Flexible Computational Pipelines for Robust Abstraction-Based Control Synthesis

Eric S. Kim(✉), Murat Arcak, and Sanjit A. Seshia

UC Berkeley, Berkeley, CA, USA
{eskim,arcak,sseshia}@eecs.berkeley.edu

Abstract. Successfully synthesizing controllers for complex dynamical systems and specifications often requires leveraging domain knowledge as well as making difficult computational or mathematical tradeoffs. This paper presents a flexible and extensible framework for constructing robust control synthesis algorithms and applies this to the traditional abstraction-based control synthesis pipeline. It is grounded in the theory of relational interfaces and provides a principled methodology to seamlessly combine different techniques (such as dynamic precision grids, refining abstractions while synthesizing, or decomposed control predecessors) or create custom procedures to exploit an application's intrinsic structural properties. A Dubins vehicle is used as a motivating example to showcase memory and runtime improvements.

Keywords: Control synthesis · Finite abstraction · Relational interface

1 Introduction

A control synthesizer's high level goal is to automatically construct control software that enables a closed loop system to satisfy a desired specification. A vast and rich literature contains results that mathematically characterize solutions to different classes of problems and specifications, such as the Hamilton-Jacobi-Isaacs PDE for differential games [3], Lyapunov theory for stabilization [8], and fixed-points for temporal logic specifications [11,17]. While many control synthesis problems have elegant mathematical solutions, there is often a gap between a solution's theoretical characterization and the algorithms used to compute it. What data structures are used to represent the dynamics and constraints? What operations should those data structures support? How should the control synthesis algorithm be structured? Implementing solutions to the questions above can require substantial time. This problem is especially critical for computationally challenging problems, where it is often necessary to let the user *rapidly* identify and exploit structure through analysis or experimentation.

The authors were funded in part by AFOSR FA9550-18-1-0253, DARPA Assured Autonomy project, iCyPhy, Berkeley Deep Drive, and NSF grant CNS-1739816.

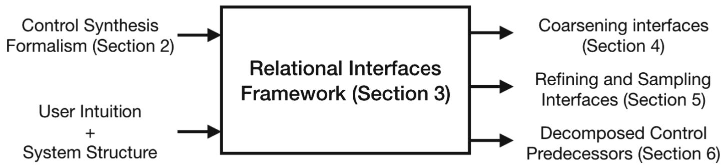

Control Synthesis Formalism (Section 2)
User Intuition + System Structure
Relational Interfaces Framework (Section 3)
Coarsening interfaces (Section 4)
Refining and Sampling Interfaces (Section 5)
Decomposed Control Predecessors (Section 6)

Fig. 1. By expressing many different techniques within a common framework, users are able to rapidly develop methods to exploit system structure in controller synthesis.

1.1 Bottlenecks in Abstraction-Based Control Synthesis

This paper's goal is to enable a framework to develop extensible tools for robust controller synthesis. It was inspired in part by computational bottlenecks encountered in control synthesizers that construct finite abstractions of continuous systems, which we use as a target use case. A traditional abstraction-based control synthesis pipeline consists of three distinct stages:

1. Abstracting the continuous state system into a finite automaton whose underlying transitions faithfully mimic the original dynamics [21,23].
2. Synthesizing a discrete controller by leveraging data structures and symbolic reasoning algorithms to mitigate combinatorial state explosion.
3. Refining the discrete controller into a continuous one. Feasibility of this step is ensured through the abstraction step.

This pipeline appears in tools PESSOA [12] and SCOTS [19], which can exhibit acute computational bottlenecks for high dimensional and nonlinear system dynamics. A common method to mitigate these bottlenecks is to exploit a specific dynamical system's topological and algebraic properties. In MASCOT [7] and CoSyMA [14], multi-scale grids and hierarchical models capture notions of state-space locality. One could incrementally construct an abstraction of the system dynamics while performing the control synthesis step [10,15] as implemented in tools ROCS [9] and ARCS [4]. The abstraction overhead can also be reduced by representing systems as a collection of components composed in parallel [6,13]. These have been developed in isolation and were not previously interoperable.

1.2 Methodology

Figure 1 depicts this paper's methodology and organization. The existing control synthesis formalism does not readily lend itself to algorithmic modifications that reflect and exploit structural properties in the system and specification. We use the theory of relational interfaces [22] as a foundation and augment it to express control synthesis pipelines. Interfaces are used to represent both system models and constraints. A small collection of atomic operators manipulates interfaces and is powerful enough to reconstruct many existing control synthesis pipelines.

One may also add new composite operators to encode desirable heuristics that exploit structural properties in the system and specifications. The last three sections encode the techniques for abstraction-based control synthesis from Sect. 1.1 within the relational interfaces framework. By deliberately deconstructing those techniques, then reconstructing them within a compositional framework it was possible to identify implicit or unnecessary assumptions then generalize or remove them. It also makes the aforementioned techniques interoperable amongst themselves as well as future techniques.

Interfaces come equipped with a refinement partial order that formalizes when one interface abstracts another. This paper focuses on preserving the refinement relation and sufficient conditions to refine discrete controllers back to concrete ones. Additional guarantees regarding completeness, termination, precision, or decomposability can be encoded, but impose additional requirements on the control synthesis algorithm and are beyond the scope of this paper.

1.3 Contributions

To our knowledge, the application of relational interfaces to robust abstraction-based control synthesis is new. The framework's building blocks consist of a collection of small, well understood operators that are nonetheless powerful enough to express many prior techniques. Encoding these techniques as relational interface operations forced us to simplify, formalize, or remove implicit assumptions in existing tools. The framework also exhibits numerous desirable features.

1. It enables compositional tools for control synthesis by leveraging a theoretical foundation with compositionality built into it. This paper showcases a principled methodology to seamlessly combine the methods in Sect. 1.1, as well as construct new techniques.
2. It enables a declarative approach to control synthesis by enforcing a strict separation between the high level algorithm from its low level implementation. We rely on the availability of an underlying data structure to encode and manipulate predicates. Low level predicate operations, while powerful, make it easy to inadvertently violate the refinement property. Conforming to the relational interface operations minimizes this danger.

This paper's first half is domain agnostic and applicable to general robust control synthesis problems. The second half applies those insights to the finite abstraction approach to control synthesis. A smaller Dubins vehicle example is used to showcase and evaluate different techniques and their computational gains, compared to the unoptimized problem. In an extended version of this paper available at [1], a 6D lunar lander example leverages all techniques in this paper and introduces a few new ones.

1.4 Notation

Let $=$ be an *assertion* that two objects are mathematically equivalent; as a special case '$\equiv$' is used when those two objects are sets. In contrast, the operator '$==$' *checks* whether two objects are equivalent, returning true if they are and false otherwise. A special instance of '$==$' is logical equivalence '$\Leftrightarrow$'.

Variables are denoted by lower case letters. Each variable v is associated with a domain of values $\mathcal{D}(v)$ that is analogous to the variable's type. A composite variable is a set of variables and is analogous to a bundle of wrapped wires. From a collection of variables $v_1, \ldots, v_M$ a composite variable v can be constructed by taking the union $v \equiv v_1 \cup \ldots \cup v_M$ and the domain $\mathcal{D}(v) \equiv \prod_{i=1}^{M} \mathcal{D}(v_i)$. Note that the variables $v_1, \ldots, v_M$ above may themselves be composite. As an example if v is associated with a M-dimensional Euclidean space $\mathbb{R}^M$, then it is a composite variable that can be broken apart into a collection of atomic variables $v_1, \ldots, v_M$ where $\mathcal{D}(v_i) \equiv \mathbb{R}$ for all $i \in \{1, \ldots, M\}$. The technical results herein do not distinguish between composite and atomic variables.

Predicates are functions that map variable assignments to a Boolean value. Predicates that stand in for expressions/formulas are denoted with capital letters. Predicates P and Q are logically equivalent (denoted by $P \Leftrightarrow Q$) if and only if $P \Rightarrow Q$ and $Q \Rightarrow P$ are true for all variable assignments. The universal and existential quantifiers $\forall$ and $\exists$ eliminate variables and yield new predicates. Predicates $\exists w P$ and $\forall w P$ do not depend on w. If w is a composite variable $w \equiv w_1 \cup \ldots \cup w_N$ then $\exists w P$ is simply a shorthand for $\exists w_1 \ldots \exists w_N P$.

2 Control Synthesis for a Motivating Example

As a simple, instructive example consider a planar Dubins vehicle that is tasked with reaching a desired location. Let $x = \{p_x, p_y, \theta\}$ be the collection of state variables, $u = \{v, \omega\}$ be a collection input variables to be controlled, $x^+ = \{p_x^+, p_y^+, \theta^+\}$ represent state variables at a subsequent time step, and $L = 1.4$ be a constant representing the vehicle length. The constraints

$$p_x^+ == p_x + v\cos(\theta) \tag{F_x}$$

$$p_y^+ == p_y + v\sin(\theta) \tag{F_y}$$

$$\theta^+ == \theta + \frac{v}{L}\sin(\omega) \tag{F_θ}$$

characterize the discrete time dynamics. The continuous state domain is $\mathcal{D}(x) \equiv [-2, 2] \times [-2, 2] \times [-\pi, \pi)$, where the last component is periodic so $-\pi$ and π are identical values. The input domains are $\mathcal{D}(v) \equiv \{0.25, 0.5\}$ and $\mathcal{D}(\omega) \equiv \{-1.5, 0, 1.5\}$

Let predicate $F = F_x \wedge F_y \wedge F_\theta$ represent the monolithic system dynamics. Predicate T depends only on x and represents the target set $[-0.4, 0.4] \times [-0.4, 0.4] \times [-\pi, \pi)$, encoding that the vehicle's position must reach a square with any orientation. Let Z be a predicate that depends on variable x^+ that encodes a collection of states at a future time step. Equation (1) characterizes the robust controlled predecessor, which takes Z and computes the set of states from which there exists a non-blocking assignment to u that guarantees x^+ will satisfy Z, despite any non-determinism contained in F. The term $\exists x^+ F$ prevents state-control pairs from blocking, while $\forall x^+(F \Rightarrow Z)$ encodes the state-control pairs that guarantee satisfaction of Z.

$$\texttt{cpre}(F, Z) = \exists u(\exists x^+ F \wedge \forall x^+(F \Rightarrow Z)). \tag{1}$$

The controlled predecessor is used to solve safety and reach games. We can solve for a region for which the target T (respectively, safe set S) can be reached (made invariant) via an iteration of an appropriate `reach` (`safe`) operator. Both iterations are given by:

$$\text{Reach Iter:} \qquad Z_0 = \bot \qquad Z_{i+1} = \texttt{reach}(F, Z_i, T) = \texttt{cpre}(F, Z_i) \vee T. \qquad (2)$$

$$\text{Safety Iter:} \qquad Z_0 = S \qquad Z_{i+1} = \texttt{safe}(F, Z_i, S) = \texttt{cpre}(F, Z_i) \wedge S. \qquad (3)$$

The above iterations are not guaranteed to reach a fixed point in a finite number of iterations, except under certain technical conditions [21]. Figure 2 depicts an approximate region where the controller can force the Dubins vehicle to enter T. We showcase different improvements relative to a base line script used to generate Fig. 2. A toolbox that adopts this paper's framework is being actively developed and is open sourced at [2]. It is written in `python 3.6` and uses the `dd` package as an interface to `CUDD` [20], a library in `C/C++` for constructing and manipulating binary decision diagrams (BDD). All experiments were run on a single core of a 2013 Macbook Pro with 2.4 GHz Intel Core i7 and 8 GB of RAM.

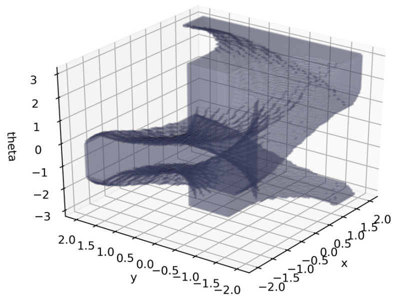


Fig. 2. Approximate solution to the Dubins vehicle reach game visualized as a subset of the state space.

The following section uses relational interfaces to represent the controlled predecessor $\texttt{cpre}(\cdot)$ and iterations (2) and (3) as a computational pipeline. Subsequent sections show how modifying this pipeline leads to favorable theoretical properties and computational gains.

3 Relational Interfaces

Relational interfaces are predicates augmented with annotations about each variable's role as an input or output[1]. They abstract away a component's internal implementation and only encode an input-output relation.

Definition 1 (Relational Interface [22]). *An interface $M(i, o)$ consists of a predicate M over a set of input variables i and output variables o.*

For an interface $M(i, o)$, we call (i, o) its input-output *signature*. An interface is a sink if it contains no outputs and has signature like $(i, \varnothing)$, and a source if it contains no inputs like $(\varnothing, o)$. Sinks and source interfaces can be interpreted as sets whereas input-output interfaces are relations. Interfaces encode relations through their predicates and can capture features such as non-deterministic outputs or

[1] Relational interfaces closely resemble assume-guarantee contracts [16]; we opt to use relational interfaces because inputs and outputs play a more prominent role.

blocking (i.e., disallowed, error) inputs. A system blocks for an input assignment if there does not exist a corresponding output assignment that satisfies the interface relation. Blocking is a critical property used to declare *requirements*; sink interfaces impose constraints by modeling constrain violations as blocking inputs. Outputs on the other hand exhibit non-determinism, which is treated as an *adversary*. When one interface's outputs are connected to another's inputs, the outputs seek to cause blocking whenever possible.

3.1 Atomic and Composite Operators

Operators are used to manipulate interfaces by taking interfaces and variables as inputs and yielding another interface. We will show how the controlled predecessor `cpre`$(\cdot)$ in (1) can be constructed by composing operators appearing in [22] and one additional one. The first, output hiding, removes interface outputs.

Definition 2 (Output Hiding [22]). *Output hiding operator* $\texttt{ohide}(w, F)$ *over interface* $F(i, o)$ *and outputs* w *yields an interface with signature* $(i, o \setminus w)$*.*

$$\texttt{ohide}(w, F) = \exists w F \tag{4}$$

Existentially quantifying out w ensures that the input-output behavior over the unhidden variables is still consistent with potential assignments to w. The operator `nb`$(\cdot)$ is a special variant of $\texttt{ohide}(\cdot)$ that hides all outputs, yielding a sink encoding all non-blocking inputs to the original interface.

Definition 3 (Nonblocking Inputs Sink). *Given an interface* $F(i, o)$*, the nonblocking operation* $\texttt{nb}(F)$ *yields a sink interface with signature* $(i, \varnothing)$ *and predicate* $\texttt{nb}(F) = \exists o F$*. If* $F(i, \varnothing)$ *is a sink interface, then* $\texttt{nb}(F) = F$ *yields itself. If* $F(\varnothing, o)$ *is a source interface, then* $\texttt{nb}(F) = \bot$ *if and only if* $F \Leftrightarrow \bot$*; otherwise* $\texttt{nb}(F) = \top$*.*

The interface composition operator takes multiple interfaces and "collapses" them into a single input-output interface. It can be viewed as a generalization of function composition in the special case where each interface encodes a total function (i.e., deterministic output and inputs never block).

Definition 4 (Interface Composition [22]). *Let* $F_1(i_1, o_1)$ *and* $F_2(i_2, o_2)$ *be interfaces with disjoint output variables* $o_1 \cap o_2 \equiv \varnothing$ *and* $i_1 \cap o_2 \equiv \varnothing$ *which signifies that* F_2*'s outputs may not be fed back into* F_1*'s inputs. Define new composite variables*

$$io_{12} \equiv o_1 \cap i_2 \tag{5}$$

$$i_{12} \equiv (i_1 \cup i_2) \setminus io_{12} \tag{6}$$

$$o_{12} \equiv o_1 \cup o_2. \tag{7}$$

Composition $\texttt{comp}(F_1, F_2)$ *is an interface with signature* (i_{12}, o_{12}) *and predicate*

$$F_1 \wedge F_2 \wedge \forall o_{12}(F_1 \Rightarrow \texttt{nb}(F_2)). \tag{8}$$

Interface subscripts may be swapped if instead F_2*'s outputs are fed into* F_1*.*

Interfaces F_1 and F_2 are composed in parallel if $io_{21} \equiv \varnothing$ holds in addition to $io_{12} \equiv \varnothing$. Equation (8) under parallel composition reduces to $F_1 \wedge F_2$ (Lemma 6.4 in [22]) and $\texttt{comp}(\cdot)$ is commutative and associative. If $io_{12} \not\equiv \varnothing$, then they are composed in series and the composition operator is only associative. Any acyclic interconnection can be composed into a single interface by systematically applying Definition 4's binary composition operator. Non-deterministic outputs are interpreted to be *adversarial*. Series composition of interfaces has a built-in notion of robustness to account for F_1's non-deterministic outputs and blocking inputs to F_2 over the shared variables io_{12}. The term $\forall o_{12}(F_1 \Rightarrow \texttt{nb}(F_2))$ in Eq. (8) is a predicate over the composition's input set i_{12}. It ensures that if a potential output of F_1 may cause F_2 to block, then $\texttt{comp}(F_1, F_2)$ must preemptively block.

The final atomic operator is input hiding, which may only be applied to sinks. If the sink is viewed as a constraint, an input variable is "hidden" by an angelic environment that chooses an input assignment to satisfy the constraint. This operator is analogous to projecting a set into a lower dimensional space.

Definition 5 (Hiding Sink Inputs). *Input hiding operator* $\texttt{ihide}(w, F)$ *over sink interface* $F(i, \varnothing)$ *and inputs* w *yields an interface with signature* $(i \setminus w, \varnothing)$*.*

$$\texttt{ihide}(w, F) = \exists w F \tag{9}$$

Unlike the composition and output hiding operators, this operator is not included in the standard theory of relational interfaces [22] and was added to encode a controller predecessor introduced subsequently in Eq. (10).

3.2 Constructing Control Synthesis Pipelines

The robust controlled predecessor (1) can be expressed through operator composition.

Proposition 1. *The controlled predecessor operator* (10) *yields a sink interface with signature* $(x, \varnothing)$ *and predicate equivalent to the predicate in* (1)*.*

$$\texttt{cpre}(F, Z) = \texttt{ihide}(u, \texttt{ohide}(x^+, \texttt{comp}(F, Z))). \tag{10}$$

The simple proof is provided in the extended version at [1]. Proposition 1 signifies that controlled predecessors can be interpreted as an instance of robust composition of interfaces, followed by variable hiding. It can be shown that $\texttt{safe}(F, Z, S) = \texttt{comp}(\texttt{cpre}(F, Z), S)$ because $S(x, \varnothing)$ and $\texttt{cpre}(F, Z)$ would be composed in parallel.[2] Figure. 3 shows a visualization of the safety game's fixed point iteration from the point of view of relational interfaces. Starting from the right-most sink interface S (equivalent to Z_0) the iteration (3) constructs a sequence of sink interfaces $Z_1, Z_2, ...$ encoding relevant subsets of the state space. The numerous $S(x, \varnothing)$ interfaces impose constraints and can be interpreted as monitors that raise errors if the safety constraint is violated.

[2] Disjunctions over sinks are required to encode $\texttt{reach}(\cdot)$. This will be enabled by the shared refinement operator defined in Definition 10.

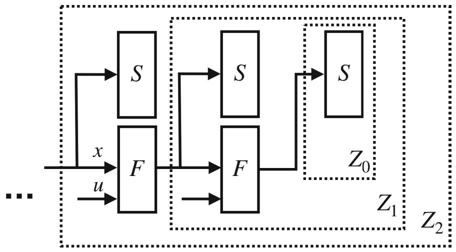


Fig. 3. Safety control synthesis iteration (3) depicted as a sequence of sink interfaces.

3.3 Modifying the Control Synthesis Pipeline

Equation (10)'s definition of `cpre`$(\cdot)$ is oblivious to the domains of variables x, u, and x^+. This generality is useful for describing a problem and serving as a blank template. Whenever problem structure exists, pipeline modifications refine the general algorithm into a form that reflects the specific problem instance. They also allow a user to inject implicit preferences into a problem and reduce computational bottlenecks or to refine a solution. The subsequent sections apply this philosophy to the abstraction-based control techniques from Sect. 1.1:

- Sect. 4: Coarsening interfaces reduces the computational complexity of a problem by throwing away fine grain information. The synthesis result is conservative but the degree of conservatism can be modified.
- Sect. 5: Refining interfaces decreases result conservatism. Refinement in combination with coarsening allows one to dynamically modulate the complexity of the problem as a function of multiple criteria such as the result granularity or minimizing computational resources.
- Sect. 6: If the dynamics or specifications are decomposable then the control predecessor operator can be broken apart to refect that decomposition.

These sections do more than simply reconstruct existing techniques in the language of relational interfaces. They uncover some implicit assumptions in existing tools and either remove them or make them explicit. Minimizing the number of assumptions ensures applicability to a diverse collection of systems and specifications and compatibility with future algorithmic modifications.

4 Interface Abstraction via Quantization

A key motivator behind abstraction-based control synthesis is that computing the game iterations from Eqs. (2) and (3) exactly is often intractable for high-dimensional nonlinear dynamics. Termination is also not guaranteed. Quantizing (or "abstracting") continuous interfaces into a finite counterpart ensures that each predicate operation of the game terminates in finite time but at the cost of the solution's precision. Finer quantization incurs a smaller loss of precision but can cause the memory and computational requirements to store and manipulate the symbolic representation to exceed machine resources.

This section first introduces the notion of interface abstraction as a refinement relation. We define the notion of a quantizer and show how it is a simple generalization of many existing quantizers in the abstraction-based control literature. Finally, we show how one can inject these quantizers anywhere in the control synthesis pipeline to reduce computational bottlenecks.

4.1 Theory of Abstract Interfaces

While a controller synthesis algorithm can analyze a simpler model of the dynamics, the results have no meaning unless they can be extrapolated back to the original system dynamics. The following interface refinement condition formalizes a condition when this extrapolation can occur.

Definition 6 (Interface Refinement [22]). *Let $F(i, o)$ and $\hat{F}(\hat{i}, \hat{o})$ be interfaces. $\hat{F}$ is an abstraction of F if and only if $i \equiv \hat{i}$, $o \equiv \hat{o}$, and*

$$\boldsymbol{nb}(\hat{F}) \Rightarrow \boldsymbol{nb}(F) \tag{11}$$

$$\left(\boldsymbol{nb}(\hat{F}) \wedge F\right) \Rightarrow \hat{F} \tag{12}$$

are valid formulas. This relationship is denoted by $\hat{F} \preceq F$.

Definition 6 imposes two main requirements between a concrete and abstract interface. Equation (11) encodes the condition where if $\hat{F}$ accepts an input, then F must also accept it; that is, the abstract component is more aggressive with rejecting invalid inputs. Second, if both systems accept the input then the abstract output set is a superset of the concrete function's output set. The abstract interface is a conservative representation of the concrete interface because the abstraction accepts fewer inputs and exhibits more non-deterministic outputs. If both the interfaces are sink interfaces, then $\hat{F} \preceq F$ reduces down to $\hat{F} \subseteq F$ when $F, \hat{F}$ are interpreted as sets. If both are source interfaces then the set containment direction is flipped and $\hat{F} \preceq F$ reduces down to $F \subseteq \hat{F}$.

The refinement relation satisfies the required reflexivity, transitivity, and antisymmetry properties to be a partial order [22] and is depicted in Fig. 4. This order has a bottom element $\perp$ which is a universal abstraction. Conveniently, the bottom element $\perp$ signifies both boolean false and the bottom of the partial order. This interface blocks for every potential input. In contrast, Boolean $\top$ plays no special role in the partial order. While $\top$ exhibits totally non-deterministic outputs, it also accepts inputs. A blocking input is considered "worse" than non-deterministic outputs in the refinement order. The refinement relation $\preceq$ encodes a direction of conservatism such that any reasoning done over the abstract models is sound and can be generalized to the concrete model.

Theorem 1 (Informal Substitutability Result [22]). *For any input that is allowed for the abstract model, the output behaviors exhibited by an abstract model contains the output behaviors exhibited by the concrete model.*

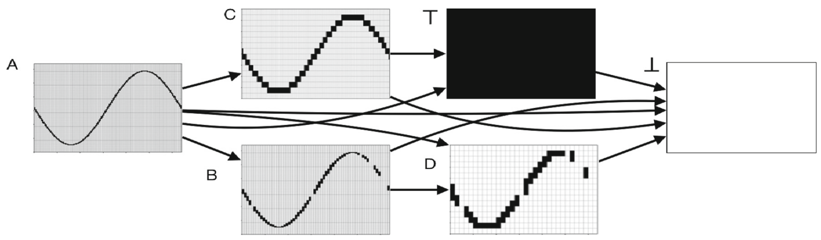


Fig. 4. Example depiction of the refinement partial order. Each small plot on the depicts input-output pairs that satisfy an interface's predicate. Inputs (outputs) vary along the horizontal (vertical) axis. Because B blocks on some inputs but A accepts all inputs $B \preceq A$. Interface C exhibits more output non-determinism than A so $C \preceq A$. Similarly $D \preceq B$, $D \preceq C$, $\top \preceq C$, etc. Note that B and C are incomparable because C exhibits more output non-determinism and B blocks for more inputs. The false interface $\bot$ is a universal abstraction, while $\top$ is incomparable with B and D.

If a property on outputs has been established for an abstract interface, then it still holds if the abstract interface is replaced with the concrete one. Informally, the abstract interface is more conservative so if a property holds with the abstraction then it must also hold for the true system. All aforementioned interface operators preserve the properties of the refinement relation of Definition 6, in the sense that they are monotone with respect to the refinement partial order.

Theorem 2 (Composition Preserves Refinement [22]**).** *Let $\hat{A} \preceq A$ and $\hat{B} \preceq B$. If the composition is well defined, then $\mathtt{comp}(\hat{A}, \hat{B}) \preceq \mathtt{comp}(A, B)$.*

Theorem 3 (Output Hiding Preserves Refinement [22]**).** *If $A \preceq B$, then $\mathtt{ohide}(w, A) \preceq \mathtt{ohide}(w, B)$ for any variable w.*

Theorem 4 (Input Hiding Preserves Refinement). *If A, B are both sink interfaces and $A \preceq B$, then $\mathtt{ihide}(w, A) \preceq \mathtt{ihide}(w, B)$ for any variable w.*

Proofs for Theorems 2 and 3 are provided in [22]. Theorem 4's proof is simple and is omitted. One can think of using interface composition and variable hiding to horizontally (with respect to the refinement order) navigate the space of all interfaces. The synthesis pipeline encodes one navigated path and monotonicity of these operators yields guarantees about the path's end point. Composite operators such as $\mathtt{cpre}(\cdot)$ chain together multiple incremental steps. Furthermore since the composition of monotone operators is itself a monotone operator, any composite constructed from these parts is also monotone. In contrast, the coarsening and refinement operators introduced later in Definitions 8 and 10 respectively are used to move vertically and construct abstractions. The "direction" of new composite operators can easily be established through simple reasoning about the cumulative directions of their constituent operators.

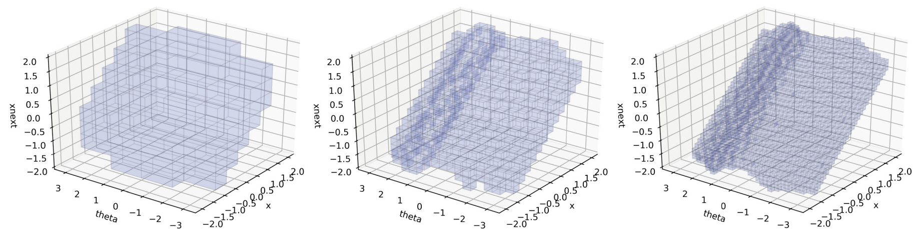


Fig. 5. Coarsening of the F_x interface to $2^3, 2^4$ and 2^5 bins along each dimension for a fixed v assignment. Interfaces are coarsened within milliseconds for BDDs but the runtime depends on the finite abstraction's data structure representation.

4.2 Dynamically Coarsening Interfaces

In practice, the sequence of interfaces Z_i generated during synthesis grows in complexity. This occurs even if the dynamics F and the target/safe sets have compact representations (i.e., fewer nodes if using BDDs). Coarsening F and Z_i combats this growth in complexity by effectively reducing the amount of information sent between iterations of the fixed point procedure.

Spatial discretization or *coarsening* is achieved by use of a quantizer interface that implicitly aggregates points in a space into a partition or cover.

Definition 7. *A quantizer $Q(i, o)$ is any interface that abstracts the identity interface $(i == o)$ associated with the signature (i, o).*

Quantizers decrease the complexity of the system representation and make synthesis more computationally tractable. A coarsening operator abstracts an interface by connecting it in series with a quantizer. Coarsening reduces the number of non-blocking inputs and increases the output non-determinism.

Definition 8 (Input/Output Coarsening). *Given an interface $F(i, o)$ and input quantizer $Q(\hat{i}, i)$, input coarsening yields an interface with signature $(\hat{i}, o)$.*

$$\mathtt{icoarsen}(F, Q(\hat{i}, i)) = \mathtt{ohide}(i, \mathtt{comp}(Q(\hat{i}, i), F)) \tag{13}$$

Similarly, given an output quantizer $Q(o, \hat{o})$, output coarsening yields an interface with signature $(i, \hat{o})$.

$$\mathtt{ocoarsen}(F, Q(o, \hat{o})) = \mathtt{ohide}(o, \mathtt{comp}(F, Q(o, \hat{o}))) \tag{14}$$

Figure 5 depicts how coarsening reduces the information required to encode a finite interface. It leverages a variable precision quantizer, whose implementation is described in the extended version at [1].

The corollary below shows that quantizers can be seamlessly integrated into the synthesis pipeline while preserving the refinement order. It readily follows from Theorems 2, 3, and the quantizer definition.

Corollary 1. *Input and output coarsening operations* (13) *and* (14) *are monotone operations with respect to the interface refinement order $\preceq$.*

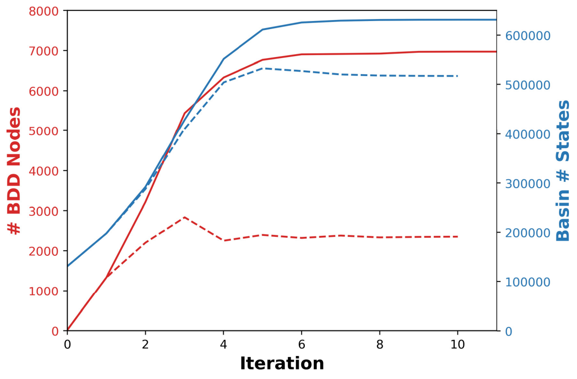


Fig. 6. Number of BDD nodes (red) and number of states in reach basin (blue) with respect to the reach game iteration with a greedy quantization. The solid lines result from the unmodified game with no coarsening heuristic. The dashed lines result from greedy coarsening whenever the winning region exceeds 3000 BDD nodes. (Color figure online)

It is difficult to know a priori where a specific problem instance lies along the spectrum between mathematical precision and computational efficiency. It is then desirable to coarsen dynamically in response to runtime conditions rather than statically beforehand. Coarsening heuristics for reach games include:

– *Downsampling with progress* [7]: Initially use coarser system dynamics to rapidly identify a coarse reach basin. Finer dynamics are used to construct a more granular set whenever the coarse iteration "stalls". In [7] only the Z_i are coarsened during synthesis. We enable the dynamics F to be as well.
– *Greedy quantization*: Selectively coarsening along certain dimensions by checking at runtime which dimension, when coarsened, would cause Z_i to shrink the least. This reward function can be leveraged in practice because coarsening is computationally cheaper than composition. For BDDs, the winning region can be coarsened until the number of nodes reduces below a desired threshold. Figure 6 shows this heuristic being applied to reduce memory usage at the expense of answer fidelity. A fixed point is not guaranteed as long as quantizers can be dynamically inserted into the synthesis pipeline, but is once quantizers are always inserted at a fixed precision.

The most common quantizer in the literature never blocks and only increases non-determinism (such quantizers are called "strict" in [18,19]). If a quantizer is interpreted as a partition or cover, this requirement means that the union must be equal to an entire space. Definition 7 relaxes that requirement so the union can be a subset instead. It also hints at other variants such as interfaces that don't increase output non-determinism but instead block for more inputs.

5 Refining System Dynamics

Shared refinement [22] is an operation that takes two interfaces and merges them into a single interface. In contrast to coarsening, it makes interfaces more precise. Many tools construct system abstractions by starting from the universal abstraction $\bot$, then iteratively refining it with a collection of smaller interfaces that represent input-output samples. This approach is especially useful if the canonical concrete system is a black box function, Simulink model, or source code file. These representations do not readily lend themselves to the predicate operations or be coarsened directly. We will describe later how other tools implement a restrictive form of refinement that introduces unnecessary dependencies.

Interfaces can be successfully merged whenever they do not contain contradictory information. The shared refinability condition below formalizes when such a contradiction does not exist.

Definition 9 (Shared Refinability [22]). *Interfaces $F_1(i,o)$ and $F_2(i,o)$ with identical signatures are shared refinable if*

$$(\texttt{nb}(F_1) \wedge \texttt{nb}(F_2)) \Rightarrow \exists o(F_1 \wedge F_2) \tag{15}$$

For any inputs that do not block for all interfaces, the corresponding output sets must have a non-empty intersection. If multiple shared refinable interfaces, then they can be combined into a single one that encapsulates all of their information.

Definition 10 (Shared Refinement Operation [22]). *The shared refinement operation combines two shared refinable interfaces F_1 and F_2, yielding a new identical signature interface corresponding to the predicate*

$$\texttt{refine}(F_1, F_2) = (\texttt{nb}(F_1) \vee \texttt{nb}(F_2)) \wedge (\texttt{nb}(F_1) \Rightarrow F_1) \wedge (\texttt{nb}(F_2) \Rightarrow F_2). \tag{16}$$

The left term expands the set of accepted inputs. The right term signifies that if an input was accepted by multiple interfaces, the output must be consistent with each of them. The shared refinement operation reduces to disjunction for sink interfaces and to conjunction for source interfaces.

Shared refinement's effect is to move up the refinement order by combining interfaces. Given a collection of shared refinable interfaces, the shared refinement operation yields the least upper bound with respect to the refinement partial order in Definition 6. Violation of (15) can be detected if the interfaces fed into $\texttt{refine}(\cdot)$ are not abstractions of the resulting interface.

5.1 Constructing Finite Interfaces Through Shared Refinement

A common method to construct finite abstractions is through simulation and overapproximation of forward reachable sets. This technique appears in tools such as PESSOA [12], SCOTS [19], MASCOT [7], ROCS [9] and ARCS [4]. By covering a sufficiently large portion of the interface input space, one can construct larger composite interfaces from smaller ones via shared refinement.

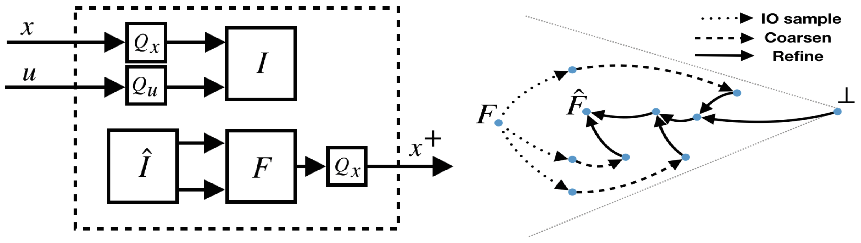


Fig. 7. (Left) Result of sample and coarsen operations for control system interface $F(x \cup u, x^+)$. The I and $\hat{I}$ interfaces encode the same predicate, but play different roles as sink and source. (Right) Visualization of finite abstraction as traversing the refinement partial order. Nodes represent interfaces and edges signify data dependencies for interface manipulation operators. Multiple refine edges point to a single node because refinement combines multiple interfaces. Input-output (IO) sample and coarsening are unary operations so the resulting nodes only have one incoming edge. The concrete interface F refines all others, and the final result is an abstraction $\hat{F}$.

Smaller interfaces are constructed by sampling regions of the input space and constructing an input-output pair. In Fig. 7's left half, a sink interface $I(x \cup u, \varnothing)$ acts as a filter. The source interface $\hat{I}(\varnothing, x \cup u)$ composed with $F(x \cup u, x^+)$ prunes any information that is outside the relevant input region. The original interface refines any sampled interface. To make samples *finite*, interface inputs and outputs are coarsened. An individual sampled abstraction is not useful for synthesis because it is restricted to a local portion of the interface input space. After sampling many finite interfaces are merged through shared refinement. The assumption $\hat{I}_i \Rightarrow \texttt{nb}(F)$ encodes that the dynamics won't raise an error when simulated and is often made implicitly. Figure 7's right half depicts the sample, coarsen, and refine operations as methods to vertically traverse the interface refinement order.

Critically, `refine`$(\cdot)$ can be called within the synthesis pipeline and does not assume that the sampled interfaces are disjoint. Figure 8 shows the results from refining the dynamics with a collection of state-control hyper-rectangles that are randomly generated via uniformly sampling their widths and offsets along each dimension. These hyper-rectangles may overlap. If the same collection of hyper-rectangles were used in MASCOT, SCOTS, ARCS, or ROCS then this would yield a much more conservative abstraction of the dynamics because their implementations are not robust to overlapping or misaligned samples. PESSOA and SCOTS circumvent this issue altogether by enforcing disjointness with an exhaustive traversal of the state-control space, at the cost of unnecessarily coupling the refinement and sampling procedures. The lunar lander in the extended version [1] embraces overlapping and uses two mis-aligned grids to construct a grid partition with p^N elements with only $p^N(\frac{1}{2})^{N-1}$ samples (where p is the number of bins along each dimension and N is the interface input dimension). This technique introduces a small degree of conservatism but its computational savings typically outweigh this cost.

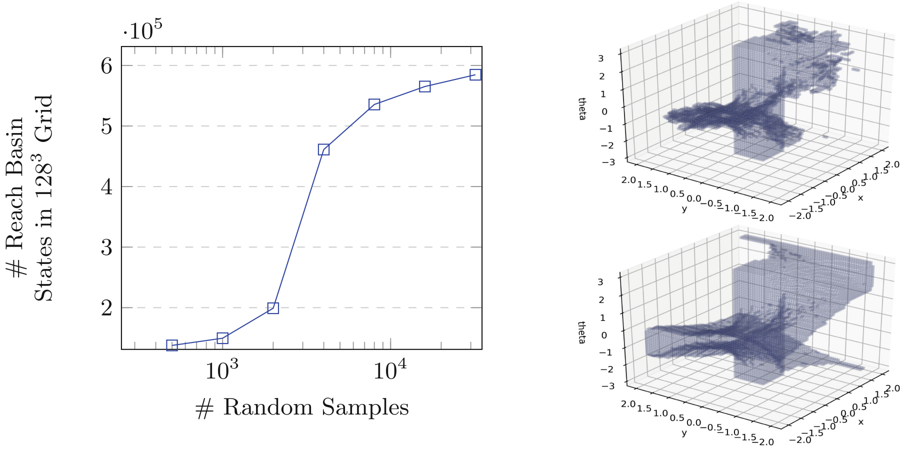


Fig. 8. The number of states in the computed reach basin grows with the number of random samples. The vertical axis is lower bounded by the number of states in the target $131k$ and upper bounded by $631k$, the number of states using an exhaustive traversal. Naive implementations of the exhaustive traversal would require 12 million samples. The right shows basins for 3000 (top) and 6000 samples (bottom).

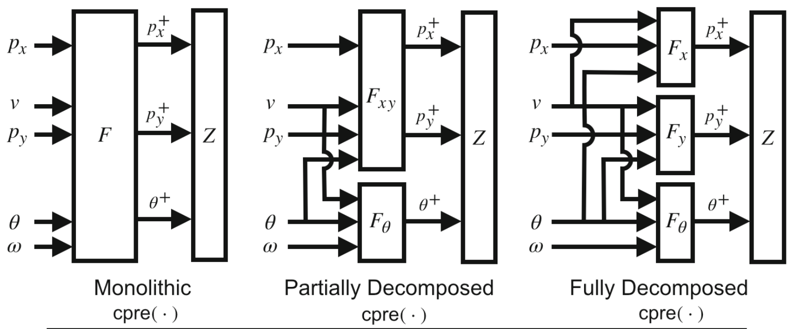
6 Decomposed Control Predecessor

A decomposed control predecessor is available whenever the system state space consists of a Cartesian product and the dynamics are decomposed componentwise such as F_x, F_y, and F_θ for the Dubins vehicle. This property is common for continuous control systems over Euclidean spaces. While one may construct F directly via the abstraction sampling approach, it is often intractable for larger dimensional systems. A more sophisticated approach abstracts the lower dimensional components F_x, F_y, and F_θ individually, computes $F = \texttt{comp}(F_x, F_y, F_\theta)$, then feeds it to the monolithic $\texttt{cpre}(\cdot)$ from Proposition 1. This section's approach is to avoid computing F at all and decompose the monolithic $\texttt{cpre}(\cdot)$. It operates by breaking apart the term $\textit{ohide}(x^+, \texttt{comp}(F, Z))$ in such a way that it respects the decomposition structure. For the Dubins vehicle example $\textit{ohide}(x^+, \texttt{comp}(F, Z))$ is replaced with

$$\textit{ohide}(p_x^+, \texttt{comp}(F_x, \textit{ohide}(p_y^+, \texttt{comp}(F_y, \textit{ohide}(\theta^+, \texttt{comp}(F_\theta, Z))))))$$

yielding a sink interface with inputs p_x, p_y, v, θ, and ω. This representation and the original $\textit{ohide}(x^+, \texttt{comp}(F, Z))$ are equivalent because $\texttt{comp}(\cdot)$ is associative and interfaces do not share outputs $x^+ \equiv \{p_x^+, p_y^+, \theta^+\}$. Figure 9 shows multiple variants of $\texttt{cpre}(\cdot)$ and improved runtimes when one avoids preemptively constructing the monolithic interface. The decomposed $\texttt{cpre}(\cdot)$ resembles techniques to exploit partitioned transition relations in symbolic model checking [5].

No tools from Sect. 1.1 natively support decomposed control predecessors. We've shown a decomposed abstraction for components composed in parallel



Decomposition	**Parallel Compose Runtime (s)**	**Reach Game Runtime (s)**
F (Monolithic)	0.56	103.09
$F_{y\theta}, F_x$ (Partially Decomp.)	0.02	28.31
$F_{x\theta}, F_y$ (Partially Decomp.)	0.01	28.71
F_{xy}, F_θ (Partially Decomp.)	0.06	10.61
F_x, F_y, F_θ (Fully Decomp.)	n/a	4.42

Fig. 9. A monolithic `cpre`$(\cdot)$ incurs unnecessary pre-processing and synthesis runtime costs for the Dubins vehicle reach game. Each variant of `cpre`$(\cdot)$ above composes the interfaces F_x, F_y and F_θ in different permutations. For example, F_{xy} represents `comp`(F_x, F_y) and F represents `comp`(F_x, F_y, F_θ).

but this can also be generalized to series composition to capture, for example, a system where multiple components have different temporal sampling periods.

7 Conclusion

Tackling difficult control synthesis problems will require exploiting *all* available structure in a system with tools that can *flexibly adapt* to an individual problem's idiosyncrasies. This paper lays a foundation for developing an extensible suite of interoperable techniques and demonstrates the potential computational gains in an application to controller synthesis with finite abstractions. Adhering to a simple yet powerful set of well-understood primitives also constitutes a disciplined methodology for algorithm development, which is especially necessary if one wants to develop concurrent or distributed algorithms for synthesis.

References

1. http://arxiv.org/abs/1905.09503
2. https://github.com/ericskim/redax/tree/CAV19
3. Basar, T., Olsder, G.J.: Dynamic Noncooperative Game Theory, vol. 23. Siam, Philadelphia (1999)

4. Bulancea, O.L., Nilsson, P., Ozay, N.: Nonuniform abstractions, refinement and controller synthesis with novel BDD encodings. CoRR, arXiv: abs/1804.04280 (2018)
5. Burch, J., Clarke, E., Long, D.: Symbolic model checking with partitioned transition relations (1991)
6. Gruber, F., Kim, E., Arcak, M.: Sparsity-aware finite abstraction. In: 2017 IEEE 56th Conference on Decision and Control (CDC), December 2017
7. Hsu, K., Majumdar, R., Mallik, K., Schmuck, A.-K.: Multi-layered abstraction-based controller synthesis for continuous-time systems. In: Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (Part of CPS Week), HSCC 2018, pp. 120–129. ACM, New York (2018)
8. Khalil, H.K., Grizzle, J.W.: Nonlinear Systems, vol. 3. Prentice Hall, Upper Saddle River, New Jersey (2002)
9. Li, Y., Liu, J.: ROCS: a robustly complete control synthesis tool for nonlinear dynamical systems. In: Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (Part of CPS Week), HSCC 2018, pp. 130–135. ACM, New York (2018)
10. Liu, J.: Robust abstractions for control synthesis: completeness via robustness for linear-time properties. In: Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control, HSCC 2017, pp. 101–110. ACM, New York (2017)
11. Majumdar, R.: Symbolic algorithms for verification and control. Ph.D. thesis, University of California, Berkeley (2003)
12. Mazo Jr., M., Davitian, A., Tabuada, P.: PESSOA: a tool for embedded controller synthesis. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 566–569. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_49
13. Meyer, P.J., Girard, A., Witrant, E.: Compositional abstraction and safety synthesis using overlapping symbolic models. IEEE Trans. Autom. Control. **63**, 1835–1841 (2017)
14. Mouelhi, S., Girard, A., Gössler, G.: CoSyMA: a tool for controller synthesis using multi-scale abstractions. In: 16th International Conference on Hybrid Systems: Computation and Control, pp. 83–88. ACM (2013)
15. Nilsson, P., Ozay, N., Liu, J.: Augmented finite transition systems as abstractions for control synthesis. Discret. Event Dyn. Syst. **27**(2), 301–340 (2017)
16. Nuzzo, P.: Compositional design of cyber-physical systems using contracts. Ph.D. thesis, EECS Department, University of California, Berkeley, August 2015
17. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 364–380. Springer, Heidelberg (2005). https://doi.org/10.1007/11609773_24
18. Reißig, G., Weber, A., Rungger, M.: Feedback refinement relations for the synthesis of symbolic controllers. IEEE Trans. Autom. Control. **62**(4), 1781–1796 (2017)
19. Rungger, M., Zamani, M.: SCOTS: a tool for the synthesis of symbolic controllers. In: 19th International Conference on Hybrid Systems: Computation and Control, pp. 99–104. ACM (2016)
20. Somenzi, F.: CUDD: CU Decision Diagram Package. http://vlsi.colorado.edu/~fabio/CUDD/, Version 3.0.0 (2015)
21. Tabuada, P.: Verification and Control of Hybrid Systems. Springer, New York (2009). https://doi.org/10.1007/978-1-4419-0224-5

22. Tripakis, S., Lickly, B., Henzinger, T.A., Lee, E.A.: A theory of synchronous relational interfaces. ACM Trans. Program. Lang. Syst. (TOPLAS) **33**(4), 14 (2011)
23. Zamani, M., Pola, G., Mazo, M., Tabuada, P.: Symbolic models for nonlinear control systems without stability assumptions. IEEE Trans. Autom. Control **57**(7), 1804–1809 (2012)