, $C_{q,u} \neq \emptyset$ and $C_{q,u} \subset X$.

The procedure can be slightly modified to account for other combinations of quantifiers (i.e. $\forall, \forall$). The first step can be done via one traversal of $\mathcal{L}$, and the second via $|C||\mathcal{U}|$ traversals. Thus the complexity for computing Pre is upper bounded by $\mathcal{O}(|\mathrm{Pre}_{\exists,\exists}(X)||\mathcal{U}||\mathcal{T}|)$. However, since the same sets $C_{q,u}$ are typically computed many times when evaluating a fixed point, step 2 can be boosted by storing the sets $C_{q,u}$, which improves the time complexity of a single Pre computation towards $\mathcal{O}(|\mathcal{T}|)$ at the expense of a larger memory footprint.

Computing Pre *with BDD representation:* When the set of final states X is represented as a BDD B_X using an encoding E, the set $\mathrm{Pre}_{\sharp_1,\sharp_2}(X)$ can be represented as the binary mapping

$$B_{\mathrm{Pre}_{\sharp_1,\sharp_2}(X)} = \begin{cases} \sharp_{1\mathcal{U}} \exists_{\mathcal{Q}'} (B_{\mathcal{T}} \wedge B_X), & \sharp_2 = \exists \\ \sharp_{1\mathcal{U}} \forall_{\mathcal{Q}'} (\neg B_{\mathcal{T}} \vee B_X), & \sharp_2 = \forall \end{cases}, \tag{16}$$

which can be computed symbolically from B_X via quantifier elimination. [1] The run time of this operation ultimately depends on the sizes of intermediate results, but considering the complexity and worst-case size result of

[1] The domain of possible assignments could be larger than the domain of elements. As such, one would need to modify (16), to $B_{\mathcal{Q}} \wedge \sharp_{\mathcal{U}} (\forall_{\mathcal{Q}'} (\neg B_{\mathcal{T}} \vee B_X))$ if $\sharp_2 = \forall$, further replace $\exists_{\mathcal{U}}(\dots)$ with $\exists_{\mathcal{U}}(B_{\mathcal{U}} \wedge \dots)$ if also $\sharp_1 = \exists$, and if $(\sharp_1, \sharp_2) = (\forall, \exists)$ replace $\forall_{\mathcal{U}}(\dots)$ with $\forall_{\mathcal{U}}(\neg B_{\mathcal{U}} \vee \dots)$, to not include assignments corresponding to non-existent states.

each operation involved, an upper bound can be obtained as $\mathcal{O}((|B_X||B_\mathcal{T}|)^{2^{n_u+n_q}})$, when using n_u action variables and n_q end state variables.

Equipped with the Pre operator, we give fixed-point characterizations for the winning set of (2). We borrow notation from μ-calculus for succinct expression of fixed points. Let $\kappa : 2^\mathcal{Q} \to 2^\mathcal{Q}$ be a mapping that is monotone with respect to set inclusion, i.e., $V \subset W \implies \kappa(V) \subset \kappa(W)$. Then the **greatest fixed point of** κ, written $\nu V\ \kappa(V)$, is the value after convergence of the set sequence

$$V_0 = \mathcal{Q}, \quad V_{k+1} = \kappa(V_k). \tag{17}$$

Correspondingly, the **smallest fixed point of** κ, written $\mu V\ \kappa(V)$, is the value after convergence of

$$V_0 = \emptyset, \quad V_{k+1} = \kappa(V_k). \tag{18}$$

Due to monotonicity and finiteness of $\mathcal{Q}$, both these sequences converge in a finite number of steps. With this notation, the winning set of (2) is as follows:

$$\begin{aligned}\mathrm{Win}_{\exists,\forall}(\varphi) = \mu V_2\ \nu V_1 \bigcap_{i\in I} \mu V_0\ \mathrm{Pre}_{\exists,\forall}(V_2)\\ \cup \left(B \cap G^i \cap \mathrm{Pre}_{\exists,\forall}(V_1)\right) \cup \left(B \cap \mathrm{Pre}_{\exists,\forall}(V_0)\right).\end{aligned} \tag{19}$$

Specification-Guided Abstraction Refinement: In the event that the winning set $\mathrm{Win}_{\exists,\forall}(\varphi)$ computed via (19) is empty, or otherwise not satisfactory (e.g., it does not cover an expected initial condition), the abstraction can be refined in an attempt to extract more information about the underlying concrete system. Instead of doing this blindly, we select refinement regions guided by the internals of the fixed point computation (19). Loosely speaking, for a greatest fixed point (17) we perform refinement in the set $\mathrm{Pre}_{\exists,\exists}(V_\infty) \setminus V_\infty$ just outside the fixed point V_∞, with the hope that the greatest fixed point will be enlarged in the refined system. For a smallest fixed point, refinement is instead done in $V_1 \setminus V_\infty$, where V_k is the k'th iteration of (18). For multi-level fixed points such as (19) we select the refinement regions as the union of the refinement regions corresponding to lower-level fixed points. For more details see (Nilsson et al., 2017).

Controller Extraction: In addition to computing the winning set, in practice also a controller that enforces the specification inside the winning set is required. Fundamentally, such a controller can be extracted by saving the set of u's satisfying the quantification in low-level calls to Pre in (15), and storing these u's in a memory hierarchy whose structure depends on the type of fixed point.

5. RESULTS AND COMPARISONS

In this section, we present results comparing different representations of transition systems. Our toolbox ARCS, available at `https://github.com/pettni/abstr-refinement`, implements the examples in this section. ARCS has a MATLAB front-end for handling continuous dynamics, computation of transitions, and list representations, and a C back-end for BDD operations using the CUDD library (Somenzi, 2015).

As benchmarks we consider hydronic radiant systems for buildings, in which chilled water is run through concrete slabs to regulate the temperature of the rooms to which they are connected. The systems are controlled by turning on/off flow to any one slab, thus changing the temperature of zone i according to the heating dynamics

$$c_i \dot{T}_i = \sum_{j\neq i} \frac{1}{R_{ij}}(T_i - T_j) + k_i, \tag{20}$$

where the sum is taken over all temperature zones in the system, including other rooms, slabs, supply water sources, and the outside (the last two are assumed to have constant temperature). Thermal capacitances c_i, thermal resistances $R_{ij} = R_{ji}$ and nominal heat gains k_i are determined by sets of parameters that together define room types. In this article we use parameters for the outside and water temperature and two room types as defined in Nilsson et al. (2017), slightly extending the framework to take into account adjacency with multiple rooms of different types, and multiple slabs in different configurations. A setup with n_r rooms and n_s slabs results in a dynamical system with $n_r + n_s$ continuous states (room and slab temperatures), and 2^{n_s} discrete control inputs (water flow through each slab turned on or off).

We construct a collection of benchmark systems and perform different types of run time tests.For these tests a persistence specification $\varphi = \Diamond\Box B$ is considered, with B being the proposition corresponding to all rooms and slabs having temperatures in $[22, 25]^\circ C$ and $[21, 27]^\circ C$ respectively, in a total domain of $([20, 28]^\circ C)^{n_r+n_s}$.

In the first test, different configurations of adjacent rooms and connecting slabs, as shown in Fig. 3, are used. After refining the abstraction of each system 2000 times, resulting in roughly 2000 states in each abstraction, a final synthesis is performed and timed. The final synthesis is done using the BDD representation with the suggested log and split encoding, both as is and after reordering of the BDD variables using the simulated annealing algorithm implemented in CUDD. Table 1 shows the resulting run times for each system and representation scenario. It is worth noting that for all systems except 3, which is special in the sense that one room lacks a controller, the order from slowest to fastest run times is consistently measured as log, reordered log, split and reordered split encodings.

System	Log (Reordered)	Split (Reordered)	List
1	1.15 (0.515)	0.107 (0.0982)	31.4
2	0.413 (0.200)	0.159 (0.0622)	21.0
3	0.000 530 (0.000 620)	0.000 690 (0.000 340)	0.453
4	6.54 (2.24)	0.854 (0.439)	34.1
5	10.0 (2.78)	0.741 (0.664)	34.0
6	1.39 (0.456)	0.0518 (0.0350)	4.91
7	1.57 (0.361)	0.113 (0.0464)	4.78

Table 1. Comparison of stand-alone synthesis run times (in seconds) using the different versions of BDD encoding, with and without reordering.

In the second test, System 2 in Fig. 3 is used in measuring the synthesis time at different levels of refinement. An abstraction is obtained after a number of synthesis-refinement iterations between 500 and 4000, and the run time of synthesis on that abstraction is measured. In Fig. 2 (Left) the run time of synthesis is plotted against the number of transitions present in the abstraction. Just as in the first test, the split encoding is seen to have lower run time than the log encoding for both the ordered and unordered case. It is also interesting to note how for this

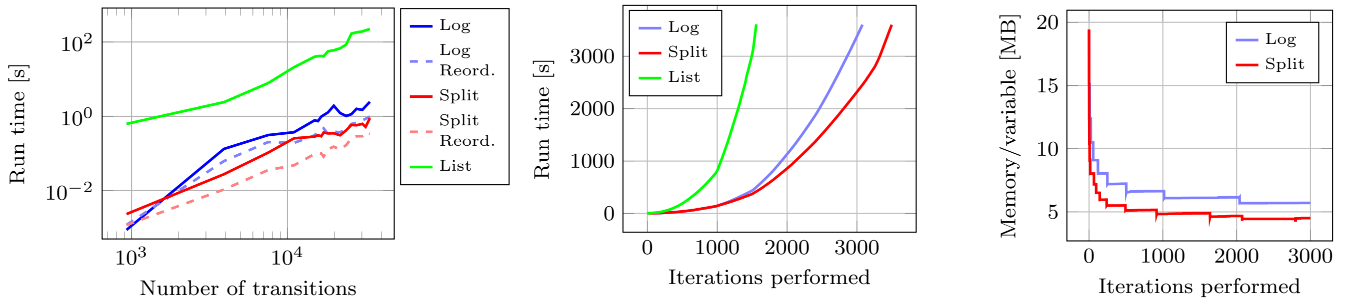

Fig. 2. Left: Comparison of stand-alone synthesis run times for System 2 (Fig. 3) using the different representations and BDD encodings, with and without reordering, for variable level of refinement. Middle: End-to-end synthesis-refinement performance test for System 2 (figure 3), letting the synthesis-refinement algorithm run for one hour, using the different representations and BDD encodings. Right: Memory used per binary variable as a function of refinement steps for System 2.

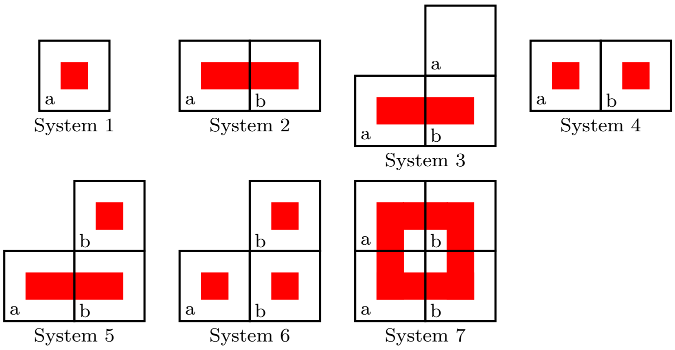

Fig. 3. Test configurations based on different building layouts. Rooms of different types (a or b) having different heating dynamics, controlled by cooling slabs (red).

test, the slope of the reordered split encoding graph in this loglog-plot is close to linear and less than that of the list encoding toward the larger number of transitions.

As a final test, the synthesis-refinement procedure is ran on System 2 for one hour using the list representation, log and split encoding, and cumulative time together with run time of each synthesis-refinement iteration is measured. In Fig. 2 (Middle), the cumulative time is plotted against the total number of iterations achieved, which shows that the split encoding manages a few hundred more iterations in total than the log encoding, especially during the time the abstraction is more refined and transition relations are more complex. Finally, we also note that in terms of memory requirements, the number of nodes allocated by CUDD to represent the BDDs for split encoding is approximately twice as much as that for the log encoding, with 2% to 10% extra memory usage at iteration steps 2000 and 3000, respectively. This extra memory usage is not surprising given that the number of binary variables in split encoding is more than the minimum number achieved by the log encoding. On the other hand, split encoding achieves a better compression in terms of memory used per number of binary variables as shown in Fig. 2 (Right). Moreover, this redundancy in representation seems to improve computation times significantly.

6. CONCLUSIONS

In this paper, we presented an abstraction-refinement based controller synthesis framework and, specifically, discussed several ways of representing the transition systems resulting from abstractions. We proposed a novel BDD-based encoding, namely *split encoding*, of the states of the transition system that takes into account the geometry of the underlying continuous states and how they evolve with refinement. A comparative study of various representations shows the effectiveness of the new encoding. The presented ideas are implemented in a toolbox, ARCS, which is made publicly available.

REFERENCES

Baier, C. and Katoen, J.P. (2008). *Principles of model checking*. MIT press.

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

Bollig, B. and Wegener, I. (1996). Improving the variable ordering of obdds is np-complete. *IEEE Trans. Comp.*, 45(9), 993–1002.

Bollig, B., Lbbing, M., and Wegener, I. (1995). Simulated annealing to improve variable orderings for obdds. In *Int'l IWLS*, 5–5.

Bryant, R.E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comp.*, (8), 677–691.

Bryant, R.E. (1992). Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3), 293–318.

Coogan, S. and Arcak, M. (2015). Efficient finite abstraction of mixed monotone systems. In *Proc. of the 18th HSCC*, 58–67. ACM.

Filippidis, I., Dathathri, S., Livingston, S.C., Ozay, N., and Murray, R.M. (2016). Control design for hybrid systems with tulip: The temporal logic planning toolbox. In *Proc. MSC*, 1030–1041. URL `tulip-control.org`.

Gruber, F., Kim, E.S., and Arcak, M. (2017). Sparsity-sensitive finite abstraction. *arXiv preprint arXiv:1704.03951*.

Lindvall Bulancea, O., Nilsson, P., and Ozay, N. (2018). Nonuniform abstractions, refinement and controller synthesis with novel bdd encodings. *arXiv preprint arXiv:1804.04280*.

Mazo Jr, M., Davitian, A., and Tabuada, P. (2010). Pessoa: A tool for embedded controller synthesis. In *Computer Aided Verification*, 566–569. Springer.

Mouelhi, S., Girard, A., and Gössler, G. (2013). Cosyma: a tool for controller synthesis using multi-scale abstractions. In *Proc. of the 16th HSCC*, 83–88.

Nilsson, P., Ozay, N., and Liu, J. (2017). Augmented finite transition systems as abstractions for control synthesis. *Discrete Event Dynamic Systems*, 27(2), 301–340.

Rungger, M. and Zamani, M. (2016). Scots: A tool for the synthesis of symbolic controllers. In *Proc. of the 19th HSCC*, HSCC '16, 99–104. ACM, New York, NY, USA.

Somenzi, F. (2015). Cudd: Cu decision diagram package release 3.0.0. URL `http://vlsi.colorado.edu/~fabio/CUDD`.

Tabuada, P. (2009). *Verification and control of hybrid systems: a symbolic approach*. Springer Science & Business Media.

Wongpiromsarn, T., Topcu, U., Ozay, N., Xu, H., and Murray, R.M. (2011). TuLiP: a software toolbox for receding horizon temporal logic planning. In *HSCC*, 313–314. URL `http://www.tulip.org`.