Submitted, Proceedings of the IEEE (Jul 2017)
http://www.cds.caltech.edu/~murray/preprints/fm17-pieee_s.pdf

Layering assume-guarantee contracts for hierarchical system design

Ioannis Filippidis and Richard M. Murray

***Abstract*—Specifications for complex engineering systems are typically decomposed into specifications for individual subsystems in a manner that ensures they are implementable and simpler to develop further. We describe a method to algorithmically construct component specifications that implement a given specification when assembled. By eliminating variables that are irrelevant to realizability of each component, we simplify the specifications and reduce the amount of information necessary for operation. To identify which variables can be hidden while preserving realizability, we parametrize the information flow between components, and eliminate the selected variables from component specifications. The resulting specifications describe component viewpoints with full information with respect to the remaining variables, which is essential for tractable algorithmic synthesis of implementations. The specifications are written in TLA$^+$, with liveness properties restricted to an implication of conjoined recurrence properties, known as GR(1). We define an operator for forming open-systems from closed-systems, based on a variant of the "while-plus" operator. The resulting open-system properties are realizable when expected to be, without need for separate side conditions. To convert the generated specifications from binary decision diagrams to readable formulas over integer variables, we symbolically solve a minimal covering problem. We show with examples how the method can be applied to obtain contracts that formalize the hierarchical structure of system design.**

I. Introduction

A. Motivation

The design and construction of a large system relies on the ability to divide the problem into smaller ones that involve parts of the system. Each subproblem may itself be refined further into smaller problems, as illustrated in Fig. 1. Typically the subsystems interact with each other, either physically, via software, or both. This interaction between modules needs to be constrained, in order to ensure that the assembled system behaves as intended. For example, if we consider a component that controls the manipulator arm of a rover for exploring the geology of other planets, it depends on a camera for deciding how to position the arm, and on a power supply, as sketched in Fig. 2. The maximum manipulator speed that the controller can safely command is limited by the camera frame rate. Depending on power supply and type of operation, the manipulator controller can request a lower frame rate from the camera, in order to economize on power. During grasping operations though, the controller does require high fidelity and frequent frames. Based on the available power supply, the controller may decide to decline a request for grasping, due to insufficient power for completing the operation. Such an issue could arise in rovers that depend on solar energy, because their power supply is contigent on environmental conditions.

The authors are with Control and Dynamical Systems, California Institute of Technology, Pasadena, CA, 91125, USA. E-mail: {ifilippi, murray}@caltech.edu

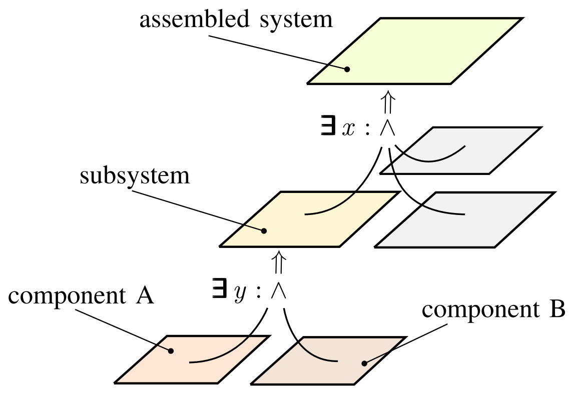


Fig. 1: Anatomy of hierarchical system design in TLA$^+$: composition is represented by conjunction ($\wedge$), hiding of details by (temporal) existential quantification ($\exists$), and refinement by logical implication $\Rightarrow$.

Systems are built from designs, and designs are created incrementally. A common direction is to start thinking in terms of larger pieces, and divide those in smaller ones that are more detailed, but also more specific and local in nature [1]. The design activity should be captured accurately [2], [3]. A representation with precise syntax and semantics, or *specification*, is desirable to describe how each component should behave in the context of other modules. When a specification is available, we can attempt to prove that a system is safe to operate and useful. Unsafe designs can have a high cost, for example in the context of airliners, automotive subsystems, nuclear power plant controllers, and several other application areas.

This decomposition involves distributing functionality among components, and creating interfaces between them [4], [5]. We are motivated to decompose in order to *focus* and *isolate*. Focus of attention allows for fewer errors. Isolation makes reasoning easier, and more tractable to automate [6]. Decomposition also makes possible the use of off-the-shelf components, and the assignment of tasks to different subsystem manufacturers. Obtaining conclusions about a system by using facts about its subsystems comes at an extra cost [7], [8]. But it may be the only scalable way of approaching the design of a large system [9], [7, pp. 421–422], [10, p.168].

A system can be described at different levels of detail [11], [12, p.192]. A description that corresponds closely to available physical elements is directly implementable [1]. However, writing specifications at this level of detail is often more difficult than specifying behavior at a higher level. A specification at the implementation level can then be derived by hand or using (automated) *synthesis*, which relies on notions from the theory of games [13]. Synthesis has attracted considerable interest in the past two decades, and advances both in theory and implementation have been made, as described in Sec. II-B

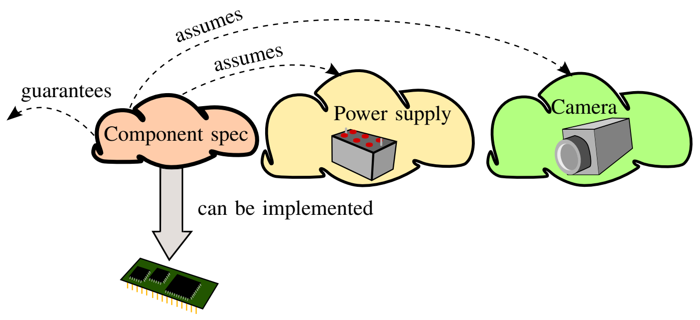


Fig. 2: A component is specified by expressing requirements that the implementation should fulfill under assumptions about other parts of the assembled system.

[14], [15], [16], [17], [18], [19], [20]. In this work, we are interested in automated decomposition of specifications that yields *implementable* component specifications. In particular, we aim at automatically modularizing a design that has been partially specified by a human. Human input is necessary, in one form or another, because an algorithm cannot know what the assembled system is intended for, and what part of the system each component represents. Algorithmic synthesis can be used to implement the specifications that result after some iterations of decomposition.

B. *Proposed approach*

A component mathematically means a collection of variables. A contract is a collection of realizable component specifications that combined imply the specification we decomposed. It is easier to write a centralized specification, referring to any variable as needed. But it is simpler to specify internal details in absence of variables from other components. In this paper, we study the problem of eliminating variables during the decomposition of specifications into assume-guarantee contracts between components. In order to detect which variables each component needs to know about, we use parameters that prescribe whether each variable is hidden or not. This can be regarded as parametrizing the information communicated between components.

We prove that variables selected to be hidden can be eliminated from the resulting specifications. Thus, information is hidden without producing component specifications that would be computationally expensive or intractable to implement [21], [22], [23]. Both the property to be decomposed and the generated component properties are expressed with liveness restricted to an implication between conjunctions of recurrence properties, a fragment called GR(1) [15]. The polynomial computational complexity of implementing open-system GR(1) specifications motivates this choice, discussed more in Sec. III-C. The fulfillment of liveness requirements between components is acyclic, in order to avoid circular dependencies.

The second problem that we study is writing the constructed specifications in a form that humans can read, so that they can work with the produced specifications at a lower level of refinement, for example to specify details internal to a subsystem before decomposing the subsystem into components. The algorithms that we develop are symbolic, in that they manipulate binary decision diagrams (BDDs), which are graph-based data structures that represent sets of states [24], [25], [26]. The predicates of the assume-guarantee component specifications are computed as BDDs first, which are not suitable for reading. Assuming that shorter formulas are more readable, we formulate as a minimal set covering problem the construction of minimal formulas in disjunctive normal form of interval constraints over integer variables. The covering problem is solved exactly with a symbolic branch and bound algorithm originally proposed for two-level logic minimization [27].

The paper is organized as follows. A short literature review is given in Sec. II. Sec. III introduces the mathematical language we use, a formalized notion of implementability, and some elements from algorithmic game theory. Assume-guarantee contracts are defined in Sec. IV-A and open systems in Sec. IV-B. The parameterization of which variables are hidden when solving a game is developed in Sec. V. The decomposition algorithm is described in Sec. VI. How minimal specifications are generated is described in Sec. VII. An example is analyzed in Sec. VIII, and conclusions summarized in Sec. IX.

II. Previous Work

A. *Modular design by contract*

The dependence of a component on its outside world is known as assumption-commitment, or rely-guarantee, paradigm for describing behaviors [3]. The assumption-commitment paradigm about reactive systems is an evolved instance of reasoning about conditions before and after a terminating behavior. Early formulations [28, pp. 26–29], [29, p.4] were the assertion boxes used by Goldstine and von Neumann [30], and the tabulated assertions used by Turing [31], [32]. A formalism for reasoning using triples of a *precondition*, a program, and a *postcondition* was introduced by Hoare [33], following the work of Floyd [34], [29, pp. 3–4] on proving properties of elements in a flowchart, based on ideas by Perlis and Gorn [35, p.122], [28, p.32 and Ref.25, p.44].

Hoare's logic applies to terminating programs. However, many systems are not intended to terminate, but instead continue to operate, by reacting to their environment [36]. Francez and Pnueli [37] introduced a first generalization of Hoare-style reasoning to cyclic programs. They also considered concurrent programs. Their formalism uses explicit mention of time and is structured into pairs of assumptions and commitments.

Lamport [12] observed that such a style of specification is essential to reason about complex systems in a modular way [38, p.131]. Lamport and Schneider [39], [40] introduced, and related to previous approaches, what they called *generalized Hoare logic*. This is a formalism for reasoning with pre- and post-conditions, in order to prove program invariants. Misra and Chandy introduced the rely-guarantee approach for safety properties of distributed systems [41]. All properties up to this point were safety, and not expressed in temporal logic [42]. Two developments followed, and the work presented here is based on them.

The first was Lamport's introduction of *proof lattices* [43]. A proof lattice is a finite rooted directed acyclic graph, labeled with assertions. If u is a node labeled with property U, and v, w are its successors, labeled with properties V, W, then if U holds at any time, eventually either V or W will hold. In temporal logic, this can be expressed as $\Box(U \Rightarrow \Diamond(V \vee W))$. Owicki and Lamport [44] revised the proof lattice approach, by labeling nodes with *temporal* properties, instead of atemporal ones ("immediate assertions").

The second development was the expression by Pnueli [38] of assume-guarantee pairs in temporal logic, i.e., without reference to an explicit *time* variable. In addition, Pnueli proposed a proof method for *liveness* properties, which is based on well-founded induction. This method can be understood as starting with some temporal premises for each component, and iteratively tightening these properties into consequents that are added to the collection of available premises, for the purpose of deriving further consequents. This method enables proving liveness properties of modular systems. Informally, the requirement of well-foundedness allows using as premises only properties from an earlier stage of the deductive process [45], [46]. This prevents circular existential reasoning about the future, i.e., circular dependencies of liveness properties [47, §2.2, p.512], [48, §5.4, p.264], [49]. As a simple example, consider Alice and Bob. Alice promises that, if she sees b, then she will do a at *some* time in the future. Reciprocally, Bob promises to eventually do b, after he sees a. As raw TLA formulas, these read $\Box(b \Rightarrow \Diamond a')$ for Alice, and $\Box(a \Rightarrow \Diamond b')$ for Bob. If both Alice and Bob default to not doing any of a or b, then they both satisfy their specifications. This problem arises because existential quantification over time allows simultaneous antecedent failure. Otherwise, if Bob was *required* to do b for the first time, then Alice would have to do a, then Bob do b again, etc.

Compositional approaches to verification have treated the issue of circularity by using the description of the implementation under verification as a vehicle for carrying out the proof. The implementation's immediate behavior should constrain the system sufficiently much so as to enable deducing its liveness guarantees. This approach is suitable for verification, because an implementation is available at that stage. Specifications intended to be used for synthesis are more permissive. For this reason liveness properties, and minimal reliance on step-by-step details, are preferred in the context of synthesis. Stark [49] proposed a proof rule for assume-guarantee reasoning about a non-circular collection of liveness properties. McMillan [50] introduced a proof rule for circular reasoning about liveness. However, this proof system is intended for verification, so it still relies on the availability of a model. It requires the definition of a proof lattice, and introduces graph edges that *consume* time, as a means to break simultaneity cycles. The method we propose in this work constructs specifications that can have dependencies of liveness goals, but in a way that avoids circularity (Sec. VI).

The assumption-guarantee paradigm has since evolved, and is known by several names. Lamport remarks that a module's specification may be viewed as a contract between user and implementer [12, p.191]. Meyer [51] called the paradigm *design by contract* and supported its use for abstracting software libraries and validating the correct operation of software. The notion of a contract has several forms. For example, an *interface automaton* [52] describes assumptions implicitly, as those environments that can be successfully connected to the interface. An interface automaton abstracts the internal details of a module and serves as its "surface appearance" towards other modules.

More recently, contracts have been proposed for specifying the design of systems with both physical and computational aspects [53]. In this context, contracts are used broadly, as an umbrella term that encompasses both interface theories and assume-guarantee contracts [54], [55], [53], with extensions to timed and probabilistic specifications. A proof system has been developed for verifying that a set of contracts refines a contract for the composite system [56], as well as a verification tool of contract refinement using an SMT solver [57]. This body of work focuses mainly on using and modifying existing contracts. We are interested in *constructing* contracts.

B. Synthesis of implementations

This section samples the literature on games of infinite duration. Synthesizing an implementation from a specification can be formulated as a game between component and environment. The type of game depends on:

- whether one or more components are being designed,
- whether components are designed in groups,
- when components change their state,
- the liveness part of specifications, and
- the visibility of variables.

Games can be turn-based or concurrent [58], [59], [60]. Inability to observe external state changes makes a game asynchronous [22], [61]. If we want to construct a single component, then the synthesis problem is *centralized*. Synchronous centralized synthesis from LTL has time complexity doubly exponential in the length of the formula [36], and polynomial in the number of states. By restricting to a less expressive fragment of LTL, the complexity can be lowered to polynomial in the formula [15]. Asynchronous centralized synthesis does not yield to such a reduction [22]. Partial information games pose a challenge similar to full LTL properties, due to the need for a powerset-like construction [62]. To avoid this route alternative methods have been developed, using universal co-Büchi automata [19], or antichains [20].

If we want to construct several communicating modules to obtain some collective behavior, then synthesis is called *distributed*. Of major importance in distributed synthesis is who talks to whom, and how much, called the communication architecture. A distributed game with full information is in essence a centralized synthesis problem. Distributed *synchronous* games with partial information are undecidable [23], unless we restrict the communication architecture to avoid information forks [63], or restrict the specifications to limited fragments of LTL [64]. The undecidability of distributed synthesis motivates our parametrization for finding a suitable connectivity architecture, instead of deciding whether a given architecture suffices. *Bounded synthesis* circumnavigates this intractability by searching for systems with a priori bounded memory [65]. Asynchronous distributed synthesis is undecidable [61]. Besides synthesis of a distributed implementation, the more general notion of *assume-guarantee* synthesis [66] constructs modules that can interface with a set of other modules, as described by an assumption property. This is the same viewpoint with the approach proposed here. A difference is that we are interested in synthesizing temporal properties with a liveness part, instead of implementations. In addition, we are interested in "distributed" also in the sense that the modules will be synthesized separately.

Another body of relevant work is the construction of assumptions that make an unrealizable problem realizable. The methods originally developed for this purpose have been targeted at compositional verification, and use the $L^\star$ algorithm for learning deterministic automata [67], and implemented symbolically [68]. Later work addressed synthesis by separating the construction of assumptions into safety and liveness [69]. The safety assumption is obtained by property *closure*, which also plays a key role in the composition theorem presented in [70]. Our work is based on this separate treatment of safety and liveness. Methods that use opponent strategies [71] to refine the assumptions of a GR(1) specification, searching over syntactic patterns were proposed in [72], [73]. The syntactic approach of [73] was used in [74] to refine assume-guarantee specifications of coupled modules. However, that work cannot handle circularly connected modules, thus neither circular liveness dependencies. Another approach is cooperative reactive synthesis, where a logic with non-classical semantics is used, and synthesis corresponds to this semantics [75].

Our work uses parametrization, based on ideas of approximating asynchronous with GR(1) synthesis [22], [17]. Another form of parametrization studied in the context of synthesis is that of safety and reachability goals [76]. Instead of hiding specific variables, an alternative approach in the context of verification [77] identifies predicates that capture essential information for carrying out proofs with less coupling between processes. Also relevant is the separation of GR(1) synthesis into the design of a memoryless observer (estimating based on current state only) and of a controller with full information [21]. Identifying what variables provide information essential for realizability (Sec. V) relates to work on synthesizing probabilistic sensing strategies [78]. A hierarchical approach where an observer for the continuous state is designed separately from synthesizing a discrete controller from temporal logic specifications [79], and decomposition of properties for synthesizing implementations have been studied in the context of aircraft management systems [80]. Layering as a method for structuring system design has been applied in the context of the DisCo method, which is based on TLA [81], [82], [5].

The Quine-McCluskey minimization method, which takes exponential amount of space and time and so is impractical, has been used before for simplifying Boolean logic expressions in manuals [83], robot path planning among planar rectangles [84] and recently for simplifying enumerated robot controllers [85]. In the context of synthesis, prime implicants (used here for minimal covering) have been used for refining abstractions [18], and have been mentioned in the context of debugging specifications [86]. For theories more general than propositional logic there has been work on deriving prime implicants in the context of SMT solvers [87].

III. Preliminaries

A. Predicate Logic and Set Theory

We use the temporal logic of actions (TLA$^+$) [10], with some minor modifications that accommodate for a smoother connection to the literature on games. At places, we also use "raw" TLA$^+$, which is a fragment that allows stutter-sensitive temporal properties (stutter invariance is defined below) [88, §4], [89, p.34]. The motivation for choosing TLA$^+$ is its precise syntax and semantics, the use of stuttering steps and hiding as a refinement mechanism, and the structuring of specifications, by using modules, and within modules by definitions and planar arrangement of formulas.

TLA$^+$ is based on Zermelo-Fraenkel (ZF) set theory [10, p.300], which is regarded as a foundation for mathematics [90]. Every entity in TLA$^+$ is a set (also called a *value*). A function f is a set with the property that, for every $x \in \textsc{domain}\, f$, we know what value $f[x]$ is. Functions can be defined with the syntax

$$f \;\triangleq\; [x \in S \mapsto e(x)],$$

where $e(x)$ some expression [10, p.303, p.71]. If a value f equals the function constructor that maps values in $\textsc{domain}\, f$ to the values obtained by function application, then it is a function

$$\mathit{IsAFunction}(f) \;\triangleq\; f = [x \in \textsc{domain}\, f \mapsto f[x]],$$

For any $x \notin \textsc{domain}\, f$, $f[x]$ is some value, unspecified by the axioms of TLA$^+$. The collection of functions with domain S and range $R \subseteq T$ forms a set, denoted by $[S \to T]$.

Operators are defined to equal some expression, with no domain specified. Unlike functions, which are sets, operators are a syntactic mechanism for naming. All occurrences of operators are syntactically replaced by their definitions before semantic interpretation takes place. Parentheses instead of brackets distinguish an operator from a function, for example $g(x) \triangleq x$ defines the operator g to be the identity mapping. Unnamed operators are built with the construct $\textsc{lambda}$ [91]. A first-order operator takes as arguments operators without arguments (nullary). An operator that takes a first-order operator as argument is called *second-order*. For example, the expression $F(x, G(\_))$ denotes an operator F that takes as arguments a nullary operator x and a unary operator G [10, §17.1.1]. TLA$^+$ includes [10, §16.1.2] Hilbert's choice operator [92], [93]. If $\exists x : P(x)$, then the expression $\textsc{choose}\, x : P(x)$ equals some value that satisfies $P(x)$. Otherwise, this expression is some unspecified value that can differ depending on P.

The operator $\wedge$ denotes conjunction, $\vee$ disjunction, $\neg$ negation. *Nat* denotes the set of natural numbers [10, §18.6, p.348],

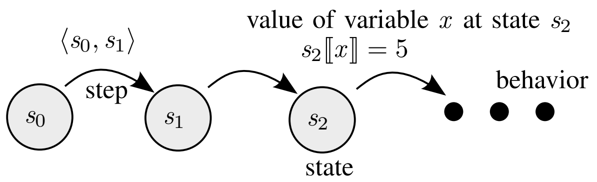

value of variable x at state s_2
$s_2[\![x]\!] = 5$
$\langle s_0, s_1 \rangle$
step
behavior
s_0
s_1
s_2
state

Fig. 3: Semantic concepts of the temporal logic TLA+.

and for $i, j \in Nat$ the set of integers between i and j is denoted by

$$i..j \;\triangleq\; \{n \in Nat : i \leq n \land n \leq j\}.$$

A function with domain $1..n$ for some $n \in Nat$ is called a *tuple* and denoted with angle brackets, for example $\langle a, b \rangle$.

There are two kinds of variables: *rigid* and *flexible*. Rigid variables are also called *constants*. They are unchanged through steps of a behavior (behaviors are defined below). Rigid quantification can be bounded, as in the formula $\forall x \in S : P(x)$, or unbounded, as in $\forall x : P(x)$. The former is defined in terms of the latter as

$$\forall x \in S : P(x) \;\triangleq\; \forall x : (x \in S) \Rightarrow P(x).$$

So the "bound" is an antecedent. Substitution of the expression e_1 for occurrences of the identifier x in the expression e is written as the formula $\text{LET } x \triangleq e_1 \text{ IN } \quad e$.

B. Semantics of Modal Logic

Temporal logic serves for reasoning about dynamics, because it is interpreted over sequences of states (for linear semantics). A *state* s is an assignment of values to *all* variables. A *step* is a pair of states $\langle s_1, s_2 \rangle$, and a *behavior* σ is an infinite sequence of states, i.e., a function from Nat to states. An *action* (*state predicate*) is a Boolean-valued formula over steps (states). Given a step $\langle s_1, s_2 \rangle$, the formulas x and x' denote the values $s_1[\![x]\!]$ and $s_2[\![x]\!]$, respectively. We will use the temporal operators: $\Box$ "always" and $\Diamond$ "eventually". If formula f is TRUE in every (some) state of behavior σ, then $\sigma \models \Box f$ ($\sigma \models \Diamond f$) is TRUE. Formal semantics are defined in [10, §16.2.4].

A property is a collection of behaviors described by a temporal formula. If a property φ cannot distinguish between two behaviors that differ only by repetition of states, then φ is called *stutter-invariant*. Stutter-invariance is useful for refining systems by adding lower-level details [11]. In TLA+ stutter-invariance is ensured by the constructs [48, Prop.2.1]

$$[A]_v \;\triangleq\; A \lor (v' = v),$$
$$\langle A \rangle_v \;\triangleq\; \neg[A]_v = A \land (v' \neq v).$$

So, $\Box[x = x + 1]_x$ is satisfied by a behavior whose each step either increments x by one, or leaves x unchanged.

If every behavior σ that violates property φ has a finite prefix that cannot be extended to satisfy φ, then φ is a *safety* property. If any finite behavior can be extended to satisfy φ, then φ is a liveness property [94] [35, p.49]. A property of the form $\Box\Diamond p$ ($\Diamond\Box p$) is called *recurrence* (*persistence*).

We briefly mention a few more concepts that we will use later. An informal definition is sufficient to follow the discussion, and a formal one can be found in the semantics of nonconstant operators [10, Ch.16.2]. The thick existential quantifier $\boldsymbol{\exists}$ denotes *temporal* existential quantification over (flexible) variables. The main purpose of temporal quantification is to "hide" variables that represent details internal to a subsystem. The expression ENABLED A is true at states from where some step could be taken that satisfies the action A. Using enabledness, weak fairness is defined as

$$\text{WF}_v(A) \;\triangleq\; (\Diamond\Box\text{ENABLED } \langle A \rangle_v) \Rightarrow \Box\Diamond\langle A \rangle_v$$

Collection versus set: Not every statement in ZF defines a set. Some statements describe collections that are too large to be sets [10, p.66]. In naive set theory this phenomenon gives rise to Rusell's and other paradoxes [90]. A collection that is not a set is called a proper class [95, p.20]. The semantics of TLA+ involve states that assign values to all variable names. Any finite formula we write will omit some variable names. For each state that satisfies the formula, we can assign arbitrary values to variables that do not occur in the formula, and thus obtain another state that satisfies the same formula. Thus, the collection of states that satisfy a formula is not a set [96, p.65] (within the theory that the semantics is discussed). So to accommodate for TLA+ semantics we should use the term "collection" instead of "set". However, to use common terminology and for brevity, we will refer to "sets" of states, even when "collection" would be appropriate.

C. Synthesis of implementations

Synthesis is the algorithmic construction of an implementation that satifies the specification of a component. Let y be a variable that represents the component, and x a variable representing the component's environment. Consider a property described by the temporal formula $Phi(x, y)$. If an implementation of $Phi(x, y)$ exists, then the property is called *realizable*. The notion of realizability [97], [98], [36] can be formalized [96] as follows

$$\begin{array}{l}
Realization(x,\, y,\, mem,\, f,\, g,\, y0,\, mem0,\, e) \;\triangleq \\
\quad \text{LET } v \;\triangleq\; \langle mem,\, x,\, y\rangle \\
\qquad\quad A \;\triangleq\; \land\, y' = f[v] \\
\qquad\qquad\quad\;\; \land\, mem' = g[v] \\
\quad \text{IN} \quad \land\, \langle mem,\, y\rangle = \langle mem0,\, y0\rangle \\
\qquad\quad\;\; \land\, \Box[e \Rightarrow A]_v \\
\qquad\quad\;\; \land\, \text{WF}_{\langle mem,\, y\rangle}(e \land A)
\end{array}$$

The variable mem serves as internal memory, with initial value $mem0$. The functions f and g control the component state y and internal memory mem, and $y0$ is the initial value of y. The action e determines when the component is allowed to change its state. Behaviors that satisfy $Realization(\dots)$ are those that could result when the component is implemented with the functions f (for externally visible behavior) and g (for internal behavior). Let

$$\begin{array}{l}
IsAFiniteFcn(f) \;\triangleq\; \land\, IsAFunction(f) \\
\qquad\qquad\qquad\qquad\quad \land\, IsFiniteSet(\text{DOMAIN } f) \\
IsRealizable(Phi(\_,\, \_),\, e(\_,\, \_)) \;\triangleq \\
\quad \exists\, f,\, g,\, y0,\, mem0 : \\
\qquad \land\, IsAFiniteFcn(f) \land IsAFiniteFcn(g) \\
\qquad \land\, \text{LET } \;\; R(mem,\, u,\, v) \;\triangleq\; Realization(
\end{array}$$

```
            $u,\ v,\ mem,\ f,\ g,\ y0,\ mem0,\ e(u,\ v))$
        IN  $\forall x, y : (\exists mem : R(mem, x, y)) \Rightarrow Phi(x, y)$
```

where the operator *IsFiniteSet* requires finite cardinality [10, p.341]. The expression $IsRealizable(Phi, e)$ means that property *Phi* can be implemented with resources e. The above definition is based on a note by Lamport [8], (see also [12, p.221]), and formalizes realizability as described in the literature on synthesis [99, §4, pp. 46–47], [100, §2.3, pp. 914–915], with a difference regarding initial conditions.

Tractable liveness: A formula described by the schema

$$StreettPair \triangleq \bigvee_{j \in 1..m} \Diamond\Box P_j \lor \bigwedge_{i \in 1..n} \Box\Diamond R_i$$

defines a liveness property categorized as generalized Streett(1), or GR(1) [15]. A formula of this form is useful for expressing the dependence of a component on its environment. Rewriting the above as

$$(\bigwedge_{j \in 1..m} \Box\Diamond\neg P_j) \Rightarrow \bigwedge_{i \in 1..n} \Box\Diamond R_i$$

emphasizes this use case. Usually, the formulas $\neg P_j$ express recurrence properties that the component requires from its environment in order to be able to realize the properties R_i. If the environment lets the behavior satisfy $\Diamond\Box P_j$, then the component cannot and so is not required to satisfy the properties $\Box\Diamond R_i$. An example that simplifies the landing gear example of Sec. VIII is $\Box\Diamond(DoorsOpen) \Rightarrow \Box\Diamond(ExtensionRequest \Rightarrow GearExtended)$. As a specification for the gear subsystem of an aircraft, this property requires that the gear respond to any request to extend under suitable conditions.

Conjoining k Streett pairs yields a liveness property called GR(k), which can be regarded as a modal conjunctive normal form [22]. Synthesis of a controller that implements a GR(k) property has computational complexity factorial in the number of Streett pairs k [16]. This is why GR(1) properties are preferred to write specifications for synthesis. An implementation that satisfies a GR(1) property can be computed by applying the controllable step operator (defined in Sec. III-D) $m \times n \times S^3$ times, where S the number of states (usually exponential in the number of variables) [100], [101], [13]. When a symbolic implementation is used the runtime is in practice much smaller than the upper bound S^3, because the state space is much "shallower" than the number of states.

A controller that implements a generalized Streett property can require additional state (memory) as large as $1..n$. There are properties that admit memoryless controllers, but searching for them is NP-complete in the number of states [102], so exponentially more expensive than GR(1) synthesis [15]. For this reason GR(1) synthesis algorithms unconditionally add a memory variable that ranges over $1..n$.

D. Elements of synthesis algorithms

Reasoning about open systems involves computing from which states a controller exists that can guide the system to a desired set of states. Given a set of states as destination, an *attractor* is the set of states from where such a controller exists. Computing an attractor can be viewed as solving a "multi-step" control problem that iteratively solves a "one-step" control problem [59]. The "one-step" control problem is described by the controllable step operator *Step* (commonly denoted as *CPre*), which is defined as follows

```
    $Step(x,\ y,\ Target(\_,\ \_)) \triangleq \exists y' :$
        $\land\ SysNext(x,\ y,\ y')$
        $\land\ \forall x' :\ EnvNext(x,\ y,\ x') \Rightarrow Target(x',\ y')$
```

A state satisfies *Step* if x, y take values in that state such that the system can choose a next value y' allowed by *SysNext*, and any next environment value x' that *EnvNext* allows leads to a state that satisfies *Target*. The above definition of *Step* is for specifications where in each step at most one component can change its state in multiple ways. For more general specifications, the corresponding *Step* operator is

```
$GeneralStep(x,\ y,\ Target(\_,\ \_)) \triangleq \exists y' :$
    $\land\ SysNext(x,\ y,\ y')$
    $\land\ \forall x' :$  $\lor\ \neg EnvNext(x,\ y,\ x')$
                   $\lor\ \land\ AssemblyNext(x,\ y,\ x',\ y')$
                     $\land\ Target(x',\ y')$
```

The general case is mentioned for completeness. We will use the first version, which applies to the interleaving specifications that we consider.

Remark 1: The expression $\exists y'$ is ungrammatical in TLA$^+$ [10, p.281, p.110]. Instead we should write fresh rigid variables, for example $\exists v$. Having said this, we continue with $\exists y'$ below because it makes reading easier. $\Box$

Assume that the state predicate $Goal(\_,\_)$ describes the destination. The states from where a controller exists that can guide the system in at most k steps to some state that satisfies the predicate $Goal(x, y)$ are those that satisfy the following operator [13], [101]

```
$kStepAttractor(x,\ y,\ Goal(\_,\ \_),\ k) \triangleq$
  LET RECURSIVE $F(\_,\ \_,\ \_)$
      $F(u,\ v,\ m) \triangleq$ IF $m = 0$
          THEN $Goal(u,\ v)$
          ELSE LET $Target(a,\ b) \triangleq F(a,\ b,\ m-1)$
                IN  $\lor\ Target(u,\ v)$
                    $\lor\ Step(u,\ v,\ Target)$
  IN  $F(x,\ y,\ k)$
```

Recursive definitions as the above are part of TLA^{+2} [91]. The attractor of *Goal* is the fixpoint of the *kStepAttractor*

```
$Attractor(x,\ y,\ Goal(\_,\ \_)) \triangleq$
    LET
        $Attr(u,\ v,\ n) \triangleq kStepAttractor(u,\ v,\ Goal,\ n)$
        $r \triangleq$ CHOOSE $k \in Nat :\ \forall u,\ v :$
            $Attr(u,\ v,\ k) \equiv Attr(u,\ v,\ k+1)$
    IN
        $Attr(x,\ y,\ r)$
```

In this definition *Goal* is a first-order operator. An attractor definition with *Goal* being a set is possible too [103, §IV-A]. The above operators are simpler cases of those that we define in later sections. Dual to the attractor is the trap operator, defined as follows

```
$kStepSafe(x,\ y,\ Stay(\_,\ \_),\ Escape(\_,\ \_),\ k) \triangleq$
  LET RECURSIVE $F(\_,\ \_,\ \_)$
      $F(u,\ v,\ m) \triangleq$ IF $m = 0$
          THEN TRUE
          ELSE LET $Safe(a,\ b) \triangleq F(a,\ b,\ m-1)$
```

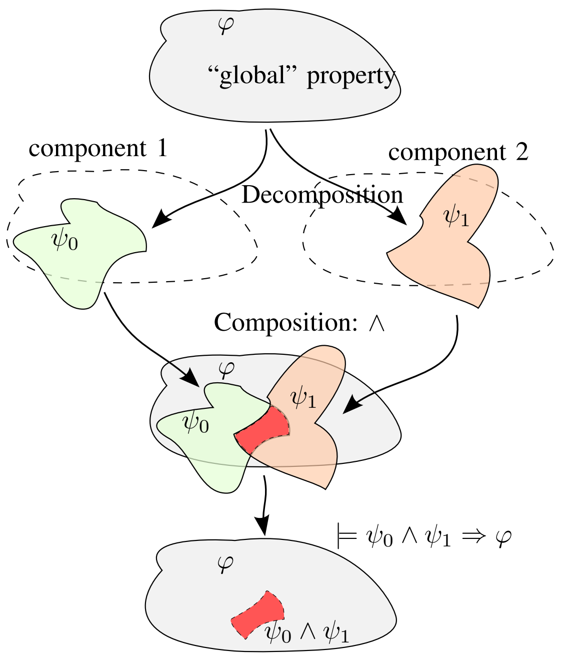


Fig. 4: Each component is specified by a property that may allow behaviors that violate the desired global property. These undesired behaviors would be caused by arbitrary behavior of other components. Nonetheless, when we conjoin the component specifications of the contract, the result implies the global property, because of mutual fulfillment of assumptions between components.

```
                IN   $\vee$ $Escape(u, v)$
                     $\vee$ $\wedge$ $Stay(u, v)$
                         $\wedge$ $Step(u, v, Safe)$
   IN   $F(x, y, k)$

$Trap(x, y, Stay(\_, \_), Escape(\_, \_)) \triangleq$
    LET
        $Safe(u, v, n) \triangleq kStepSafe(u, v, Stay, Escape, n)$
        $r \triangleq$ CHOOSE $k \in Nat : \forall u, v :$
            $Safe(u, v, k) \equiv Safe(u, v, k+1)$
    IN
        $Safe(x, y, r)$
```

Game solving involves reasoning about sets of states. Symbolic methods using binary decision diagrams (BDDs) [24] are used for compactly representing sets of states, instead of enumeration. To use BDDs for specifications in untyped logic we need to identify those (integer) values that are relevant, a common requirement that arises in automated reasoning [104]. This information is declared as type hints [105] to enable automatically rewriting the problem in terms of newly declared variables, so that all relevant values be Boolean (instead of integer), thus enabling use of BDDs. This process is called bitblasting and bears similarity to program compilation.

IV. Contracts

A. Assume-guarantee contracts between components

The purpose of a contract is to represent the assumptions that each component in an assembly makes about other components, and the guarantees that it provides when these assumptions are satisfied. The assignment of obligations to components should be balanced. It is unreasonable to specify an assumption by one component that is infeasible by any other component. So the specifications should suffice for ensuring that the assembly behaves as desired, and also not overconstrain any of the components. We can view these requirements as placing a lower and an upper bound on component specifications. The lower bound ensures that each component is implementable, and the upper bound ensures that the assembled system operates correctly. These requirements are formalized with the following definition. A *contract* [106] is a partition of variables among n components, defined by n actions

$$eA(\_, \_), \ldots, eW(\_, \_)$$

and a collection of temporal properties

$$A(\_, \_), \ldots, W(\_, \_)$$

that satisfy the following theorem schema

$$\begin{aligned}
&\wedge IsRealizable(A, eA)\\
&\vdots\\
&\wedge IsRealizable(W, eW)\\
&\wedge \big(A(x, y) \wedge \ldots \wedge W(z, w)\big) \Rightarrow Phi(x, \ldots, w)
\end{aligned}$$

In other words, a contract is a collection of assume-guarantee properties for each component that are realizable and conjoined imply the desired behavior for the system assembled from those components. The notion of composition of properties is illustrated in Fig. 4.

Remark 2: Two alternative definitions are possible. Instead of using the letters $A, \ldots, W$ we can use an index j, with the understanding that j is part of the identifier, not a value in the object language (here TLA$^+$) [107]. Another approach is to incorporate the index within the object language. This can be done by defining a single property parametrized by an index, for example $Prop(\_, \_, \_)$ and $eR(\_, \_, \_)$. The schema can then be written as

```
$\wedge \forall j \in 1 \,..\, n :$  LET $R(u, v) \triangleq Prop(u, v, j)$
                    $e(u, v) \triangleq eR(u, v, j)$
                IN  $IsRealizable(R, e)$
$\wedge$ $\forall x, y :$
    $(\forall j \in 1 \,..\, n : \; Prop(x[j], y[j], j)) \Rightarrow Phi(x, y)$
```

□

Example 1: As an example used throughout the paper, consider a charging station for mobile robots that has two spots, and a robot that requests a spot for charging. The charging station keeps track of which spots are taken (variables $spot\_1, spot\_2$), as well as the spot number that becomes available for the robot to dock ($free\_x, free\_y$), when the variable $free = 1$. The robot can request docking by setting the variable req, and is represented by its coordinates on the plane pos_x, pos_y. Not all spots are free. One other spot is occupied by another robot, which forms part of the environment of the charging station and the robot. This spot is indicated by the variable occ, and to keep the example small, occ remains unchanged through time, so the occupied spot does not change, but neither the station nor the robot control which this spot is. The specification of the entire system is the following:

$$\begin{array}{l}
\text{EXTENDS } \mathit{Integers}\\
\text{VARIABLES } \mathit{spot\_1},\ \mathit{spot\_2},\ \mathit{free\_x},\ \mathit{free\_y},\ \mathit{free},\\
\qquad\qquad \mathit{req},\ \mathit{pos\_x},\ \mathit{pos\_y},\ \mathit{occ},\ \mathit{turn}\\
\mathit{station\_vars} \triangleq \langle \mathit{spot\_1},\ \mathit{spot\_2},\ \mathit{free\_x},\ \mathit{free\_y},\ \mathit{free} \rangle\\
\mathit{robot\_vars} \triangleq \langle \mathit{req},\ \mathit{pos\_x},\ \mathit{pos\_y} \rangle\\
\mathit{vars} \triangleq \langle \mathit{station\_vars},\ \mathit{robot\_vars},\ \mathit{occ},\ \mathit{turn} \rangle\\
\mathit{StationStep} \triangleq \begin{array}[t]{l} \land\ \mathit{turn} = 1\\ \land\ (\mathit{req} = 0) \Rightarrow (\mathit{free}' = 0) \end{array}\\
\mathit{StationNext} \triangleq\\
\quad \land\ \mathit{spot\_1} \in 0\,..\,1 \land \mathit{spot\_2} \in 0\,..\,1 \land \mathit{free} \in 0\,..\,1\\
\quad \land\ \mathit{free\_x} \in 0\,..\,18 \land \mathit{free\_y} \in 0\,..\,18\\
\quad \land\ \lor\ (\mathit{free} = 0)\\
\qquad \lor\ (\mathit{free\_x} = 1 \land \mathit{free\_y} = 1 \land \mathit{spot\_1} = 0)\\
\qquad \lor\ (\mathit{free\_x} = 2 \land \mathit{free\_y} = 1 \land \mathit{spot\_2} = 0)\\
\quad \land\ (\mathit{free} = 1) \Rightarrow \begin{array}[t]{l} \land\ \mathit{spot\_1} = 0 \ \Rightarrow\ \mathit{occ} \neq 1\\ \land\ \mathit{spot\_2} = 0 \ \Rightarrow\ \mathit{occ} \neq 2 \end{array}\\
\quad \land\ \mathit{StationStep} \lor \text{UNCHANGED } \mathit{station\_vars}
\end{array}$$

$$\begin{array}{l}
\mathit{RobotNext} \triangleq\\
\quad \land\ \mathit{pos\_x} \in 1\,..\,15 \land \mathit{pos\_y} \in 1\,..\,15 \land \mathit{req} \in 0\,..\,1\\
\quad \land\ ((\mathit{req} = 1 \land \mathit{req}' = 0) \Rightarrow \begin{array}[t]{l} \land\ \mathit{free} = 1\\ \land\ \mathit{free\_x} = \mathit{pos\_x}'\\ \land\ \mathit{free\_y} = \mathit{pos\_y}') \end{array}\\
\quad \land\ (\mathit{turn} = 2) \lor \text{UNCHANGED } \mathit{robot\_vars}
\end{array}$$

$$\begin{array}{l}
\mathit{OthersNext} \triangleq (\mathit{occ} \in 1\,..\,3) \land (\mathit{occ}' = \mathit{occ})\\
\mathit{SchedulerNext} \triangleq \begin{array}[t]{l} \land\ \mathit{turn} \in 1\,..\,2\\ \land\ (\mathit{turn} = 1) \Rightarrow (\mathit{turn}' = 2)\\ \land\ (\mathit{turn} = 2) \Rightarrow (\mathit{turn}' = 1) \end{array}
\end{array}$$

$$\begin{array}{l}
\mathit{Env} \triangleq \begin{array}[t]{l} \land\ \mathit{turn} \in 1\,..\,2 \ \ \land\ \mathit{occ} \in 1\,..\,3\\ \land\ \Box[\mathit{OthersNext} \land \mathit{SchedulerNext}]_{\mathit{vars}}\\ \land\ \Box\Diamond\langle \mathit{SchedulerNext} \rangle_{\mathit{turn}} \end{array}\\
\mathit{Init} \triangleq \begin{array}[t]{l} \land\ \mathit{spot\_1} = 0 \land \mathit{spot\_2} = 0\\ \land\ \mathit{free\_x} = 0 \land \mathit{free\_y} = 0 \land \mathit{free} = 0\\ \land\ \mathit{pos\_x} = 1 \land \mathit{pos\_y} = 1 \land \mathit{req} = 0 \end{array}\\
\mathit{Next} \triangleq \mathit{StationNext} \land \mathit{RobotNext}\\
L \triangleq \Box\Diamond(\mathit{req} = 0) \land \Box\Diamond(\mathit{req} = 1)\\
\mathit{Assembly} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}} \land L\\
\mathit{Phi} \triangleq \mathit{Env} \Rightarrow \mathit{Assembly}
\end{array}$$

We wrote the property $\mathit{Assembly}$ using implication. In general, the operator Unzip (defined in Sec. IV-B) or a variant should be used instead of implication. Nonetheless, in this case the environment can realize Env independently of the two components, so we can simply use implication (and defer discussing how open-systems should be defined to Sec. IV-B). An alternative would be to let Phi be the conjunction $\mathit{Env} \land \mathit{Assembly}$ (a closed-system). In that case, we would have to consider components for the scheduler and other robots. Let the actions

$$\begin{array}{l}
\mathit{eStation}(\mathit{turn}) \triangleq (\mathit{turn} = 1) \land (\mathit{turn}' \neq \mathit{turn})\\
\mathit{eRobot}(\mathit{turn}) \triangleq (\mathit{turn} = 2) \land (\mathit{turn}' \neq \mathit{turn})
\end{array}$$

define when each implementation takes nonstuttering steps. Note that the property $\mathit{Assembly}$ defines synchronous and interleaving changes to the components. This is the case for the implementations too, due to the presence of variable turn in $\mathit{eStation}$ and eRobot, together with the antecedent Env in formula Phi.

A contract between the charging station and the robot has the form of two properties PhiS, PhiR such that

$$\begin{array}{l}
\land\ \mathit{IsRealizable}(\mathit{PhiS},\ \mathit{eStation})\\
\land\ \mathit{IsRealizable}(\mathit{PhiR},\ \mathit{eRobot})\\
\land\ \lor\ \neg \land\ \mathit{PhiS}(\mathit{spot\_1},\ \mathit{spot\_2},\ \mathit{free\_x},\ \mathit{free\_y},\\
\qquad\qquad\qquad \mathit{free},\ \mathit{req},\ \mathit{occ},\ \mathit{turn})\\
\qquad\quad\ \land\ \mathit{PhiR}(\mathit{req},\ \mathit{pos\_x},\ \mathit{pos\_y},\ \mathit{free\_x},\ \mathit{free\_y},\\
\qquad\qquad\qquad \mathit{free},\ \mathit{turn})\\
\quad\ \lor\ \mathit{Phi}
\end{array}$$

We used operators that take several arguments, instead of only two, but this difference is inessential (we could have used record-valued variables instead). To demonstrate what the realizability condition means for the robot, we can define it as

$$\begin{array}{l}
\mathit{IsRealizable}(\mathit{PhiR}(\_,\_,\_,\_,\_,\_,\_),\ \mathit{eR}(\_)) \triangleq\\
\quad \exists\, \mathit{f\_req},\ \mathit{f\_pos\_x},\ \mathit{f\_pos\_y},\ g,\\
\qquad\quad \mathit{req}0,\ \mathit{pos\_x}0,\ \mathit{pos\_y}0,\ \mathit{mem}0 :\\
\qquad \land\ \mathit{IsAFiniteFcn}(g) \land \mathit{IsAFiniteFcn}(\mathit{f\_req})\\
\qquad \land\ \mathit{IsAFiniteFcn}(\mathit{f\_pos\_x}) \land \mathit{IsAFiniteFcn}(\mathit{f\_pos\_y})\\
\qquad \land\ \forall\, \mathit{req},\ \mathit{pos\_x},\ \mathit{pos\_y},\ \mathit{free\_x},\ \mathit{free\_y},\ \mathit{free},\ \mathit{turn} :\\
\qquad\quad \text{LET } R(\mathit{mem}) \triangleq\\
\qquad\qquad \text{LET } v \triangleq \langle \mathit{mem},\ \mathit{req},\ \mathit{pos\_x},\ \mathit{pos\_y},\ \mathit{free\_x},\\
\qquad\qquad\qquad\qquad \mathit{free\_y},\ \mathit{free},\ \mathit{turn} \rangle\\
\qquad\qquad\qquad N \triangleq \begin{array}[t]{l} \land\ \mathit{mem}' = g[v]\\ \land\ \mathit{req}' = \mathit{f\_req}[v]\\ \land\ \mathit{pos\_x}' = \mathit{f\_pos\_x}[v]\\ \land\ \mathit{pos\_y}' = \mathit{f\_pos\_y}[v] \end{array}\\
\qquad\qquad\qquad e \triangleq \mathit{eR}(\mathit{turn})\\
\qquad\qquad \text{IN } \begin{array}[t]{l} \land\ \mathit{mem} = \mathit{mem}0 \land \mathit{req} = \mathit{req}0\\ \land\ \mathit{pos\_x} = \mathit{pos\_x}0 \land \mathit{pos\_y} = \mathit{pos\_y}0\\ \land\ \Box[e \Rightarrow N]_v\\ \land\ \mathrm{WF}_{\langle \mathit{mem},\, \mathit{req},\, \mathit{pos\_x},\, \mathit{pos\_y} \rangle}(e \land N) \end{array}\\
\qquad\quad \text{IN}\\
\qquad\qquad (\exists\, \mathit{mem} :\ R(\mathit{mem}))\\
\qquad\qquad\quad \Rightarrow \mathit{PhiR}(\mathit{req},\ \mathit{pos\_x},\ \mathit{pos\_y},\ \mathit{free\_x},\\
\qquad\qquad\qquad\quad \mathit{free\_y},\ \mathit{free},\ \mathit{turn})
\end{array}$$

□

Remark 3: Realizability can be defined without design of initial conditions for the component (the $\exists \mathit{req}0, \mathit{pos\_x}0, \mathit{pos\_y}0$ above). With such a definition, the property PhiR should not constrain the component's initial state, because that would render PhiR unrealizable [96, §3.3, pp. 14–16]. So initial conditions within PhiR would be part of an antecedent. In consequence, a conjunction of realizable component specifications would not imply any closed-system property. The choice between these different definitions is a matter of specification style. □

B. Open systems

1) Defining an open system: An open system is one constrained with respect to variables it does not control [47], [99, §3.1], [88, §9.5.3]. Usually the components we build rely on their environment in order to operate as intended. For example, a laptop should be able to connet to the Internet, but this is impossible in absence of a wired or wireless network compatible with the laptop's interface (ports or other). If we describe the laptop as a system that is able to connect to the Internet, our specification is fictitious, because it wrongly predicts that the laptop will be online in the middle of a desert. We could augment the specification by adding that there is a wireless network, and that the laptop connects to it. In this attempt we are *overspecifying*, by promising to deliver both a laptop and a wireless network. Laptops are usually designed separately from the buildings that host wireless networks. What we should instead do is to guarantee a connection to the Internet *assuming* that a wireless network is available. In absence of a network, the laptop is free to remain disconnected.

The notion of *open* system can be defined mathematically using the notion of proper class. As remarked earlier, any temporal property defines a proper class of behaviors. So any system defined by a temporal property is modeled by a proper class of behaviors. This indicates that the notion of open system is *relative* to some designated variables. These observations motivate the following definition.

$$\begin{array}{l}
\mathit{IsAClosedSystem}(P(\_)) \triangleq \\
\quad \exists\, S : \ \boldsymbol{\forall}\, v : \ P(v) \Rightarrow \Box(v \in S) \\
\mathit{IsAnOpenSystem}(P(\_)) \triangleq \neg \mathit{IsAClosedSystem}(P) \\
\text{THEOREM} \\
\quad \text{ASSUME TEMPORAL } P(\_) \\
\quad \text{PROVE } \ \mathit{IsAnOpenSystem}(P) \equiv \\
\qquad \forall\, S : \ \boldsymbol{\exists}\, v : \ P(v) \wedge \Diamond(v \notin S)
\end{array}$$

Let x be a variable and $P(x)$ a temporal property. The property $P(x)$ defines an *open system* if and only if $\mathit{IsAnOpenSystem}(P)$. So the possibility of *diverging behavior* characterizes a system as open. A property P defines a closed-system if it implies a type invariant that bounds all variables that occur in P. Therefore, closed systems can be defined using Δ_0 formulas [108, p.161]. Diverging behavior is also the main concept in how initial conditions affect realizability [96, Lemma 6, p.12].

Remark 4: Thus, cardinality distinguishes an open from a closed system; a criterion applicable even if we decide to restrict our attention to only finitely many states. Considering a system closed if its variables take values from a finite set, an open system is one whose variables take values from an infinite set. □

2) Specifying interaction with an environment: A component's specification should not constrain its environment. This is expressible with a formula that spreads implication incrementally over a behavior [70], [109], [110], [48], [111], [41]. We define the operator *Unzip* for forming open-systems from closed-systems

$$\begin{array}{l}
\mathit{WhilePlusHalf}(A(\_,\_),\ G(\_,\_),\ x,\ y) \triangleq \\
\ \boldsymbol{\forall}\, b : \ \vee \neg \wedge b \in \text{BOOLEAN} \ \wedge \Box[b' = \text{FALSE}]_b \\
\qquad\quad \wedge\, \boldsymbol{\exists}\, u,\, v : \ \wedge A(u,\, v) \\
\qquad\qquad\qquad\qquad \wedge \Box \vee b \neq \text{TRUE} \\
\qquad\qquad\qquad\qquad\quad\ \vee \langle u,\, v\rangle = \langle x,\, y\rangle \\
\qquad \vee\, \boldsymbol{\exists}\, u,\, v : \ \wedge G(u,\, v) \wedge (v = y) \\
\qquad\qquad\qquad\ \wedge \Box \vee b \neq \text{TRUE} \\
\qquad\qquad\qquad\quad\ \vee \langle u,\, v\rangle = \langle x,\, y\rangle \\
\qquad\qquad\qquad \wedge \Box[(b = \text{TRUE}) \Rightarrow (v' = y')]_{\langle b,\, v,\, y\rangle} \\
\mathit{Unzip}(P(\_,\_),\ x,\ y) \triangleq \\
\quad \text{LET } A(u,\, v) \triangleq \mathit{WhilePlusHalf}(P,\ P,\ v,\ u) \\
\quad \text{IN } \ \mathit{WhilePlusHalf}(A,\ P,\ x,\ y)
\end{array}$$

The operator *WhilePlusHalf* is a slight variant of how the "while-plus" operator $\overset{+}{-\!\triangleright}$ can be defined within TLA$^+$ [10, p.337], [48, p.262] ($\overset{+}{-\!\triangleright}$ is defined by TLA$^+$ semantics [10, p.316]). If A, G are temporal operators, then $\mathit{WhilePlusHalf}(A, G)$ can be thought of as being true of a behavior σ if every finite prefix of σ that can be extended to a behavior that satisfies A can also be extended, starting with a state that satisfies $v = y$, to a behavior that satisfies G.

Remark 5 (Comparison to $\overset{+}{-\!\triangleright}$ *):* Only v is constrained in the first state of the suffix, thus the "half" in the name. In contrast, the operator $\overset{+}{-\!\triangleright}$ constrains both u and v in the first state of the suffix. For disjoint-state specifications, this additional constraint results in unrealizability, using the definition of synthesis from Sec. III-C. To obtain a realizable property, the property G should be sufficiently permissive [96, §5.2.4, pp. 26–27]. However, this leads to possible underspecification of what $\boldsymbol{\exists}\, u, v \ : \ G(u, v) \ldots$ means (i.e., the closure of G may allow behavior undetermined by the axioms of the logic). These observations motivate the above modification, which corresponds to definitions of "strict implication" from the literature on games [100]. □

The operator *Unzip* takes a closed-system property and yields an open-system property, and roughly means

> While the environment does not take any step that definitely blocked the assembly, the component's next step should not definitely block the assembly, and the assembly should not have been blocked in the past.

Writing a closed-system specification is typically easier than reasoning about how to separate it into two properties A, G. More fundamentally, the environment behavior should be mentioned in its entirety within G. Otherwise, the disjunct that contains G can allow the component to behave as if the environment has unreasonable future capabilities. How *Unzip* is defined is reminiscent of how $\overset{+}{-\!\triangleright}$ is defined for safety properties in terms of $-\!\triangleright$ [48, p.262], [110, Prop.1, p.501].

For the synthesis of implementations for properties specified using *Unzip*, relating this operator to existing results about synthesis from GR(1) properties is useful [100]. This is possible, via the following definition in raw TLA$^+$

$$\begin{array}{l}
\mathit{AsmGrt}(\mathit{Init}(\_,\_),\ \mathit{EnvNext}(\_,\_,\_),\ \mathit{SysNext}(\_,\_,\_), \\
\qquad \mathit{Next}(\_,\_,\_,\_),\ \mathit{Liveness}(\_,\_),\ x,\ y) \triangleq \\
\quad \wedge \quad \exists\, u : \ \mathit{Init}(u,\, y) \\
\quad \wedge \quad \vee \neg\exists\, v : \ \mathit{Init}(x,\, v) \\
\qquad\quad \vee \wedge \mathit{Init}(x,\, y) \\
\qquad\qquad \wedge \Box(\mathit{Earlier}(\mathit{EnvNext}(x,\, y,\, x')) \\
\qquad\qquad\qquad \Rightarrow \wedge \mathit{Earlier}(\mathit{Next}(x,\, y,\, x',\, y')) \\
\qquad\qquad\qquad\quad\ \ \wedge \mathit{SysNext}(x,\, y,\, y')) \\
\qquad\qquad \wedge (\Box \mathit{EnvNext}(x,\, y,\, x')) \Rightarrow \mathit{Liveness}(x,\, y)
\end{array}$$

The second conjunct expresses "stepwise implication", so that if at some step the environment violates the assumed action *EnvNext*, then the system is not obliged to satisfy the action

SysNext in later steps. The operator $AsmGrt$ is a modification of [109] to avoid circularity [106], [111, §5, ▷ on p.59]. The operator $Earlier$ abbreviates the composition of the past LTL operators $WeakPrevious$ and $Historically$ (expressible in TLA$^+$ using temporal quantification and history variables). It is possible to define $Earlier$ by using a modified satisfaction relation $\models$ in raw TLA$^+$, but we will omit this definition here.

It can be shown that if $P \equiv \mathit{Init} \wedge \Box[\mathit{Next}]_{\langle x,y\rangle} \wedge L$, where L a GR(1) liveness property, and the pair of properties $\mathit{Init} \wedge \Box[\mathit{Next}]_{\langle x,y\rangle}, L$ is machine-closed (meaning that L does not constrain the safety property $\mathit{Init} \wedge \Box[\mathit{Next}]_{\langle x,y\rangle}$ [70]), then an implementation synthesized for the property

$$\begin{aligned}&AsmGrt(\mathit{Init}, [\exists y' : \mathit{Next}]_{\langle x,y\rangle}, [\exists x' : \mathit{Next}]_{\langle x,y\rangle},\\ &\qquad [\mathit{Next}]_{\langle x,y\rangle}, L, x, y)\end{aligned}$$

using existing algorithms [100] also realizes $\mathit{Unzip}(P, x, y)$.

Remark 6 (Symmetry): Dijkstra requires symmetry from solutions to the mutual exclusion problem [112, item (a)]. The approach we follow asserts that all components are implemented as “Moore machines” (f and g are independent of primed variable values). Alternatives are possible where one component is Mealy and its environment Moore [100], [71]. Specifying such components in a way that avoids circular reasoning leads to using more than one operators for defining open-systems, which is asymmetric. In the presence of multiple components, a spectrum of Moore to Mealy machines needs to be considered, not unlike typed components [59]. $\Box$

Example 2: The specification of the robot in the charging station example can be defined using the operator Unzip by first defining a closed-system property that describes the robot together with its environment $P \triangleq \mathit{Env} \wedge \mathit{Assembly}$ and let $\mathit{Unzip}(P, \mathit{spot\_1}, \ldots, \mathit{turn}, \mathit{req}, \mathit{pos\_x}, \mathit{pos\_y})$ specify the robot (the number of arguments has been adapted, as in similar remarks above). In the next section, we will see how some of these external variables can be eliminated to define a property for the robot that mentions fewer details about the rest of the system. $\Box$

We consider specifications that are interleaving [10, p.137] among most of the components involved (the scheduler is an exception), in that they allow variables of only one component to change in each step (a “move”). Components move in a fixed order that repeats, so the resulting interaction can be viewed as a turn-based game between the components. Each variable is controlled by a single component throughout time, thus the specifications are disjoint-state [10, p.144]. Let C be a collection of indices that identify components, and e_i be actions that attribute state changes to components (as the e in *Realization*, Sec. III-C), and v a tuple of relevant variables. The concept of interleaving can be defined as the following pairwise orthogonality condition [70, p.514], [96, §5, p.22]

$$\mathit{Interleaving} \triangleq \Box[\forall i, j \in C : (i \neq j) \Rightarrow (e_j \Rightarrow \neg e_i)]_v$$

In words, no step changes the state of more than one component. Let V be a collection of indices of variable identifiers x_k, and $e_{j,k}$ attribute changes of variable x_k to component j. The concept of disjoint state can be defined as follows [96, §5.1.2, p.24].

$$\begin{aligned}\mathit{DisjointState} \triangleq\ &\forall k \in V : \exists i \in C :\\ &\qquad \forall j \in C \setminus i : \Box[\neg e_{j,k}]_{x_k}\end{aligned}$$

In words, all changes to each variable are attributed to no more than one component. These definitions are schematic, in that i, j, k are metatheoretic notation [107].

V. PARAMETRIZED HIDING OF VARIABLES

A. Motivation and overview

Precision is essential for specification, but adding details makes a specification less manageable by both humans and machines. Decomposition in general involves as much computation as solving the problem in a monolithic way [7]. Structuring the specification hierarchically to defer introducing lower-level details is a solution in the middle. Hierarchy corresponds to how real systems are designed, for example airplanes. The deferred details should be irrelevant to the higher level design, and specific to subsystems only. Some internal component details may be relevant to the higher levels, and be mentioned before decomposition of a specification to component specifications. Mentioning these details can make writing the specification easier, or these details may concern the interaction of some components, but not others. Another case is state that is relevant to designing how components interact but that need not be communicated during operation.

We want to remove irrelevant details from the specification of each component. We do so by detecting which variables can be eliminated from a component's specification. The specification that results after the selected variables have been eliminated should be realizable, otherwise no component that implements that specification exists. This objective can be summarized as follows.

Problem 1 (Hiding variables): Given a realizable open-system GR(1) property φ, which environment variables can be hidden from the component (by making the values of those variables unavailable to the controller function) without preventing the component from realizing φ?

In this section, we *parametrize the selection* of which variables to hide. This parametrization is obtained by modifying the controllable step operator. This operator is used in later sections to *construct* a property φ that is realizable, by reasoning about dependencies between components.

Synthesizing implementations from specifications with partial information is computationally hard. Since the hidden variables are chosen to preserve realizability, and reasoning about hidden behavior at the time of component synthesis is computationally expensive, we eliminate the hidden variables. We show that the resulting specifications can be written as if only the visible variables were declared to the component, thus as if the component had full information; an objective summarized as follows.

Problem 2 (Expressibility): Can we write the component specifications with formulas in which hidden variables do not occur?

In Sec. V-C we discuss the abstraction from the controllable step operator for specific variables, and in Sec. V-D we parameterize the *choice* of which variables to hide. We start by considering the safety part of the specification in Sec. V-B.

B. Preventing safety violations

The starting point is a specification for the assembled system. Suppose that this is a property of the form

$$Assembly \;\triangleq\; Init \land \Box[Next]_{vrs} \land Liveness$$

where the conjunct $Liveness$ is a conjunction of liveness properties, for example $\Box\Diamond Goal_1 \land \Box\Diamond Goal_2$. The property $Liveness$ can impose constraints on the safety property $SM \triangleq Init \land \Box[Next]_{vrs}$. If this is not the case, then the pair of properties $SM, Liveness$ is called *machine-closed* [113, p.261], [70, p.519]. We are about to decompose the property $Assembly$. To avoid circularity, we create the safety and liveness parts of component properties separately, in two stages. Only "pieces" of liveness constraints will end up in each component's specification. This means that we should ensure that the safety part is strong enough to prevent any component from "straying away" to an extent that would violate the assembly's $Liveness$ property.

Having established this goal, let us focus on safety. The property SM may be too permissive to ensure that $Liveness$ will be satisfiable in the future. We need to strengthen SM. The weakest safety property W that suffices is the strongest safety property implied by $Assembly$, i.e., such that

$$\models Assembly \Rightarrow W.$$

The property W is known as *closure* of the property $Assembly$ [114, p.120], [70, p.518], [48, pp. 261–262], due to topological considerations [94]. If W is written in the form

$$W \equiv Init \land \Box[Next]_{vrs} \land \Box Inv,$$

then the invariant Inv defines the largest set of states that can occur in any behavior that satisfies the property $Assembly$. The weakest invariant yields also the (unique) weakest safety assumption necessary in turn-based games with full information (set of cooperatively winning states) [69], [106, §III-A].

Example 3: For the charging station example, the assembly invariant Inv (when Env holds) is

$$\begin{array}{l}
\land\ turn \in 1 \,..\, 2 \land free \in 0 \,..\, 1 \\
\land\ free\_x \in 0 \,..\, 18 \land free\_y \in 0 \,..\, 18 \land occ \in 1 \,..\, 3 \\
\land\ pos\_x \in 1 \,..\, 15 \land pos\_y \in 1 \,..\, 15 \\
\land\ spot\_1 \in 0 \,..\, 1 \land spot\_2 \in 0 \,..\, 1 \\
\land \quad \lor \quad \land (free\_x = 1) \land (free\_y = 1) \land (occ \in 2 \,..\, 3) \\
\qquad\qquad \land (spot\_1 = 0) \land (spot\_2 = 1) \\
\qquad \lor \quad \land (free\_x = 2) \land (free\_y = 1) \land (occ = 1) \\
\qquad\qquad \land (spot\_1 = 1) \land (spot\_2 = 0) \\
\qquad \lor \quad \land (free\_x \in 1 \,..\, 2) \land (free\_y = 1) \land (occ = 3) \\
\qquad\qquad \land (spot\_1 = 0) \land (spot\_2 = 0) \\
\qquad \lor (free = 0) \\
\qquad \lor \land (free\_x = 2) \land (free\_y = 1) \land (occ = 3) \\
\qquad\quad \land (spot\_2 = 0)
\end{array}$$

This invariant was symbolically computed as the greatest fixpoint of the assembly's action. The BDD resulting from this computation was then converted to a minimal formula in disjunctive normal form, with integer variable constraints as conjuncts. □

A component's action should constrain the next values of only variables that the component controls. In addition, the component should be constrained to preserve the invariant Inv. The property W can be written as $Init \land Inv \land \Box[WNext]_{vrs}$, where [88, by INV2, Fig.5, p.888]

$$WNext \;\triangleq\; Inv \land Next \land Inv'.$$

The property $Unzip(Assembly, x, y)$ can be defined by quantifying the primed variables of other components, as follows

$$\begin{array}{l}
EnvNext(x,\, y,\, x') \;\triangleq \\
\quad \exists\, y' : \; \land\, Inv(x,\, y) \land Inv(x',\, y') \\
\qquad\qquad \land\, Next(x,\, y,\, x',\, y') \\
SysNext(x,\, y,\, y') \;\triangleq \\
\quad \exists\, x' : \; \land\, Inv(x,\, y) \land Inv(x',\, y') \\
\qquad\qquad \land\, Next(x,\, y,\, x',\, y')
\end{array}$$

For specifications that are interleaving for all components except a deterministic scheduler, as those we discuss are (in general, for specifications that in each step allow multiple alternatives for state changes for at most one component), the $Step$ operator with the above actions implies that $Inv \land Next$ is satisfied by each step. The reason is that in each step, at most one component can change in a non-unique way.

Remark 7: In the charging station example, any nonstuttering step of the assembly is a nonstuttering step of the scheduler, which is assumed to take infinitely many nonstuttering steps. The fixpoint algorithms we develop correspond to a raw TLA$^+$ context. When transitioning to the raw logic, after stuttering steps are removed, the property $\Box\Diamond\langle SchedulerNext\rangle_{turn}$ reduces to safety, because any nonstuttering step of the assembly changes the variable $turn$. □

C. Hiding specific variables

Suppose we want to hide variable h in predicate $P(h, x, y, y')$. The environment controls variables h and x, and the component y. If we use unbounded quantification, $\forall h : P(h, x, y, y')$, then in most cases the result will be too restrictive, or FALSE. The quantified variable h should be bounded, so a suitable antecedent $Bound$ is needed. Using this bound should not permit previously unallowed values for x and y, thus

$$\begin{array}{l}
\land\, \exists\, h : \; Bound(h,\, x,\, y) \\
\land\, \forall\, h : \; Bound(h,\, x,\, y) \Rightarrow P(h,\, x,\, y,\, y')
\end{array}$$

We will use $Inv(h, x, y)$ as a bound on h. It will be the case that $\models Bound(h, x, y) \Rightarrow \exists y' : P(h, x, y, y')$. As defined in Sec. III, the controllable step operator when the component can observe the values of variables x and h takes the form (to reduce verbosity we omit the argument $Target$)

$$\begin{array}{l}
Step(x,\, y,\, h) \;\triangleq\; \exists\, y' : \; \forall\, x',\, h' : \\
\qquad \land\, SysNext(h,\, x,\, y,\, y') \\
\qquad \land\, EnvNext(h,\, x,\, y,\, h',\, x') \Rightarrow Target(h',\, x',\, y')
\end{array}$$

The component's decisions cannot depend on the variable h, leading to the modified operator

$StepH(x,\, y) \;\triangleq$

$$\begin{array}{l}\wedge\ \ \exists\, h:\ \mathit{Inv}(h,\, x,\, y)\\ \wedge\ \ \exists\, y':\ \forall\, x',\, h':\ \forall\, h:\\ \quad \vee\ \neg \mathit{Inv}(h,\, x,\, y)\\ \quad \vee\ \wedge\ \mathit{SysNext}(h,\, x,\, y,\, y')\\ \qquad \wedge\ \mathit{EnvNext}(h,\, x,\, y,\, h',\, x') \Rightarrow \mathit{Target}(h',\, x',\, y')\end{array}$$

Algebraic manipulation yields

$$\begin{array}{l}\mathit{StepH}(x,\, y) \equiv\\ \quad \exists\, y':\ \forall\, x':\\ \qquad \wedge\ \wedge \exists\, h:\ \mathit{Inv}(h,\, x,\, y)\\ \qquad\quad \wedge \forall\, h: \mathit{Inv}(h,\, x,\, y) \Rightarrow \mathit{SysNext}(h,\, x,\, y,\, y')\\ \qquad \wedge \forall\, h',\, h:\\ \qquad\quad \vee \neg \wedge \mathit{Inv}(h,\, x,\, y)\\ \qquad\qquad \wedge\ \mathit{EnvNext}(h,\, x,\, y,\, h',\, x')\\ \qquad\quad \vee\ \mathit{Target}(h',\, x',\, y')\end{array}$$

If Target is independent of h', (which is the case in Sec. VI), then confining universal quantification to the first disjunct yields

$$\begin{array}{l}\mathit{StepH}(x,\, y) \equiv\\ \quad \exists\, y':\ \forall\, x':\\ \qquad \wedge\ \wedge \exists\, h:\ \mathit{Inv}(h,\, x,\, y)\\ \qquad\quad \wedge \forall\, h: \mathit{Inv}(h,\, x,\, y) \Rightarrow \mathit{SysNext}(h,\, x,\, y,\, y')\\ \qquad \wedge\ \vee \neg\exists\, h,\, h':\ \ \wedge \mathit{Inv}(h,\, x,\, y)\\ \qquad\qquad\qquad\qquad \wedge \mathit{EnvNext}(h,\, x,\, y,\, h',\, x')\\ \qquad\quad \vee\ \mathit{Target}(x',\, y')\end{array}$$

By defining

$$\begin{array}{l}\mathit{SimplerSysNext}(x,\, y,\, y') \triangleq\\ \quad \wedge \exists\, h:\ \mathit{Inv}(h,\, x,\, y)\\ \quad \wedge \forall\, h:\ \mathit{Inv}(h,\, x,\, y) \Rightarrow \mathit{SysNext}(h,\, x,\, y,\, y')\end{array}$$

$$\begin{array}{l}\mathit{SimplerEnvNext}(x,\, y,\, x') \triangleq\\ \quad \exists\, h,\, h':\ \ \wedge \mathit{Inv}(h,\, x,\, y)\\ \qquad\qquad\quad \wedge \mathit{EnvNext}(h,\, x,\, y,\, h',\, x')\end{array}$$

we obtain

$$\begin{array}{l}\mathit{StepH}(x,\, y) \equiv\\ \quad \exists\, y':\ \forall\, x':\\ \qquad \wedge\ \mathit{SimplerSysNext}(x,\, y,\, y')\\ \qquad \wedge\ \mathit{SimplerEnvNext}(x,\, y,\, x') \Rightarrow \mathit{Target}(x',\, y')\end{array}$$

The resulting operator StepH is schematically the same with that for the full information case. So the open-system specification with hidden variables can be rewritten as an open-system specification with no hidden variables, without changing the set of states from where the property is realizable. The eliminated variables do not appear in the component specification, so further work focusing on that component can be carried out in a full information context, including GR(1) synthesis. The action $\mathit{SimplerEnvNext}$ abstracts environment details. Abstraction for the environment is appropriate in a refined open-system property, because of contravariance between component and environment (assumptions should be weakened, guarantees strengthened) [52], [3, Eq.(4.14), p.325].

Example 4: To demonstrate the effect of hiding variables in the context of the charging station example, consider the action of the charging station's environment. Without hiding any state from the station, the environment action is (shown for steps that it is the robot's turn to change)

$$\begin{array}{l}\wedge\ \mathit{turn} = 2 \wedge \mathit{turn}' = 1 \wedge \mathit{free} \in 0 \,..\, 1 \wedge \mathit{free\_x} \in 0 \,..\, 18\\ \wedge\ \mathit{free\_y} \in 0 \,..\, 18 \wedge \mathit{occ} \in 1 \,..\, 3 \wedge \mathit{occ}' \in 1 \,..\, 3\\ \wedge\ \mathit{pos\_x} \in 1 \,..\, 15 \wedge \mathit{pos\_x}' \in 1 \,..\, 15 \wedge \mathit{pos\_y} \in 1 \,..\, 15\\ \wedge\ \mathit{pos\_y}' \in 1 \,..\, 15 \wedge \mathit{req} \in 0 \,..\, 1 \wedge \mathit{req}' \in 0 \,..\, 1\\ \wedge\ \mathit{spot\_1} \in 0 \,..\, 1 \wedge \mathit{spot\_2} \in 0 \,..\, 1\\ \wedge\ \ \vee\ \ \wedge (\mathit{free} = 1) \wedge (\mathit{free\_x} \in 0 \,..\, 1) \wedge (\mathit{occ} = 3)\\ \qquad\quad \wedge (\mathit{occ}' = 3) \wedge (\mathit{pos\_x}' = 1) \wedge (\mathit{pos\_y}' = 1)\\ \qquad \vee\ \ \wedge (\mathit{free} = 1) \wedge (\mathit{free\_x} \in 2 \,..\, 18) \wedge (\mathit{occ} \in 2 \,..\, 3)\\ \qquad\quad \wedge (\mathit{occ}' = 3) \wedge (\mathit{pos\_x}' = 2) \wedge (\mathit{pos\_y}' = 1)\\ \qquad \vee\ \ \wedge (\mathit{free} = 1) \wedge (\mathit{occ} \in 1 \,..\, 2) \wedge (\mathit{occ}' = 1)\\ \qquad\quad \wedge (\mathit{pos\_x}' = 2) \wedge (\mathit{pos\_y}' = 1) \wedge (\mathit{spot\_2} = 0)\\ \qquad \vee\ \ \wedge (\mathit{free} = 1) \wedge (\mathit{occ} \in 1 \,..\, 2) \wedge (\mathit{occ}' = 2)\\ \qquad\quad \wedge (\mathit{pos\_x}' = 1) \wedge (\mathit{pos\_y}' = 1) \wedge (\mathit{spot\_1} = 0)\\ \qquad \vee (\mathit{occ} = 1) \wedge (\mathit{occ}' = 1) \wedge (\mathit{req} = 0)\\ \qquad \vee (\mathit{occ} = 1) \wedge (\mathit{occ}' = 1) \wedge (\mathit{req}' = 1)\\ \qquad \vee (\mathit{occ} = 2) \wedge (\mathit{occ}' = 2) \wedge (\mathit{req} = 0)\\ \qquad \vee (\mathit{occ} = 2) \wedge (\mathit{occ}' = 2) \wedge (\mathit{req}' = 1)\\ \qquad \vee (\mathit{occ} = 3) \wedge (\mathit{occ}' = 3) \wedge (\mathit{req} = 0)\\ \qquad \vee (\mathit{occ} = 3) \wedge (\mathit{occ}' = 3) \wedge (\mathit{req}' = 1)\\ \wedge\ \mathit{InvH}\end{array}$$

After hiding the robot's coordinates pos_x, pos_y, the environment action is simplified to

$$\begin{array}{l}\wedge\ \mathit{turn} = 2 \wedge \mathit{turn}' = 1 \wedge \mathit{free} \in 0 \,..\, 1\\ \wedge\ \mathit{free\_x} \in 0 \,..\, 18 \wedge \mathit{free\_y} \in 0 \,..\, 18\\ \wedge\ \mathit{occ} \in 1 \,..\, 3 \wedge \mathit{occ}' \in 1 \,..\, 3\\ \wedge\ \mathit{req} \in 0 \,..\, 1 \wedge \mathit{req}' \in 0 \,..\, 1\\ \wedge\ \mathit{spot\_1} \in 0 \,..\, 1 \wedge \mathit{spot\_2} \in 0 \,..\, 1\\ \wedge\ \ \vee (\mathit{free} = 1) \wedge (\mathit{occ} = 1) \wedge (\mathit{occ}' = 1)\\ \qquad \vee (\mathit{free} = 1) \wedge (\mathit{occ} = 3) \wedge (\mathit{occ}' = 3)\\ \qquad \vee\ \wedge (\mathit{free} = 1) \wedge (\mathit{occ} \in 1 \,..\, 2) \wedge (\mathit{occ}' = 2)\\ \qquad\quad \wedge (\mathit{spot\_1} = 0)\\ \qquad \vee (\mathit{occ} = 1) \wedge (\mathit{occ}' = 1) \wedge (\mathit{req} = 0)\\ \qquad \vee (\mathit{occ} = 1) \wedge (\mathit{occ}' = 1) \wedge (\mathit{req}' = 1)\\ \qquad \vee (\mathit{occ} = 2) \wedge (\mathit{occ}' = 2) \wedge (\mathit{req} = 0)\\ \qquad \vee (\mathit{occ} = 2) \wedge (\mathit{occ}' = 2) \wedge (\mathit{req}' = 1)\\ \qquad \vee (\mathit{occ} = 3) \wedge (\mathit{occ}' = 3) \wedge (\mathit{req} = 0)\\ \qquad \vee (\mathit{occ} = 3) \wedge (\mathit{occ}' = 3) \wedge (\mathit{req}' = 1)\\ \wedge\ \mathit{InvH}\end{array}$$

Details about safe positioning of the robot have been simplified, because they are not necessary information for the station's operation. These expressions have been obtained by using the invariant as a care predicate for the minimal covering problem that yields the DNF. In particular

$$\begin{array}{l}\mathit{InvH} \triangleq\\ \quad \wedge\ \ \mathit{turn} \in 1 \,..\, 2 \wedge \mathit{free} \in 0 \,..\, 1\\ \quad \wedge\ \ \mathit{free\_x} \in 0 \,..\, 18 \wedge \mathit{free\_y} \in 0 \,..\, 18\\ \quad \wedge\ \ \mathit{occ} \in 1 \,..\, 3 \wedge \mathit{spot\_1} \in 0 \,..\, 1 \wedge \mathit{spot\_2} \in 0 \,..\, 1\\ \quad \wedge\ \ \vee\ \ \wedge (\mathit{free\_x} = 1) \wedge (\mathit{free\_y} = 1) \wedge (\mathit{occ} \in 2 \,..\, 3)\\ \qquad\qquad \wedge (\mathit{spot\_1} = 0) \wedge (\mathit{spot\_2} = 1)\\ \qquad\ \ \vee\ \ \wedge (\mathit{free\_x} = 2) \wedge (\mathit{free\_y} = 1) \wedge (\mathit{occ} = 1)\\ \qquad\qquad \wedge (\mathit{spot\_1} = 1) \wedge (\mathit{spot\_2} = 0)\\ \qquad\ \ \vee\ \ \wedge (\mathit{free\_x} \in 1 \,..\, 2) \wedge (\mathit{free\_y} = 1) \wedge (\mathit{occ} = 3)\\ \qquad\qquad \wedge (\mathit{spot\_1} = 0) \wedge (\mathit{spot\_2} = 0)\\ \qquad\ \ \vee (\mathit{free} = 0)\\ \qquad\ \ \vee\ \wedge (\mathit{free\_x} = 2) \wedge (\mathit{free\_y} = 1) \wedge (\mathit{occ} = 3)\\ \qquad\qquad \wedge (\mathit{spot\_2} = 0)\end{array}$$

The concept of a care predicate will be described in Sec. VII. □

D. Choosing which variables to hide

Which variables can we hide without sacrificing realizability? We could enumerate combinations of variables to hide, and check realizability for each one. This is inefficient (there are exponentially many combinations to enumerate). Instead, we *parametrize* which variables are hidden or not. We redo Sec. V-C, but now the choice of hidden variables is *parametric*. For each variable, a *mask* constant m is introduced that "routes" the variable to take a visible or hidden value

$$\mathit{Mask}(m, v, h) \triangleq \text{IF } (m = \text{TRUE}) \text{ THEN } h \text{ ELSE } v$$

The rigid variable m models the availability or lack of information. Following Sec. V-C, we replace h with the selector expression to define a controllable step operator with parametrized hiding as follows (where variable v is h for the case that $m = \text{FALSE}$, meaning h visible)

```
$MaskedInv(h, v, x, y, m) \triangleq$ LET $r \triangleq Mask(m, v, h)$
                                        IN  $Inv(r, x, y)$
$PrmInv(v, x, y, m) \triangleq \exists h : MaskedInv(h, v, x, y, m)$
$R(v, x, y, m) \triangleq$
    $\exists y' : \forall x', v' : \forall h :$
        LET $r \triangleq Mask(m, v, h)$
        IN  $\lor \neg Inv(r, x, y)$
            $\lor \land SysNext(r, x, y, y')$
              $\land \lor \neg EnvNext(r, x, y, v', x')$
                $\lor Target(v', x', y', m)$
$PrmStep(v, x, y, m) \triangleq$
    $\land$  $PrmInv(v, x, y, m)$
    $\land$  $R(v, x, y, m)$
```

An important point is that we can "push" the substitution inwards, to obtain a controllable step operator over parametrized actions

```
$PrmStep(v, x, y, m) \equiv$
  LET
    $MskInv(h) \triangleq$
        LET $r \triangleq Mask(m, v, h)$
        IN  $Inv(r, x, y)$
    $PrmInv \triangleq \exists h : MskInv(h)$

    $MskSysNext(h, y') \triangleq$
        LET $r \triangleq Mask(m, v, h)$
        IN  $SysNext(r, x, y, y')$
    $PrmSysNext(y') \triangleq$
        $\land PrmInv$
        $\land \forall h : MskInv(h) \Rightarrow MskSysNext(h, y')$

    $MskEnvNext(h, v', x') \triangleq$
        LET $r \triangleq Mask(m, v, h)$
        IN  $EnvNext(r, x, y, v', x')$
    $PrmEnvNext(v', x') \triangleq$
        $\exists h : MskInv(h) \land MskEnvNext(h, v', x')$
  IN
    $PrmStep(v, x, y, m) \triangleq \exists y' : \forall x', v' :$
        $\land$  $PrmSysNext(y')$
        $\land$  $PrmEnvNext(v', x') \Rightarrow Target(v', x', y', m)$
```

The LET expressions can be implemented either with syntactic substitution of bitvector formulas (provided the variables v and h can take the same values, and compatible type hints are declared for them to aid bitblasting), or existential quantification. We use existential quantification. The operator $\mathit{PrmStep}$ can be rearranged to obtain an equivalent result with new actions and the full information Step, as in Sec. V-C. The assumption that Target does not depend on v', which enables the rewriting, holds only for $m = \text{TRUE}$, so this rewriting takes place for specific variables, after the parametrization has been used to select what variables to hide, as described in Sec. VI.

The parametrization is separate for each component. Fresh mask constants are declared for this purpose. These masks increase the number of Boolean-valued variables in the symbolic computation, but are not quantified during controllable step operations, and are Boolean-valued, whereas the variables they mask are integer-valued. With n components and k (integer-valued) variables in total (over all components), $(n-1)k$ Boolean mask variables are introduced. These are parameters, so the number of reachable states remains unchanged, and thus the same number of controllable step operations will be applied, and realizability fixpoints take the same number of iterations, similar to arguments developed for parametrized synthesis [76]. The number of components n involved in each individual decomposition step is expected to not be large, so that the design specification be understandable by a human.

The masks parametrize the interconnection architecture between components, and allow for computing symbolically those architectures that allow for decomposing the high-level specification into a contract. We can think of the above scheme as a *sensitivity analysis* of the problem with respect to the information available to different components.

E. Eliminating hidden variables

In Sec. V-C we defined the operator $\mathit{SimplerEnvNext}$ as

$$\mathit{SimplerEnvNext}(x, y, x') \triangleq$$
$$\exists h, h' : \mathit{Inv}(h, x, y) \land \mathit{EnvNext}(h, x, y, h', x')$$

We then mentioned some DNF expressions that are equivalent to this operator when defined in the context of a particular example. The DNF expressions can be defined as operators, but using a different identifier than $\mathit{SimplerEnvNext}$. The resulting formulas are provably equivalent, but one contains quantified variables, the other not. The quantifiers should be eliminated in order to obtain the component specifications.

These observations arise because we cannot define the same operator twice. A nullary operator stands for the expression on the right hand side of its definition [10, p.319]. For example, the definition $f \triangleq x^2$ defines the nullary operator f to be the expression x^2. We may define $g \triangleq x \times x$ and prove that $\models (x \in \mathit{Nat}) \Rightarrow (f = g)$ under the usual definitions of superscript and $\times$, but f and g are defined to be different expressions. The act of defining symbols, and how this act relates to declaring symbols as constants and introducing axioms about those symbols can be understood as extending a formal theory by definitions [107, §74, Vol.1, p.405].

VI. Decomposing a System into a Contract

A. Overview

The decomposition algorithm takes an (open or closed) system specification and produces open-system specifications for designated components. A component means a collection of variables. We assume that the specification allows components to stutter when variables from other components change. So component interaction is synchronous, in that nonstuttering environment steps are noticed by the component implementation, but the components are not required to react immediately to changes. This assumption is useful for transitions between interconnection architectures (Sec. VI-G3).

We describe the algorithm incrementally, starting with the main idea. The first description neglects hidden variables and complicated cases. We then add these details to obtain the algorithm's skeleton. The main idea is reasoning backwards about goals to create a chain of dependencies of which component is going to wait until which other component does what. These obligations can be sketched roughly as follows

$$\begin{array}{lrcl} \text{Component 1}: & L_1 & \triangleq & \Box\Diamond R_1 \\ \text{Component 2}: & L_2 & \triangleq & \Diamond\Box P_2 \vee \Box\Diamond R_2 \\ \text{Component 3}: & L_3 & \triangleq & \Diamond\Box P_3 \vee \Box\Diamond R_3, \end{array}$$

where the chaining is established by the implications

$$(R_1 \Rightarrow \neg P_2) \wedge (R_2 \Rightarrow \neg P_3).$$

Conjoining the above specifications, we can deduce the recurrence properties

$$L \;\triangleq\; \Box\Diamond R_1 \wedge \Box\Diamond R_2 \wedge \Box\Diamond R_3.$$

Each liveness property listed above should be ensured by the designated component implementation. So property L_1 is a guarantee from the perspective of component 1, and an assumption from the perspective of component 2. From the perspective of component 3, property L_2 is an assumption, and property L_3 is a guarantee.

There is no notion of a "liveness assumption" in the context of a single component specification. Viewing liveness only as a "guarantee" agrees with real world practice: there is no point in a behavior where we can decide that the liveness assumption "has been violated" [70], [110]. Liveness "assumptions" are meaningful in the context of multiple components, specified by multiple temporal properties, a situation similar to possibility properties [115], [10, §8.9.3]. The liveness part of a property defined by the *Unzip* and *WhilePlusHalf* operators has no distinct place that could be regarded as "assumption" (notably, if G is a safety property, then $F \overset{+}{\rightarrowtriangle} G$ is a safety property [48, §5.2, p.261]).

A simple but necessary property of the specifications L_1, L_2, L_3 is the acyclic arrangement of the reasoning that derives L [38], forming a *proof lattice* [44]. Mutual dependence of safety properties is admissible because it is possible to spread implication in a "stepwise" fashion over a behavior, as in the operator *WhilePlusHalf*. So what appears circular for safety properties is a well-founded chain of implications crisscrossing between components. Unlike safety properties, liveness properties allow arbitrary deferment of obligations to the future. This deferment is what allows circularity to arise when liveness properties are mutually dependent. Thus, in order to obtain sound conclusions about liveness properties of an assembly, there should be no cycles of dependence among liveness properties guaranteed by different components [49].

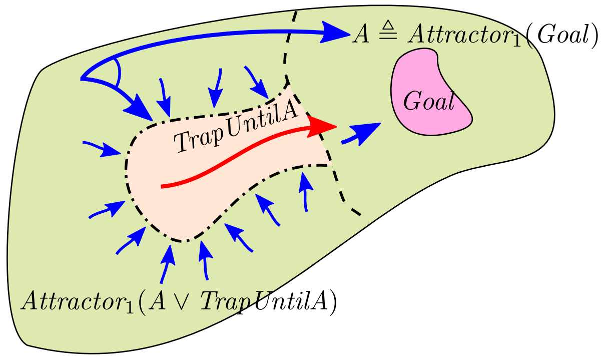

$A \triangleq \mathit{Attractor}_1(\mathit{Goal})$
Goal
TrapUntilA
$\mathit{Attractor}_1(A \vee \mathit{TrapUntilA})$

Fig. 5: The basic idea of the approach.

All the discussion that follows focuses on liveness and omits the safety part of specifications. Safety is addressed by closure and computation of component actions as described in Sec. V. In the computations, safety is taken into account in the *Step* operator within the *Attractor* and *Trap* operators.

B. The basic algorithm

Consider two components 1 and 2. Suppose that we want their assembly to satisfy the property $L \triangleq \Box\Diamond \mathit{Goal}$. We want to find liveness properties L_1, L_2 for each component that are realizable and conjoined imply L. If L is realizable by component 1 alone, then we can let L_1 be L and L_2 be TRUE. The interesting case is when accomplishing L requires interaction between components. The basic idea is shown in Fig. 5. For the objective *Goal*, the set

$$A \;\triangleq\; \mathit{Attractor}_1(\mathit{Goal})$$

contains those states from where component 1 can controllably lead the assembly to the *Goal*. Component 1 cannot ensure that *Goal* will be reached from outside A. So we need to relax the requirement $\Box\Diamond \mathit{Goal}$ on component 1, by disjoining another liveness property. Suppose that we could find a set *TrapUntilA* from where component 2 can reach A *and* component 1 can keep the assembly inside *TrapUntilA* until A is reached. We can then write the liveness specifications

$$\begin{array}{rcl} L_2 & \triangleq & \Box\Diamond\neg \mathit{TrapUntilA} \\ L_1 & \triangleq & \Diamond\Box\, \mathit{TrapUntilA} \vee \Box\Diamond \mathit{Goal} \end{array}$$

Property L_2 is realizable by component 2 (because it can reach A, which is outside *TrapUntilA*). If the set

$$C \;\triangleq\; \mathit{Attractor}_1(A \vee \mathit{TrapUntilA})$$

covers all of the assembly's initial conditions, then the property L_1 is realizable by component 1 from these initial conditions. Realizability ensures that L_1 and L_2 are implementable. Assembling the implementations specified by L_1 and L_2, we can deduce that the assembly satisfies $L_1 \wedge L_2$, and by

$$L_1 \wedge L_2 \;\Rightarrow\; \Box\Diamond \mathit{Goal}$$

the assembly will operate as desired.

We could have simply found $\mathit{Attractor}_2(A)$ (from where component 2 can lead the assembly to A), and continued

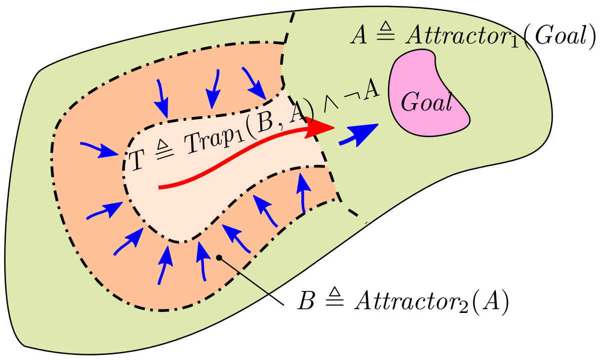

$A \triangleq Attractor_1(Goal)$
$T \triangleq Trap_1(B, A) \wedge \neg A$
Goal
$B \triangleq Attractor_2(A)$

Fig. 6: How traps are constructed (simple case).

alternating among players, until a fixpoint is reached. The resulting specifications would be chains of nested implications between recurrence goals, so not in GR(1) [106]. The construction described can be regarded as *subtracting* goals from each other, in order to avoid nested dependency.

We did not say how traps are computed, which we do now. Two attributes characterize a trap:

- Component 2 should be able to ensure that the behavior reaches A.
- Component 1 should be able to ensure that the behavior remains within the trap until A is reached.

The largest set that satisfies the first attribute is the attractor

$$B \quad \triangleq \quad Attractor_2(A).$$

The trap should be a subset of B. The largest subset of B that satisfies the second attribute can be computed as the greatest fixpoint

$$C \quad \triangleq \quad Trap_1(B, A).$$

The above is a shorthand for the trap operator defined in Sec. III-D, with B corresponding to *Stay* and A to *Escape*. The subscript 1 signifies that component 1 is existentially quantified within the controllable step operator. By definition of a trap,

$$(C \Rightarrow B) \wedge (A \Rightarrow C).$$

So the desired trap set is

$$T \quad \triangleq \quad C \wedge \neg A.$$

These sets are illustrated in Fig. 6. Letting $TrapUntilA \triangleq T$, we obtain realizable properties L_1, L_2 (the full specifications include safety, initial conditions, and are defined using *Unzip*, but this section focuses on the liveness parts).

The algorithm we described derives from an earlier version for the case without hidden variables [106], [116]. Covering all initial conditions of the assembly is not possible in general [106, §III-B], unless either safety is restricted [117, §V], or a syntactic fragment larger than GR(1) is used [106, §IV-A], which is equivalent to using auxiliary variables hidden by temporal quantification. Although possible, in this paper we do not apply any of these modifications to the specification, because the cases that require them [106, Prop.6] indicate that it is better for the specifier to reconsider the specification.

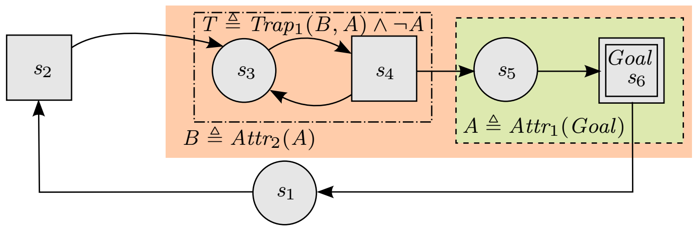

$T \triangleq Trap_1(B, A) \wedge \neg A$
s_2
s_3
s_4
s_5
Goal
s_6
$B \triangleq Attr_2(A)$
$A \triangleq Attr_1(Goal)$
s_1

Fig. 7: An example where a trap is found.

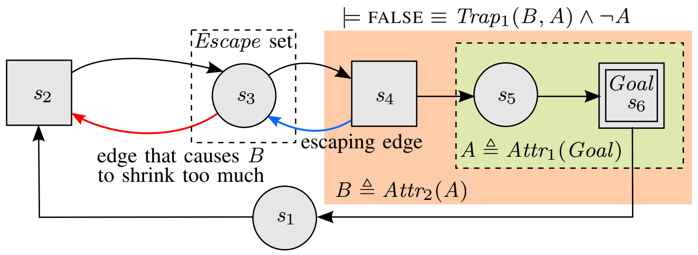

$\models$ FALSE $\equiv Trap_1(B, A) \wedge \neg A$
Escape set
s_2
s_3
s_4
s_5
Goal
s_6
edge that causes B to shrink too much
escaping edge
$A \triangleq Attr_1(Goal)$
$B \triangleq Attr_2(A)$
s_1

Fig. 8: The simple approach cannot find a trap in this example. Compared to Fig. 7, the failure is due to the edge $\langle s_4, s_3 \rangle$.

C. *Finding assumptions in more cases*

Forming a trap is the key step for constructing liveness assumptions. But the approach of Sec. VI-B can fail to find a trap, even in cases when our intuition suggests otherwise. The reason is too small a set B causing $Trap_1(B, A)$ to be empty. We use an example to explain why, and then a solution.

Example 5: Consider the graph shown in Fig. 7. Component 1 chooses the next node when at a disk, and component 2 when at a box. A trap is found for Fig. 7, because component 2 can reach A from both nodes s_3 and s_4. Fig. 8 shows a modification with the edge $\langle s_3, s_2 \rangle$ added. No trap is found in this case, because $B \wedge \neg A$ contains only node s_4, so component 2 can move "backwards" from s_4 to s_3. So a larger B is needed, but why did B shrink compared to Fig. 7?

The set B shrunk because component 2 can no longer reach A from node s_3. Nevertheless, this inability is irrelevant in the context of constructing a persistence goal for component 1. While pursuing the persistence goal T that we are about to construct, component 1 is not going to move backwards (s_3 to s_2), because it would interrupt its attempt to remain forever within T. It is this behavior that the specifier's intuition suggests. But component 2 is unaware of this premise, and neither can it depend on what component 1 will do, in order to avoid circularity (remember that we are discussing about liveness properties).

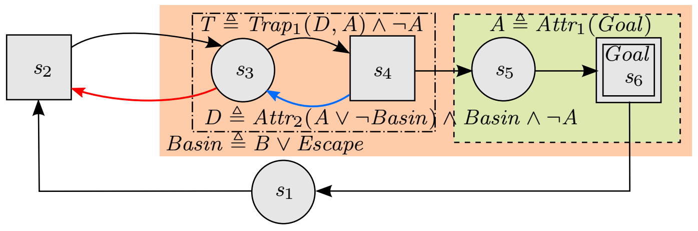
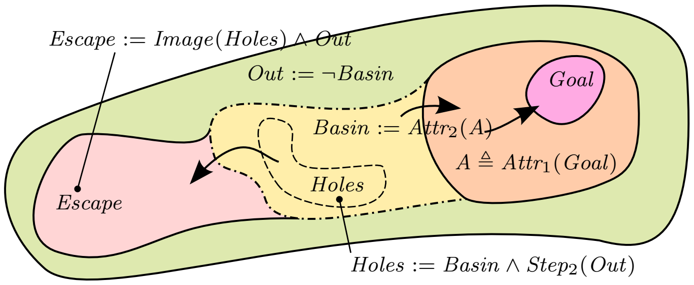

$T \triangleq Trap_1(D, A) \wedge \neg A$
$A \triangleq Attr_1(Goal)$
s_2
s_3
s_4
s_5
Goal
s_6
$D \triangleq Attr_2(A \vee \neg Basin) \wedge Basin \wedge \neg A$
$Basin \triangleq B \vee Escape$
s_1

Fig. 9: Including the states where component 2 can escape allows finding the trap suggested by the specifier's intuition.



$Escape := Image(Holes) \wedge Out$
$Out := \neg Basin$
$Goal$
$Basin := Attr_2(A)$
$A \triangleq Attr_1(Goal)$
$Holes$
$Escape$
$Holes := Basin \wedge Step_2(Out)$

Fig. 10: Collecting escapes that can cause a trap set to not form.

Enlarging B by the successors of states from where component 2 can "escape" out of B can avoid the issue described above. The result is shown in Fig. 9. Let the state predicate Escape mean that the current node is s_3. Define

$$\mathit{Basin} \;\triangleq\; B \vee \mathit{Escape}$$

We seek a trap within Basin, so a trap T that satisfies the implication

$$T \Rightarrow \mathit{Basin}$$

If component 1 can escape outside of Basin, then it can escape outside T too, by the contrapositive

$$(\neg \mathit{Basin}) \Rightarrow \neg T$$

Thus, there is no loss in relaxing the goal A to $A \vee \neg \mathit{Basin}$ for component 2. The corresponding attractor is

$$D \;\triangleq\; \mathit{Attr}_2(A \vee \neg \mathit{Basin}) \wedge \neg(A \vee \neg \mathit{Basin})$$

and accounts for the intent of component 1 to remain forever inside the trap T that is *about to be* computed. This relaxation of objective is shown in Fig. 9. The larger attractor D enables a trap to form; the set of states

$$T \;\triangleq\; \mathit{Trap}_1(D, A) \wedge \neg A$$

is nonempty. Moreover, $\mathit{Attr}_1(T \vee A)$ covers all nodes. So the components can realize the properties

$$\begin{array}{lll} \text{Component 2}: & L_2 \;\triangleq\; \Box\Diamond\neg D \\ \text{Component 1}: & L_1 \;\triangleq\; \Diamond\Box T \vee \Box\Diamond \mathit{Goal} \end{array}$$

Instead of an empty trap, we obtained a contract, because $T \Rightarrow D$, so $L_2 \Rightarrow \Box\Diamond\neg T$. The issue discussed in this section does arise in practice; for instance in the landing gear example of Sec. VIII. □

The above discussion referred to individual states. A symbolic approach relies on manipulating collections of states. Fig. 10 illustrates how what we described above is symbolically implemented. The Basin is initialized as (the symbol $:=$ indicates that the identifier Basin is going to change value during the algorithm's execution, in later sections)

$$\mathit{Basin} \;:=\; \mathit{Attr}_2(A).$$

The states from where component 2 can force a step that exits the Basin are those in the set

$$\mathit{Holes} \;:=\; \mathit{Basin} \wedge \mathit{Step}_2(\neg \mathit{Basin}).$$

Steps from Holes to the exterior of Basin lead to the set

$$\mathit{Escape} \;:=\; \mathit{Out} \wedge \mathit{Image}(\mathit{Holes})$$

where Image is the existential image operator (all unprimed flexible variables are existentially quantified), defined as

$$\begin{array}{l} \mathit{Image}(x,\, y,\, \mathit{Source}(\_,\, \_),\, \mathit{Next}(\_,\, \_,\, \_,\, \_)) \;\triangleq \\ \quad \exists\, p,\, q:\; \mathit{Source}(p,\, q) \wedge \mathit{Next}(p,\, q,\, x,\, y) \end{array}$$

The resulting Basin is used for computing $D := \mathit{Attr}_2(A \vee \neg \mathit{Basin}) \wedge \mathit{Basin} \wedge \neg A$ and $\mathit{Trap}_1(D, A) \wedge \neg A$. If the latter is nonempty, then we have found a trap. Otherwise, the above computation is iterated using the larger Basin as described in the following sections.

D. Taking observability into account

So far we ignored that each component observes different information. What information is available depends on the parameter values (Sec. V). Each component specification should be expressed using only variables that it observes, which is not the case in previous sections. In order to ensure this property, we use the following operators

$$\begin{array}{l} \mathit{Maybe}(v,\, x,\, y,\, m,\, P(\_,\, \_,\, \_)) \;\triangleq \\ \quad \exists\, h:\; \text{LET } r \;\triangleq\; \mathit{Mask}(m,\, v,\, h) \\ \quad\quad\quad \text{IN } \;\; P(r,\, x,\, y) \end{array}$$

$$\begin{array}{l} \mathit{Observable}(v,\, x,\, y,\, m,\, P(\_,\, \_,\, \_), \\ \quad\quad\quad\quad R(\_,\, \_,\, \_),\, \mathit{Inv}(\_,\, \_,\, \_)) \;\triangleq \\ \;\wedge\, \mathit{Maybe}(v,\, x,\, y,\, m,\, \mathit{Inv}) \\ \;\wedge\, \forall\, h: \text{LET } r \;\triangleq\; \mathit{Masks}(m,\, v,\, h) \\ \quad\quad\quad \text{IN } \;\; R(r,\, x,\, y) \Rightarrow P(r,\, x,\, y) \quad \text{P is observable within R} \end{array}$$

Some operator arguments are omitted in the discussion below. Expressing specification objectives using only visible variables allows for using the Step operator with suitably parametrized component and environment actions (Sec. V-C). Thus, we can apply the $\mathit{Attractor}$ and Trap operators. The sets of states when observability is taken into account are shown in Fig. 11. The indices correspond to components, with the mask parameters that correspond to each of them. The main difference with Sec. VI-C is that observability is required when alternating between components. Specifically,

- Goal is replaced by $G \triangleq \mathit{Obs}_1(\mathit{Goal})$ for computing A
- A is replaced by $U \triangleq \mathit{Obs}_2(A)$ for computing D
- D is replaced by $\mathit{Stay} \triangleq \mathit{Obs}_1(D)$ for computing T.

The next theorem establishes the connection between these objectives of components 1 and 2. The theorem is stated without mentioning the parameters, but applies also to parametrized computations.

Theorem 3 (Soundness): ASSUME : The sets of states D and T are computed as in Fig. 11. PROVE : The property

$$P \;\triangleq\; \Box\Diamond\neg D$$

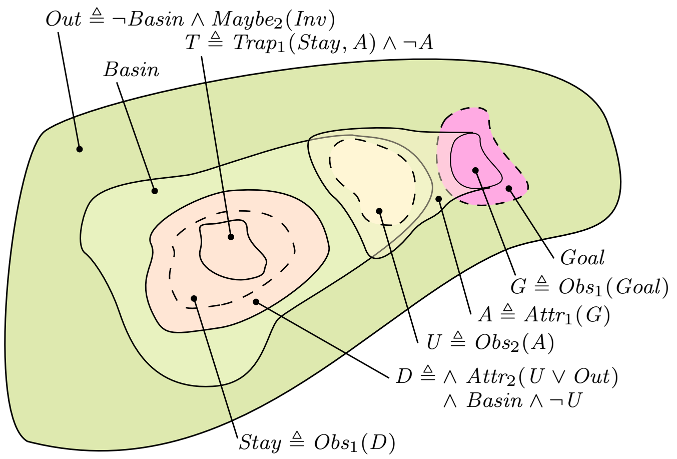
is realizable by component 2. The property

$$\begin{array}{l} Q \;\triangleq\; \Box \vee \neg(T \vee A) \\ \quad\quad \vee\, (\Diamond\Box T) \vee \Diamond A \end{array}$$

is realizable by component 1. The implication holds

$$\models T \Rightarrow D$$

A detailed proof can be found in the appendix.



$Out \triangleq \neg Basin \wedge Maybe_2(Inv)$
$T \triangleq Trap_1(Stay, A) \wedge \neg A$
$Basin$
$Goal$
$G \triangleq Obs_1(Goal)$
$A \triangleq Attr_1(G)$
$U \triangleq Obs_2(A)$
$D \triangleq \wedge\, Attr_2(U \vee Out)$ $\wedge\, Basin \wedge \neg U$
$Stay \triangleq Obs_1(D)$

Fig. 11: Accounting for observability when computing assumptions.

PROOF SKETCH: By its definition, D is contained in $Basin \wedge \neg U$, so $(U \vee Out) \Rightarrow \neg D$. States in D are contained in the attractor of $U \vee Out$, so the property $\Box(D \Rightarrow \Diamond(U \vee Out))$ is realizable by component 2. Thus, $\Box(D \Rightarrow \Diamond\neg D)$ is realizable by component 2, and this property is equivalent to $\Box\Diamond\neg D$. This proves the first claim.
From the trap $Z \triangleq Trap_1(Stay, A)$, component 1 can either eventually reach A or remain forever within $Z \wedge \neg A$, where $(Z \wedge \neg A) \equiv T$. By definition of T, it follows that $(T \vee A) \Rightarrow Z$. So from any state in $T \vee A$, component 1 can realize $\Diamond A \vee \Diamond\Box T$. This proves the second claim.
By definition of T, $T \Rightarrow Stay$. By definition of $Stay$, $Stay \Rightarrow D$. Thus, $T \Rightarrow D$. QED

Theorem 3 implies that component 1 cannot prevent component 2 from reaching $\neg D$. So it ensures that component 1 cannot stay forever within T, and that if the behavior exits T, then component 1 can ensure A is reached. As component 2 moves towards $\neg D$, the behavior does exit T. Therefore, progress of component 2 can be utilized by component 1 for progress towards its recurrence objective A. Theorem 3 is the building block for computing more complex dependencies of objectives. For a single recurrence goal of component 1, multiple traps may be needed to cover the desired set of states (for which we use the global invariant Inv). If the procedure MAKEPINFOASSUMPTION computes A, T, D, then by iterating this procedure until a least fixpoint is reached, we can find several traps, such that the corresponding persistence objectives suffice in order to eventually reach the $Goal$. This use is illustrated by the pseudocode

```
$Y := Observable_1(Goal)$
$Yold :=$ CHOOSE $r : r \neq Y$
while $Y \neq Yold$ :
  $Yold := Y$
  $A, T, D :=$ MAKEPINFOASSUMPTION$(Y, \dots)$
  (* ...store $D$ *)
  $Y := A \vee T$
```

The procedure MAKEPINFOASSUMPTION is defined in Sec. VI-F. This computation is in analogy to the least fixpoint computed for one goal in a GR(1) game [100].

Example 6: In the charging station example, for the recurrence goal $\Box\Diamond(req = 0)$ the trap that is computed when the robot can observe the variables $free, free\_x, turn$ is

$$\begin{array}{l}
T \;\triangleq \\
\wedge\; turn \in 1 \mathrel{..} 2 \wedge free \in 0 \mathrel{..} 1 \wedge free\_x \in 0 \mathrel{..} 18 \\
\wedge\; pos\_x \in 1 \mathrel{..} 15 \wedge pos\_y \in 1 \mathrel{..} 15 \wedge req \;\in 0 \mathrel{..} 1 \\
\wedge \quad \vee\, (turn = 1) \wedge (free\_x \in 1 \mathrel{..} 2) \wedge (req = 1) \\
\qquad \vee\, (free \;= 0) \wedge (req = 1)
\end{array}$$

The goal $\Diamond\Box T \vee \Box\Diamond(req = 0)$ can be understood as follows. The robot issues a request for recharging by setting $req = 1$. It cannot set $req = 0$ unless it has reached the position indicated as free by $free = 1$. The robot is allowed to wait while $free = 0$ (the station has not indicated any spot as available), or until $free = 1$ and the station has indicated an available spot, and it is not the robot's turn ($turn = 1$, not 2). The disjunct that involves $turn = 1$ appears in order to allow the charging station to satisfy the generated recurrence goal $\neg D$ (given below). If $turn = 1$ was absent from that disjunct, then the robot could raise a request ($req = 1$), and then simply ignore that the station did react by offering a spot ($free = 1$), and idle, without responding by reaching the spot, in order to be able to set $req = 0$. In other words, such a larger T would have relaxed the objective $\Diamond\Box T \vee \Box\Diamond(req = 0)$ too much.

The trap T corresponds to the recurrence objective $\Box\Diamond\neg D$ that is generated for the charging station provided it observes the variables $req, occ, turn$

$$\begin{array}{l}
D \;\triangleq \\
\wedge\; turn \in 1 \mathrel{..} 2 \wedge free \in 0 \mathrel{..} 1 \\
\wedge\; free\_x \in 0 \mathrel{..} 18 \wedge free\_y \in 0 \mathrel{..} 18 \wedge occ \in 1 \mathrel{..} 3 \\
\wedge\; req \in 0 \mathrel{..} 1 \wedge spot\_1 \in 0 \mathrel{..} 1 \wedge spot\_2 \in 0 \mathrel{..} 1 \\
\wedge\; req = 1 \\
\;\wedge \quad \vee \quad \wedge\, (turn = 1) \wedge (free\_x = 1) \wedge (free\_y = 1) \\
\qquad\qquad \wedge\, (occ \in 2 \mathrel{..} 3) \wedge (spot\_1 = 0) \wedge (spot\_2 = 1) \\
\qquad \vee \quad \wedge\, (turn = 1) \wedge (free\_x = 2) \wedge (free\_y = 1) \\
\qquad\qquad \wedge\, (occ = 1) \wedge (spot\_1 = 1) \wedge (spot\_2 = 0) \\
\qquad \vee \quad \wedge\, (turn = 1) \wedge (free\_x = 2) \wedge (free\_y = 1) \\
\qquad\qquad \wedge\, (occ = 3) \wedge (spot\_2 = 0) \\
\qquad \vee \quad \wedge\, (turn = 1) \wedge (free\_x \in 1 \mathrel{..} 2) \wedge (free\_y = 1) \\
\qquad\qquad \wedge\, (occ = 3) \wedge (spot\_1 = 0) \wedge (spot\_2 = 0) \\
\qquad \vee\, (free = 0)
\end{array}$$

This recurrence objective requires from the station to react by indicating some spot as free, and also make sure that the spot is not taken (as indicated by the variables $spot\_1, spot\_2, spot\_3$). The above expressions were computed from BDDs by using the approach described in Sec. VII, and the conjunct $req = 1$ was factored out of the disjuncts for brevity of the presentation. □

E. Multiple recurrence goals

The results of Sec. VI-D are about one recurrence goal. By repeating the computation for different recurrence goals, for example $\Box\Diamond R_1$ and $\Box\Diamond R_2$ for component 1, suitable realizable properties can be found, for example $\Diamond\Box P_1 \vee \Box\Diamond R_1$ and $\Diamond\Box P_2 \vee \Box\Diamond R_2$. However, conjoining these two properties would not yield a GR(1) property. Instead, a GR(1) property can be formed by a suitable combination described below, provided that $\Box\Diamond\neg P_1 \wedge \Box\Diamond\neg P_2$ are implemented by components that can realize them irrespective of how component 1 behaves (unconditionally).

Relaxing a property preserves realizability. More precisely, if a property P is realizable, and P implies Q, then Q is realizable.

Proposition 4 (Relaxation): ASSUME : TEMPORAL P, Q PROVE : $(\mathit{IsRealizable}(P) \land (P \Rightarrow Q)) \Rightarrow \mathit{IsRealizable}(Q)$.

For what we are interested in, let

$$\begin{aligned} L &\triangleq \land \Diamond\Box P_1 \lor \Box\Diamond R_1 \\ &\quad\ \land \Diamond\Box P_2 \lor \Box\Diamond R_2 \\ Q &\triangleq \lor \Diamond\Box P_1 \lor \Diamond\Box P_2 \\ &\quad\ \lor \Box\Diamond R_1 \land \Box\Diamond R_2 \end{aligned}$$

It is the case that $L \Rightarrow Q$, so if a component can realize L, then it can realize Q. The reverse direction does not hold in general. Nonetheless, if a behavior σ satisfies

$$\sigma \models \neg(\Diamond\Box P_1 \lor \Diamond\Box P_2)$$

and σ arises when using a component that implements Q, then it follows that $\sigma \models \Box\Diamond R_1 \land \Box\Diamond R_2$. This establishes the reverse direction in the presence of other components that implement $\Box\Diamond\neg P_1$ and $\Box\Diamond\neg P_2$. This reasoning leads to the following theorem.

Theorem 5: Let $Q \triangleq \lor \Diamond\Box T_1 \lor \Diamond\Box T_2$
$\qquad\qquad\qquad\lor \Box\Diamond R_1 \land \Box\Diamond R_2$

ASSUME : $\mathit{IsRealizable}_1(Q) \land \land \neg D_1 \Rightarrow \neg T_1$
$\qquad\qquad\qquad\qquad\qquad\land \neg D_2 \Rightarrow \neg T_2$

PROVE : $\forall f : \lor \neg \land \mathit{IsARealization}_1(f, Q)$
$\qquad\qquad\qquad\land \mathit{Realization}_1(f, Q)$
$\qquad\qquad\qquad\land \Box\Diamond\neg D_1 \land \Box\Diamond\neg D_2$
$\qquad\qquad\lor \Box\Diamond R_1 \land \Box\Diamond R_2$

where $\mathit{IsARealization}$ is the modification of $\mathit{IsRealizable}$ that results from making $f, g, y0, mem0$ arguments. To emphasize the main points, we have simplified the notation, lumping all these arguments as f, and letting the subscript 1 indicate the e being used for component 1. The discussion above generalizes to more than two recurrence properties in an analogous way.

F. Detecting solutions in the presence of parametrization

The implementation of the computation described in Sec. VI-D is shown in Algo. 12. The controllable step operator, fixpoint and other computations are parametrized with respect to the communication between the components, as described in Sec. V. The parameters are indexed by component and current recurrence goal, which is the purpose of passing Team and Player as arguments. Player corresponds to component 1 and Team to component 2 in earlier sections. The renaming is in anticipation of discussing the case of more than two components in Sec. VI-G2.

Iteratively enlarging the Basin does not necessarily lead to a monotonic behavior of the trap η_{player}. To see why, consider the effect of increasing Basin to the computation within the procedure MAKEPAIR, when T is empty. The $\mathit{TeamGoal}$ shrinks, so $\mathit{Attr}(\mathit{TeamGoal}, \mathit{Team})$ may shrink (not necessarily), but $\mathit{Basin} \land \neg\mathit{TeamGoal}$ becomes larger. Thus, D may become larger, leading to a larger Stay, thus possibly to a nonempty T. This is possible, but not necessarily the case. The largest Basin is TRUE, and corresponds to the basic case of Sec. VI-B, which can fail as demonstrated by Fig. 8. So enlarging the Basin after a trap forms can lead (back) to an empty trap.

To avoid regressing to an empty trap, as soon as a trap set is found, the iteration should terminate. In absence of parameters this is a straightforward check whether η_{player} is nonempty. However, this does not apply to parametrized computations. Each parameter valuation defines a "slice" of the state-parameter space, as shown in Fig. 13. A different number of iterations can be necessary for a trap to form in each slice. For this reason, as soon as a trap is found for some parameter values, those are "frozen" in further iterations, as illustrated in Fig. 14. The variable $\mathit{Coverged}$ is used for this purpose. The operator $\mathit{NonEmptySlices}(\eta_{player}) \triangleq \exists vars : \eta_{player}$ abstracts the variables of all players, in order to find the parameter values such that η_{player} is not FALSE. This approach ensures that traps are recorded when found, and that the iteration terminates.

Theorem 6 (Termination): ASSUME : A finite number of states satisfies the global invariant Inv. PROVE : Algo. 12 terminates in a finite number of iterations.
A structured proof style is used [118], [91].

```
⟨1⟩ $k \triangleq$ CHOOSE $n \in Nat$ : TRUE
⟨1⟩ SUFFICES $\lor\ Terminates(iter = k)$
              $\lor\ EnlargesStrictly(Basin,\ iter = k)$
    BY ASSUMPTION  Finitely many relevant states satisfy Basin.
⟨1⟩1.CASE $At(L1,\ iter = k) :\ \models Escape \equiv$ FALSE
    ⟨2⟩1. $Terminates(iter = k)$
        BY ⟨1⟩1, $WhileGuard$
    ⟨2⟩ QED
        BY ⟨2⟩1
⟨1⟩2.CASE $At(L1,\ iter = k+1) :\ \neg \models Escape \equiv$ FALSE)
    ⟨2⟩1. $At(L2,\ iter = k) :\ \models Out' \Rightarrow \neg Basin$
    ⟨2⟩2. $At(L3,\ iter = k) :\ \models Escape' \Rightarrow Out$
    ⟨2⟩3. $At(L3,\ iter = k) :\ \models Escape' \Rightarrow \neg Basin$
        BY ⟨2⟩1, ⟨2⟩2
    ⟨2⟩4. $At(L4,\ iter = k) :$
              $\models (Escape \land Basin)\ \equiv$ FALSE
        BY ⟨2⟩3
    ⟨2⟩5. $At(L4,\ iter = k) :\ \models Escape \Rightarrow Basin'$
    ⟨2⟩6. $At(L4,\ iter = k) :$
            $\neg \models (Basin' \land \neg Basin) \equiv$ FALSE
        BY ⟨1⟩2, ⟨2⟩4, ⟨2⟩5
    ⟨2⟩7. $At(L4,\ iter = k) :\ \models Basin \Rightarrow Basin'$
    ⟨2⟩8. $EnlargesStrictly(Basin,\ iter = k)$
        BY ⟨2⟩7
    ⟨2⟩ QED
        BY ⟨2⟩8
⟨1⟩ QED
    BY ⟨1⟩1, ⟨1⟩2
```

1) Characterizing the parametrization: Parameters are TLA$^+$ constants, also known as *rigid variables* [10]. Parametrization has a "static" effect: the controllable step operator quantifies over only (primed) *flexible* variables, so the number of quantified variables remains unchanged. Each "slice" obtained by assigning values to parameters has diameter (the farthest two states can be apart in number of

Algo. 12: Algorithm for constructing contracts of recurrence-persistence pairs.

```
  def $\textsc{MakePinfoAssumption}(\mathit{Goal}, \mathit{Player}, \mathit{Team})$ :
      (* The player can observe its own variables. *)
      (* Some team variables are hidden from the player, *)
      (* as determined by parameters. Vice versa for the team. *)
      (* So the parametrizations express different perspectives. *)
      $G := \mathit{Observable}(\mathit{Goal}, \mathit{Player})$
      $A := \mathit{Attr}(G, \mathit{Player})$
      $\mathit{TeamGoal} := \mathit{Observable}(A, \mathit{Inv}, \mathit{Inv}, \mathit{Team})$
      $\mathit{Basin} := \mathit{Attr}(\mathit{TeamGoal}, \mathit{Team})$
      $\mathit{Escape} := \mathrm{TRUE}$
      $\mathit{Converged} := \mathrm{FALSE}$
$\boxed{\mathrm{L1}}$  while $(\neg \models \mathit{Escape} \equiv \mathrm{FALSE})$ :
          (* Complement within team state space. *)
  $\boxed{\mathrm{L2}}$  $\mathit{Out} := \neg\mathit{Basin} \wedge \mathit{Maybe}(\mathit{Inv}, \mathit{Team})$
          $\mathit{Holes} := \mathit{Basin} \wedge \mathit{Step}(\mathit{Out}, \mathit{Team})$
          $\mathit{Escape} := \mathit{Out} \wedge \mathit{Image}(\mathit{Holes} \wedge \mathit{Inv}, \mathit{Team})$
  $\boxed{\mathrm{L3}}$  $\mathit{Escape} := \wedge\ \mathit{Maybe}(\mathit{Escape}, \mathit{Team})$
                     $\wedge\ \mathit{Out} \wedge \neg\mathit{Converged}$
  $\boxed{\mathrm{L4}}$  $\mathit{Basin} := \mathit{Basin} \vee \mathit{Escape}$
          $\eta_{player}, \eta_{team} := \textsc{MakePair}($
              $A, \mathit{Basin}, \mathit{Player}, \mathit{Team})$
          $\mathit{Converged} := \vee\ \mathit{Converged}$
                         $\vee\ \mathit{NonEmptySlices}(\eta_{player})$
      return $A, \eta_{player}, \eta_{team}$

  def $\textsc{MakePair}(A, \mathit{Basin}, \mathit{Player}, \mathit{Team})$ :
      $\mathit{TeamGoal} := \vee\ \mathit{Observable}(A, \mathit{Inv}, \mathit{Inv}, \mathit{Team})$
                     $\vee\ \neg\mathit{Basin} \wedge \mathit{Maybe}(\mathit{Inv}, \mathit{Team})$
      $D := \wedge\ \mathit{Attr}(\mathit{TeamGoal}, \mathit{Team})$
           $\wedge\ \mathit{Basin} \wedge \neg\mathit{TeamGoal}$
      $\mathit{Stay} := \mathit{Observable}(D, \mathit{Inv}, \mathit{Inv}, \mathit{Player})$
      $T := \mathit{Trap}(\mathit{Stay}, A, \mathit{Player}) \wedge \neg A$
      return $T, D$
```

transitions) no larger than the state space of the assembly without any parametrization.

So the number of iterations until reaching fixpoints in attractor and other computations is the same with and without parametrization (because the case of no hidden variables corresponds to a parameter valuation). Similar observations have been made for the case of parametrized reachability goals [76, pp. 69, 80].

One difference with parametrization of goal sets is that those can be encoded directly with existing game solvers (by letting the parameters be flexible variables constrained to remain unchanged [88, Note 16]), whereas the parametrization of information studied here requires using substitution (equivalently, rigid quantification) and quantification in order to hide the selected variables in the component actions (the resulting parametrized actions can still be used with the usual controllable step operator).

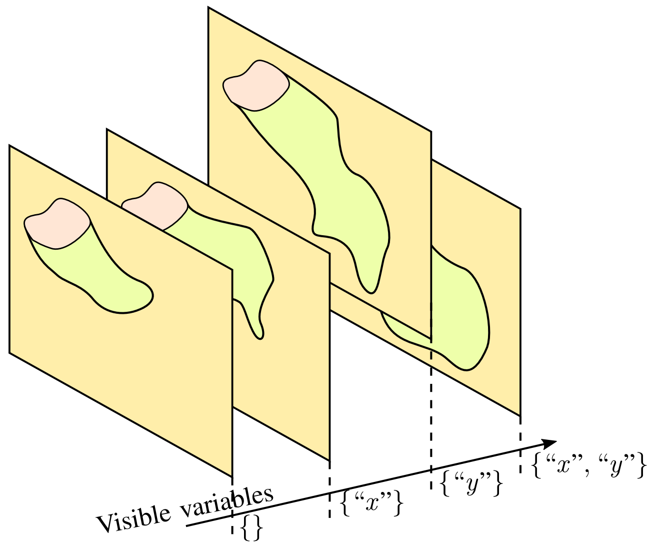


Fig. 13: Slices of the state space that correspond to different assignments of values to the parameters.

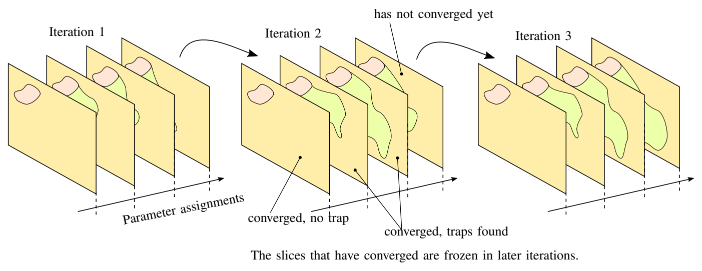


Fig. 14: In iterations of non-monotonic operators that depend on parameters, when a solution is found for some parameter values (a slice), then no further iterations should occur for those values.

G. Other considerations

1) Covering the global invariant: The selection of interconnection architecture (possibly different for each recurrence goal) is constrained to ensure that the "root" component (component 1 in the preceding sections) can realize its recurrence goals from all states that satisfy the global invariant *Inv*. The assumption that specifications do not force immediate component reactions ensures that when each recurrence goal is reached, a nonblocking step is possible in transition to pursuing the next recurrence goal. So if the states that satisfy the fixpoint Y in Sec. VI-D cover the invariant *Inv* from the viewpoint of the component, then the generated specification is realizable, in particular

$$\models \mathit{Maybe}_i(\mathit{Inv}) \Rightarrow Y,$$

where i indicates the component under consideration. This constraint is required in order to restrict the parameter values that constitute admissible solutions. An alternative formulation is possible, where an outermost greatest fixpoint is computed in order to find the largest set of states from where the root component can realize its goals, as a function of the parameters. Nonetheless, if this set of states does not cover the global invariant, this indicates that the assembly specification may need modification, in order to ensure that the assembled system can work from all states that it is expected to, based on the assembly specification before decomposition.

2) Systems with more than two components: By applying Theorem 3 hierarchically in an acyclic way, we can deduce properties of the assembled implementations from the component specifications. The previous sections were formulated in terms of two components. The same approach applies to multiple components, as follows. The components are partitioned into a "root" component, and the rest form a "team". The decomposition algorithm is applied to two players: the root component and the team. In this step, the team is treated as if it was a single player. The specification that is generated for the team needs to be decomposed further. This is achieved by applying the same algorithm recursively, using as goal the generated $\neg D$. In other words, what is generated as $\neg D$ for the team at the top layer becomes the Goal for one of the team's components at a lower layer of decomposition. Components are removed, until the team is reduced to a singleton. We will see an example of this kind with three components in Sec. VIII.

When the procedure MAKEPAIR of Algo. 12 is called for decomposing a subsystem, the set of states Stay can result smaller than intuition suggests. The reason is that when we write specifications by hand, we reason "locally", i.e., under the condition that we are constructing a specification for the team to reach Goal, so we implicitly condition our thinking in terms of $\neg \mathit{Goal} \wedge \mathit{Inv}$. This condition can be applied to the algorithm in order to improve observability. This modification is obtained by the replacement

$$\mathit{Stay} := \mathit{Observable}(D, \mathit{Within} \wedge \mathit{Inv}, \mathit{Inv}, \mathit{Player}),$$

where Within is the set of states within which the constructed objectives are needed. The trade-off is that the resulting persistence goal can "protrude" outside the set of states Within. What needs to be checked in that case is that the intersection of the persistence goal with $\neg \mathit{Within}$ is outside sets where other components depend on that component (for example, the root component), or otherwise subsumed by some other persistence goal of the same component.

3) Switching interconnection architecture: Different recurrence goals can be associated to different interconnection architectures between components. To progress towards each goal with the generated specifications, the system should switch between the different interconnections. This switching is controlled by the "root" component. In each interconnection mode, different variables are communicated between components. To model the switching between different interconnection architectures in TLA$^+$, we formalize the interface between each pair of components by using a variable that takes records as values. A record is a function with a set of strings as domain (a dictionary). For example, if x, z are variables that model component 1, then these are not declared as variables in the specification of component 2, because doing so would make them uninterruptedly visible to component 2. Instead, a record-valued variable $\mathit{vars}_1 \in [\text{SUBSET}\ \{\text{“x”}, \text{“z”}\} \rightarrow \mathit{Val}]$ models the communication channel from component 1 to component 2. In different interconnection modes, the variable vars_1 takes values with different domain, thus making different variables of component 1 visible to component 2.

In order to switch between interconnections, each interconnection should be signified in some way. There are two options for representing the interconnection mode. In the case that each component is connected to the root component, and each interconnection involves different collections of visible variables to each component, then the domain of the record itself encodes the interconnection mode. Otherwise, an extra field in the record is added to define the interconnection mode. In configurations that occur intermediately while transitioning from one interconnection to another, the components remain unblocked, because as assumed earlier the specification allows stuttering reactions.

The information available to each component is a prerequisite for realizability of its objectives. In the case of more than one interconnections, the goals of components are conditioned on the corresponding interconnection mode. In other words, persistence goals are conjoined to the interconnection mode they correspond, i.e., $\Diamond\Box(T \wedge (\mathit{cnct} = k))$, and recurrence goals are required only provided the corresponding interconnection mode is active, i.e., $\Box\Diamond((\mathit{cnct} = k) \Rightarrow \neg D)$. This can be regarded as a component assuming that if it provides enough information to its environment, then it can in return request reactions that become feasible for the environment when that information is available.

VII. GENERATING MINIMAL SPECIFICATIONS

We use binary decision diagrams [24], [119] for the symbolic computations described in previous sections. BDDs are typically used in symbolic model checking for verifying that a system has certain properties [26], [25], in synthesis of controllers [120], [121] (e.g., as circuits), and in electronic design automation [122], [123]. These applications are directed from user input to an answer of either a decision problem (yes/no), or some construct (e.g., a circuit) to be used without the need for a human to study its internal details. When more details are needed, for example if the input needs to be corrected, then in many cases the interaction between human and machine becomes enumerative, by listing counterexamples, satisfying assignments, and other witnesses that demonstrate the properties under inspection.

The BDDs in our approach represent specifications, so we want to read them. BDDs themselves are not a representation that humans can easily inspect and understand. For example, the global invariant of the charging station example was generated from the BDD shown in Fig. 15. A simple alternative would be to list the satisfying assignments for this BDD. However, there are 3.9 million satisfying assignments, so inspection of a listing would not be very helpful for understanding what predicate the BDD corresponds to. An additional difficulty is that we work with integer-valued variables, and these are represented using Boolean-valued variables ("bits") in the BDD. We are used to reading integers, not bitfields.

We are interested in representing the answer (a specification) in a readable way. A canonical form for representing Boolean

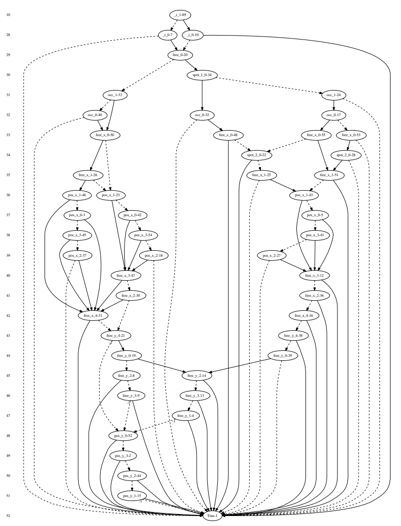


Fig. 15: The binary decision diagram from which the formula of the global invariant was generated for the charging station example in Sec. V-B. The variable names shown are the "bits" that are used to represent the integer-valued variables. A BDD isn't very suitable to help a human understand what Boolean expression it represents.

functions is in disjunctive normal form (DNF). Having to read less usually helps with understanding what a formula means, so we formulate the problem as that of finding a DNF formula with the minimal number of disjuncts necessary for representing a given Boolean predicate. The next question is how the disjuncts should be written. In the propositional case, each disjunct is a conjunction of Boolean-valued variables. We are interested in integer-valued variables, so we choose conjunctions of interval constraints of the form $x \in a..b$. In the context of circuit design the problem of finding a minimal DNF is known as two-level logic minimization [124], [125], [126], [27]. Logic minimization is useful for reducing the number of physical elements used to implement a circuit, thus the circuit's physical area. The problem of finding a minimal DNF for a given Boolean function can be formulated as a minimal set covering problem, and is NP-hard. Algorithms for logic minimization are typically based on a branch-and-bound search.

We implemented an exact minimal covering algorithm [27] that is based on a branch-and-bound search, together with symbolic computation of the essential prime implicants and cyclic core (primes that are neither essential nor dominated by other primes) during the search [127], [128], [129], [130]. The original algorithm was formulated for the general case of a finite (complete) lattice, and symbolically implemented for the case of the Boolean lattice. As remarked above, we use integer-valued variables, so we are interested in the lattice of integer hyperrectangles. The propositional minimal covering algorithm is not suitable for the case of integer variables, because the minimization is in terms of constraints on individual bits, ignoring the relation between the bits that are part of the same bitfield. This leads to awkward expressions that are difficult to understand. In other words, the "palette" of expressions available when working directly with bits is not easy to understand, as opposed to constraints of the form $x \in a..b$, where x is an integer-valued variable. For this purpose, we implemented the exact symbolic minimization method for the lattice of integer orthotopes (hypperrectangles aligned to axes). The implementation is available as part of the Python package `omega` [131]. Briefly, the problem of finding a minimal DNF formula of the form we described can be expressed as follows, where f is the Boolean function that is represented as a BDD and a formula is to be found. The *Domain* in our approach is a Cartesian product of integer variable ranges.

$\textsc{extends}\ \mathit{FiniteSets}, \mathit{Integers}$

$\textsc{constants}\ \mathit{Variables}, \mathit{Domain}, \mathit{CareSet}$

$\mathit{Assignments} \triangleq [\mathit{Variables} \rightarrow \mathit{Int}]$

$\textsc{assume}$

$\quad \land (\mathit{Domain} \subseteq \mathit{Assignments}) \land (\mathit{CareSet} \subseteq \mathit{Domain})$

$\quad \land \mathit{IsFiniteSet}(\mathit{Domain}) \land \mathit{IsFiniteSet}(\mathit{Variables})$

$\quad \land (\mathit{CareSet} \neq \{\}) \land f \in [\mathit{Domain} \rightarrow \textsc{boolean}]$

$\mathit{EndPoint}(k) \triangleq [1 \mathrel{..} k \rightarrow \mathit{Domain}]$

$\mathit{IsInOrthotope}(x, a, b) \triangleq \forall \mathit{var} \in \mathit{Variables} :$

$\quad (a[\mathit{var}] \leq x[\mathit{var}]) \land (x[\mathit{var}] \leq b[\mathit{var}])$

$\mathit{IsInRegion}(x, p, q) \triangleq \exists i \in \textsc{domain}\ p :$

$\quad \mathit{IsInOrthotope}(x, p[i], q[i])$

$\mathit{SameOver}(f, p, q, S) \triangleq \forall x \in S :$

$\quad f[x] \equiv \mathit{IsInRegion}(x, p, q)$

p, q define a cover that contains k orthotopes

$\mathit{IsMinDNF}(k, p, q, f) \triangleq$

$\quad \land \{p, q\} \subseteq \mathit{EndPoint}(k)$

$\quad \land \mathit{SameOver}(f, p, q, \mathit{CareSet})$

$\quad \land \forall r \in \mathit{Nat} : \ \forall u, v \in \mathit{EndPoint}(r) :$

$\qquad \lor \neg \mathit{SameOver}(f, u, v, \mathit{CareSet})$ not a cover, or

$\qquad \lor r \geq k$ u, v has at least as many disjuncts as p, q

A useful feature of the approach is the possibility of defining a *care predicate* (that defines a care set). A care set can be thought of as a condition to be taken as "given" by the algorithm when computing a minimal DNF. For example, consider the formula

$$\begin{array}{l} \lor\ (x \in 1..5) \land (y \in 3..4) \\ \lor\ (x \in 1..2) \land (z \in 1..3) \land (y \in 3..4) \end{array}$$

Using the care set defined by $\mathit{Care} \triangleq y \in 3..4$, the above formula can be simplified to

$$\begin{array}{l} \land \lor\ (x \in 1..5) \\ \quad \lor\ (x \in 1..2) \land (z \in 1..3) \\ \land\ y \in 3..4 \end{array}$$

This transformation is a form of factorization, where the care predicate is used as a given conjunct. When working with specifications, such factorization allows using other parts of the specification (e.g., an invariant), or other versions (e.g., a predicate before it is modified) to simplify the printed expressions.

Besides reading the final result of a symbolic computation, we have found the method of decompiling BDDs as minimal DNF formulas over integer-valued variables an indispensable

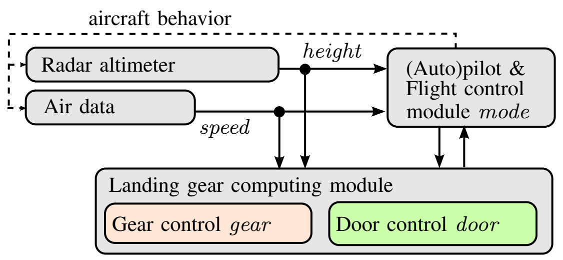


Fig. 16: Landing gear avionics.

aid during the *development* of symbolic algorithms. Symbolic operations are implicit: the developer cannot inspect the values of variables as readily as for enumerative algorithms. It is highly unlikely that any symbolic program works on first writing. Bugs will usually be present, and some debugging needed. Being able to print small expressions for the BDD values of variables in symbolic code has helped us considerably during development efforts. Another area of using the algorithm is for inspecting controllers synthesized from temporal logic specifications.

Example 7: We show the usefulness of decompiling BDDs by revisiting the charging station example from Sec. V-B. Fig. 15 shows the BDD that results from computing the invariant of the assembly in that example (the bits with names starting with $\_i$ encode the variable $turn$). This BDD was obtained after reordering the bits using a method known as sifting [132], whose purpose is to reduce the number of nodes in the BDD. Attempting to decipher what the BDD means is instructive, but not an efficient investment of time. By applying the minimal covering algorithm described above, we obtain the minimal DNF formula

$$\begin{array}{l}
\land\ turn \in 1\,..\,2 \land free \in 0\,..\,1\\
\land\ free\_x \in 0\,..\,18 \land free\_y \in 0\,..\,18 \land occ \in 1\,..\,3\\
\land\ pos\_x \in 1\,..\,15 \land pos\_y \in 1\,..\,15\\
\land\ spot\_1 \in 0\,..\,1 \ \land spot\_2 \in 0\,..\,1\\
\land \quad \lor \quad \land (free\_x = 1) \land (free\_y = 1) \land (occ \in 2\,..\,3)\\
\qquad\qquad \land (spot\_1 = 0) \land (spot\_2 = 1)\\
\qquad \lor \quad \land (free\_x = 2) \land (free\_y = 1) \land (occ = 1)\\
\qquad\qquad \land (spot\_1 = 1) \land (spot\_2 = 0)\\
\qquad \lor \quad \land (free\_x \in 1\,..\,2) \land (free\_y = 1) \land (occ = 3)\\
\qquad\qquad \land (spot\_1 = 0) \land (spot\_2 = 0)\\
\qquad \lor (free = 0)\\
\qquad \lor \land (free\_x = 2) \land (free\_y = 1) \land (occ = 3)\\
\qquad\quad \land (spot\_2 = 0)
\end{array}$$

That the minimization is performed directly for formulas over integer variables distinguishes this result from what a propositional approach would yield in terms of bitfields. □

VIII. EXAMPLE

The example we consider concerns the subsystems involved in controlling the landing gear of an aircraft [106], [133]. Three modules are involved, as shown in Fig. 16. The autopilot controls the altitude, flight speed, and mode of the aircraft. The gear module positions the landing gear, which can be extended, retracted, or in some transitory configuration. The third module operates the doors that seal the gear storage area. The variables take integer values, with appropriate units that can be ignored for our purpose here. We specify the following main properties collectively for these modules:

- If the gear is not retracted, then the doors shall be open.
- If airspeed is above $threshold\_speed$, then the doors shall be closed.
- If the aircraft is flying at or below $threshold\_height$, then the gear shall be fully extended.
- On ground the gear shall be fully extended.
- In landing mode the gear shall be fully extended.
- In cruise mode the gear shall be retracted and the doors closed.
- The autopilot shall be able to repeatedly enter the landing and cruise modes.

The specification of the assembled system is given below in TLA⁺.

$$\begin{array}{l}
\textsc{extends}\ Integers\\
\textsc{variables}\ mode,\ height,\ speed,\ door,\ gear,\ turn\\
\textsc{constants}\ max\_height,\ max\_speed,\ door\_down,\\
\qquad gear\_down,\ threshold\_height,\ threshold\_speed
\end{array}$$

mode 0, 2, 1 used below

$$\begin{array}{l}
Modes \triangleq \{\text{“landing”}, \text{“intermediate”}, \text{“cruise”}\}\\
Autopilot \triangleq \langle height,\ mode,\ speed\rangle\\
AutopilotTurn \triangleq turn = 1\\
DoorTurn \triangleq turn = 2\\
GearTurn \triangleq turn = 3\\
Init \triangleq \ \land (mode = \text{“landing”}) \land (height = 0)\\
\qquad\quad \land (speed = 0) \land (door = door\_down)\\
\qquad\quad \land (gear = gear\_down) \land (turn = 1)\\
AutopilotNext \triangleq\\
\qquad \land\ mode \in Modes \land height \in 0\,..\,max\_height\\
\qquad \land\ speed \in 0\,..\,max\_speed\\
\qquad \land\ (gear \neq gear\_down) \Rightarrow (height > threshold\_height)\\
\qquad \land\ (mode = \text{“landing”}) \Rightarrow (gear = gear\_down)\\
\qquad \land\ (mode = \text{“cruise”}) \Rightarrow ((gear = 0) \land (door = 0))\\
\qquad \land\ (height = 0) \Rightarrow (gear = gear\_down)\\
\qquad \land\ AutopilotTurn \lor \textsc{unchanged}\ \langle height,\ mode,\ speed\rangle\\
DoorNext \triangleq\\
\qquad \land\ door \in 0\,..\,door\_down\\
\qquad \land\ ((speed > threshold\_speed) \Rightarrow (door = 0))\\
\qquad \land\ DoorTurn \lor \textsc{unchanged}\ door\\
GearNext \triangleq\\
\qquad \land\ gear \in 0\,..\,gear\_down\\
\qquad \land\ (gear \neq 0) \Rightarrow (door = door\_down)\\
\qquad \land\ GearTurn \lor \textsc{unchanged}\ gear\\
SchedulerNext \triangleq\\
\qquad \land\ turn' = \textsc{if}\ turn = 3\ \textsc{then}\ 1\ \textsc{else}\ turn + 1\\
\qquad \land\ turn \in 1\,..\,3\\
Next \triangleq \ \land AutopilotNext \land GearNext\\
\qquad\qquad \land DoorNext \land SchedulerNext\\
vars \triangleq \langle mode,\ height,\ speed,\ door,\ gear,\ turn\rangle\\
Recurrence \triangleq \ \land \Box\Diamond(mode = \text{“landing”})\\
\qquad\qquad\qquad \land \Box\Diamond(mode = \text{“cruise”})\\
Spec \triangleq Init \land \Box[Next]_{vars} \land Recurrence
\end{array}$$

For brevity, we will let $mode \in 0..2$ in the discussion below. The components change in a way interleaving amongst them, based on the value of the variable $turn$. The scheduler changes its state in every step. So the scheduler changes in a noninterleaving way with respect to the other components. The specification has constant parameters $max\_height, \dots$ that define the range of values that the variables $height, speed, door, gear$ can take. Increasing the values of these constants produces instances of the specification with more states reachable.

The first operation is to restrict the assembly specification in order to ensure that it is machine-closed. The weakest invariant that ensures machine-closure is computed as the states from where the specification $\Box[Next]_{vars} \land Recurrence$ can be satisfied. For the constants $max\_height = 100, max\_speed = 40, door\_down = 5, gear\_down = 5, threshold\_height = 75, threshold\_speed = 30$, the resulting invariant is

$$\begin{array}{l}
Inv(door,\ gear,\ turn,\ height,\ mode,\ speed) \triangleq \\
\quad \land\ turn \in 1\,..\,3 \land door \in 0\,..\,5 \\
\quad \land\ gear \in 0\,..\,5 \land height \in 0\,..\,100 \\
\quad \land\ mode \in 0\,..\,2 \land speed \in 0\,..\,40 \\
\quad \land\ \quad \lor \land (door = 0) \land (gear = 0) \\
\qquad\qquad \land (height \in 76\,..\,100) \land (mode \in 1\,..\,2) \\
\qquad\quad \lor \land (door = 5) \land (gear = 5) \\
\qquad\qquad \land (mode = 0) \land (speed \in 0\,..\,30) \\
\qquad\quad \lor \land (door = 5) \land (gear = 5) \\
\qquad\qquad \land (mode = 2) \land (speed \in 0\,..\,30) \\
\qquad\quad \lor \land (door = 5) \land (height \in 76\,..\,100) \\
\qquad\qquad \land (mode = 2) \land (speed \in 0\,..\,30) \\
\qquad\quad \lor \land (gear = 0) \land (height \in 76\,..\,100) \\
\qquad\qquad \land (mode = 2) \land (speed \in 0\,..\,30)
\end{array}$$

From these states a centralized controller would be able to repeatedly enter landing and cruise mode, while taking $vars$-nonstuttering steps that satisfy the action $Next$.

We next examine the actions of the components. The result of applying the minimal covering method of Sec. VII is

$$\begin{array}{l}
AutopilotStep(door,\ gear,\ turn,\ height,\ mode,\ speed, \\
\qquad\qquad\qquad height',\ mode',\ speed') \triangleq \\
\qquad \land\ turn = 1 \land door \in 0\,..\,5 \land gear \in 0\,..\,5 \\
\qquad \land\ height \in 0\,..\,100 \land height' \in 0\,..\,100 \\
\qquad \land\ mode \in 0\,..\,2 \land mode' \in 0\,..\,2 \\
\qquad \land\ speed \in 0\,..\,40 \quad \land speed' \in 0\,..\,40 \\
\qquad \land\ \quad \lor \land (door = 0) \land (height' \in 76\,..\,100) \\
\qquad\qquad\qquad \land (mode' \in 1\,..\,2) \\
\qquad\qquad \lor (gear = 5) \land (height' \in 0\,..\,75) \\
\qquad\qquad \lor (gear = 5) \land (mode' = 0) \\
\qquad\qquad \lor \land (height' \in 76\,..\,100) \land (mode' = 2) \\
\qquad\qquad\qquad \land (speed' \in 0\,..\,30)
\end{array}$$

The two conjuncts below were used as care predicate.

$$\begin{array}{l}
\qquad \land\ Inv(door,\ gear,\ 1,\ height,\ mode,\ speed) \\
\qquad \land\ (\exists\, door,\ gear : \\
\qquad\qquad Inv(door,\ gear,\ 2,\ height,\ mode,\ speed))'
\end{array}$$

The action $AutopilotStep$ applies to steps that change the autopilot. The action that constrains the autopilot is

$$\begin{array}{l}
AutopilotNext(door,\ gear,\ turn,\ height,\ mode,\ speed, \\
\qquad\qquad\qquad height',\ mode',\ speed') \equiv \\
\qquad \land\ Inv(door,\ gear,\ turn,\ height,\ mode,\ speed) \\
\qquad \land\ \lor AutopilotStep(\\
\qquad\qquad\quad door,\ gear,\ turn,\ height,\ mode,\ speed, \\
\qquad\qquad\quad height',\ mode',\ speed') \\
\qquad\quad \lor \text{UNCHANGED}\ \langle height,\ mode,\ speed \rangle
\end{array}$$

Note that only variables that represent the autopilot appear primed in the action $AutopilotStep$.

Suppose that we have selected to hide the variable $door$. For this choice of variable, the invariant with $door$ abstracted is

$$\begin{array}{l}
InvWithDoorHidden \triangleq \exists\, door : \ Inv(\\
\qquad door,\ gear,\ turn,\ height,\ mode,\ speed)
\end{array}$$

Compared to the general case $\exists h : Inv(h, x, y)$,

- $door$ corresponds to h
- $gear, turn$ to x
- $height, mode, speed$ to y

Writing $InvWithDoorHidden$ is simple, but mysterious without recalling the definition of Inv. We cannot define the identifier $InvWithDoorHidden$ twice, but we can write another expression that is equivalent to it. Define

$$\begin{array}{l}
InvH \triangleq \\
\qquad \land\ turn \in 1\,..\,3 \\
\qquad \land\ gear \in 0\,..\,5 \land height \in 0\,..\,100 \\
\qquad \land\ mode \in 0\,..\,2 \land speed \in 0\,..\,40 \\
\qquad \land\ \quad \lor \land (gear = 0) \land (height \in 76\,..\,100) \\
\qquad\qquad\qquad \land (mode \in 1\,..\,2) \\
\qquad\qquad \lor \land (gear = 5) \land (mode = 0) \land (speed \in 0\,..\,30) \\
\qquad\qquad \lor \land (gear = 5) \land (mode = 2) \land (speed \in 0\,..\,30) \\
\qquad\qquad \lor \land (height \in 76\,..\,100) \land (mode = 2) \\
\qquad\qquad\qquad \land (speed \in 0\,..\,30)
\end{array}$$

This expression was obtained by decompiling the BDD that results after $door$ has been existentially quantified in the BDD representing Inv. This fact can be expressed by writing THEOREM $InvH \equiv InvWithDoorHidden$. How $InvH$ was obtained proves this equivalence.

Note that the type hints were used as the care set in this case, because the invariant implies them. Also, note that $InvH$ constrains all visible variables to be within the defined bounds.

The autopilot action that results after hiding the variable $door$ from the autopilot is

$$\begin{array}{l}
SimplerAutopilotStep \triangleq \\
\quad \land\ gear \in 0\,..\,5 \land height \in 0\,..\,100 \land height' \in 0\,..\,100 \\
\quad \land\ mode \in 0\,..\,2 \land mode' \in 0\,..\,2 \\
\quad \land\ speed \in 0\,..\,40 \land speed' \in 0\,..\,40 \\
\quad \land\ \quad \lor (gear = 5) \land (height' \in 0\,..\,75) \\
\qquad\quad \lor (gear = 5) \land (mode' = 0) \\
\qquad\quad \lor \land (gear \in 0\,..\,4) \land (height' \in 76\,..\,100) \\
\qquad\qquad \land (mode \in 0\,..\,1) \land (mode' \in 1\,..\,2) \\
\qquad\quad \lor \land (height' \in 76\,..\,100) \land (mode' = 2) \\
\qquad\qquad \land (speed' \in 0\,..\,30) \\
\qquad\quad \lor \land (height' \in 76\,..\,100) \land (mode' \in 1\,..\,2) \\
\qquad\qquad \land (speed \in 31\,..\,40) \\
\quad \land\ \text{LET}\ turn = 1 \quad \text{IN} \quad InvH \\
\quad \land\ (\exists\, turn,\ gear : \ InvH)'
\end{array}$$

where again we used the invariant as care set, in order to structure the resulting formulas more clearly, and modularize the covering problem. The operator $SimplerAutopilotStep$ defines the autopilot action by letting

$$\begin{array}{l}
SimplerAutopilotNext(gear,\ turn,\ height,\ mode,\ speed, \\
\qquad\qquad\qquad\qquad height',\ mode',\ speed') \triangleq \\
\qquad \land\ InvH
\end{array}$$

$\quad \wedge \; \vee (turn = 1) \wedge SimplerAutopilotStep$
$\qquad \vee \text{UNCHANGED} \langle height, mode, speed \rangle$

We chose to structure the autopilot action in this way because we already knew that the specification has an interleaving form. Hiding did not change the interleaving, but it did change how the autopilot is constrained when $turn = 1$. The environment action $SimplerEnvNext$ is obtained after existential quantification of $door$ and $door'$ from the environment action. The scheduler remains the same, changing $turn$ in each turn. In the gear module's turn ($turn = 3$), the action is

$$\begin{aligned}
&SimplerGearModuleNext \triangleq \\
&\quad \wedge gear \in 0 \,..\, 5 \wedge gear' \in 0 \,..\, 5 \\
&\quad \wedge height \in 0 \,..\, 100 \wedge mode \in 0 \,..\, 2 \\
&\quad \wedge speed \in 0 \,..\, 40 \\
&\quad \wedge \;\; \vee (gear \in 0 \,..\, 4) \wedge (gear' = 0) \\
&\qquad \vee (gear \in 1 \,..\, 5) \wedge (gear' = 5) \\
&\qquad \vee \wedge (height \in 76 \,..\, 100) \wedge (mode = 2) \\
&\qquad\quad \wedge (speed \in 0 \,..\, 30) \\
&\quad \wedge \text{LET } turn \triangleq 3 \;\text{ IN }\; InvH \\
&\quad \wedge (\text{LET } turn \triangleq 1 \\
&\qquad \text{IN } \;\exists height, mode, speed : \; InvH)'
\end{aligned}$$

This action includes primed values of both gear module and scheduler, because both form part of the autopilot's environment. The invariant has been used to define the care predicate (the last two conjuncts), which allowed for a simpler formula to be found.

The next step is the construction of the liveness parts of component specifications. Writing liveness specifications in this example is not as simple as it may appear. If we were to write these specifications by hand, a naive first attempt could be to assert that whenever the autopilot requests that the doors open and the landing gear is extended, the door and gear modules react accordingly. Such a specification would be incorrect, because it is too strong an assumption by the autopilot module, and too strong a guarantee for the door module. The door module cannot realize this requirement, because the autopilot is allowed to require this reaction while keeping the airspeed above $threshold\_speed$. This would prevent the doors from opening, thus the door module cannot realize this objective. Errors of this kind cannot result from the contract construction algorithm, because the way that it finds the module specifications ensures that each module can realize its own specification.

For constructing liveness specifications, we start with the autopilot as the "root" module, and the door module and gear module lumped into a "team". The basic algorithm described in Sec. VI-B cannot find a trap set, which demonstrates why the algorithm described in Sec. VI-C is needed. The reason is illustrated in Fig. 17, when the current goal of the autopilot is to enter cruise mode. The autopilot can enter cruise mode from the intermediate mode when the gear is retracted (up). The gear can retract when extended, but it could also idle, leading to the state on the bottom left. In that state, it is the autopilot's turn, and the autopilot could idle, or change the height to less than or equal to $threshold\_height$. This would prevent the gear from retracting. Therefore, the bottom left state is not in the team's attractor of A (the autopilot's attractor of cruise mode). This leads to an empty trap when the basic algorithm is used. By using the approach of Sec. VI-C, the $Basin$ is enlarged to incorporate the bottom left state, and a weaker goal is generated for the gear. This goal takes into account that the gear should reach either A, or the top left state (which corresponds to several states). The autopilot has a choice to not go backwards, thus it can keep the behavior within the two bottom left states, until the gear does retract.

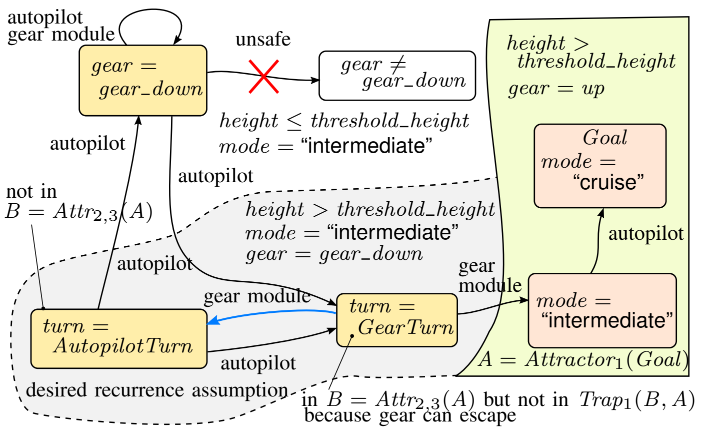


Fig. 17: The reason why the algorithm of Sec. VI-C is useful in the landing gear example.

Algo. 12 produces specifications for the autopilot and the lumped door and gear modules. Different mask parameters are used for each recurrence goal, thus different interconnection architectures. The goal that is generated for the lumped modules is used as the goal in a recursive call to Algo. 12. This recursive call refines the interconnection architecture further, by generating separate specifications for the gear module and the door module. We show next the generated specifications for when the goal of the autopilot is to enter cruise mode. The autopilot trap is

$$\begin{aligned}
&AutopilotTrap \triangleq \\
&\wedge turn \;\in 1 \,..\, 3 \wedge door \in 0 \,..\, 5 \wedge height \in 0 \,..\, 100 \\
&\wedge mode \in 0 \,..\, 2 \wedge speed \in 0 \,..\, 40 \\
&\wedge \;\; \vee \wedge (turn \;= 2) \wedge (height \in 76 \,..\, 100) \\
&\qquad\;\; \wedge (mode = 2) \wedge (speed \in 0 \,..\, 30) \\
&\quad\;\; \vee \wedge (door \;\in 1 \,..\, 5) \wedge (height \in 76 \,..\, 100) \\
&\qquad\;\; \wedge (mode = 2) \wedge (speed \in 0 \,..\, 30)
\end{aligned}$$

and the resulting persistence objective $\Diamond\Box(AutopilotTrap \wedge (cnct = 0))$. As expected, the autopilot is allowed to keep waiting while the doors are still open ($door \in 1..5$ in second disjunct), and until the gear reacts, only *provided* the autopilot has reached and maintains the height above the threshold ($height \in 76..100$), and it keeps the mode to intermediate. The last two constraints are required because height below the threshold, or mode equal to landing would prevent the gear from being able to retract. Notice that the autopilot does not need to observe the gear state, only the door state, because when the doors close, the global invariant Inv implies that the gear has been retracted too. Therefore, the specification of the autopilot in this interconnection mode is expressed without occurrence of the variable $gear$.

The corresponding recurrence goal $\Box\Diamond((cnct \neq 0) \vee \neg DTeam)$ for the door-gear subsystem is defined by

$$\begin{aligned}
&DTeam \triangleq \\
&\wedge\ turn \in 1\,..\,3 \wedge door \in 0\,..\,5 \wedge gear \in 0\,..\,5 \\
&\wedge\ height \in 0\,..\,100 \wedge mode \in 0\,..\,2 \\
&\wedge\ \quad \vee\ \wedge (turn = 2) \wedge (gear = 0) \\
&\qquad\quad \wedge (height \in 76\,..\,100) \wedge (mode = 2) \\
&\qquad \vee\ \wedge (door\ = 5) \wedge (height \in 76\,..\,100) \\
&\qquad\quad \wedge (mode = 2) \\
&\qquad \vee\ \wedge (door\ \in 1\,..\,5) \wedge (gear = 0) \\
&\qquad\quad \wedge (height \in 76\,..\,100) \wedge (mode = 2)
\end{aligned}$$

While the doors are open ($door = 5$ in second disjunct, or $door \in 1..5$), the subsystem is required to change by closing the door, which implies retracting the gear. When both gear have been retracted ($gear = 0$) and doors closed ($door = 0$), then the subsystem needs to wait until the autopilot's turn. The earliest this can happen is by the gear retracting ($turn = 3$) and then the doors closing ($turn = 2$), hence the $turn = 2$ in the first disjunct.

From the subsystem's viewpoint, both of the variables $door$ and $gear$ are visible, so its specification in this interconnection mode mentions both. Notice that there is no mention of $speed$, because it is unnecessary information for reaching cruise mode. If the doors are already closed, then they need not open while transitioning from intermediate to cruise mode, so they need not know the airspeed. If the doors are currently open, then the invariant Inv implies that the airspeed is below the threshold, and that the autopilot will maintain this invariant. The airspeed is unnecessary information while the doors transition from open to closed.

The variable $cnct$ is introduced to define the current interconnection mode, and is controlled by the autopilot. When $cnct$ changes, the other modules are constrained to change the information that they communicate, by changing the domains of the record-valued variables that are used for communication between the modules.

When the subsystem of gear module and door module is decomposed into two separate components, using $\neg DTeam$ as the goal, the generated specifications are as follows. For the gear module

$$\begin{aligned}
&Gear\_Trap \triangleq \\
&\wedge\ turn \in 1\,..\,3 \wedge door \in 0\,..\,5 \\
&\wedge\ gear \in 0\,..\,5 \wedge height \in 0\,..\,100 \wedge mode \in 0\,..\,2 \\
&\wedge\ \quad \vee\ \wedge (turn = 2) \wedge (gear = 0) \\
&\qquad\quad \wedge (height \in 76\,..\,100) \wedge (mode = 2) \\
&\qquad \vee\ \wedge (door \in 1\,..\,5) \wedge (gear = 0) \\
&\qquad\quad \wedge (height \in 76\,..\,100) \wedge (mode = 2)
\end{aligned}$$

and for the door module

$$\begin{aligned}
&D\_Door\_module \triangleq \\
&\wedge\ turn \in 1\,..\,3 \wedge door \quad \in 0\,..\,5 \wedge gear \in 0\,..\,5 \\
&\wedge\ \quad \vee (turn = 2) \wedge (gear = 0) \\
&\qquad \vee (door \in 1\,..\,5) \wedge (gear = 0)
\end{aligned}$$

Again, these goals are conditioned using the current interconnection $cnct$. The above specifications can be understood as follows. The gear assumes that the doors will close, provided the gear has retracted itself (conjunct $gear = 0$). The gear cannot assume that the doors will close while the gear is still extended, because that would be too strong an assumption. It would be realizable by the gear, but unrealizable by the doors. Similar to what we remarked about the autopilot earlier, this is an error that could arise if we specified the subsystem goals by hand, instead of generating them automatically. Provided the gear has retracted, it is allowed to wait until the door retracts, and also until it is the autopilot's turn. Note that the gear does not receive $speed$ information in this interconnection, which is shown in Fig. 18. For the doors, the requirement is that if the gear has retracted ($gear = 0$), then the doors should not be open ($door \in 1..5$).

The above discussion corresponds to the interconnection architecture while the autopilot has cruise mode as its current goal. A different interconnection architecture, shown in Fig. 19 is computed to allow the autopilot to reach landing mode. The resulting specifications have an analogous structure with those described above, though the direction of change for the entire system is the opposite (the autopilot should lower the airspeed to allow the doors to open, and also change from cruise to intermediate mode, then the autopilot is allowed to wait for the doors to open, and for the landing gear to extend, then the autopilot can enter landing mode). An interesting observation regarding the connectivity in Fig. 19 is that the gear needs to observe both $mode$ and $speed$. This requirement results because the gear module needs to be able to observe globally that the doors are still within D_Door_module, which requires information about the $mode$ and $speed$. However, we would expect this to be information necessary only to the door module. Indeed, by using the complement of the subsystem goal as $Within$ to change $Stay := Observable(D, Inv, Inv, Player)$ in Algo. 12 to $Stay := Observable(D, Within \wedge Inv, Inv, Player)$, as described in Sec. VI-G2, the generated specification for the gear becomes independent of $mode$ and $speed$, and those signals are removed from the interconnection architecture. The resulting persistence goal for the gear becomes weaker. In this example, there is one trap formed for the subsystem, so this weakening

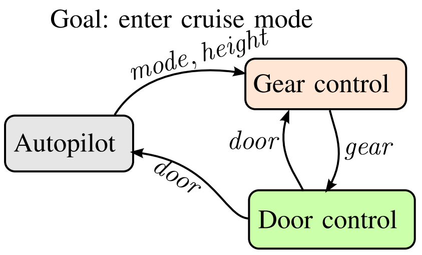


Fig. 18: Communicated variables when goal is cruise mode.

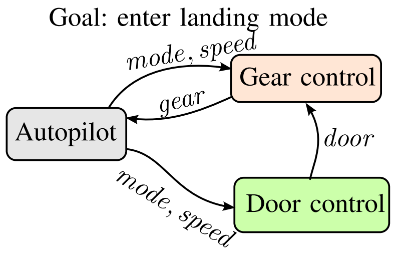


Fig. 19: Communicated variables when goal is landing mode.

is admissible. In problems where this is not the case, either an interconnection architecture with more information sharing needs to be used, or the weaker persistence goals checked to ensure that they do not intersect with other traps, or are contained within the persistence goal for the same component within another subsystem trap.

The above example demonstrated the applicability of the proposed approach to systems with multiple components, by recursive decomposition, and by construction of interconnection architectures with only necessary information shared between components. An implementation of the algorithms described is available in a Python package [134].

IX. Conclusions and Future Work

In this paper, we developed an approach for decomposing the temporal logic specification of an assembly to open-system component specifications that form a contract. We defined contracts based on a formalized definition of realizability, the notion of open-system based on cardinality, and defined an operator for forming open-systems from closed-systems. The decomposition approach relies on generating liveness requirements for individual components in a way that leads to acyclic dependencies. In order to hide unnecessary external information from each component, we parametrized contract construction with respect to the interconnection architecture, and showed how variables can be eliminated from component specifications. The generated specifications were decompiled from BDDs by using a symbolic minimal covering algorithm, adapted to the case of integer variables. Directions for future investigation include reducing the sharing of information further, relaxing the scheduling assumptions, generating more readable specifications, and comparing different formalizations of contracts and synthesis.

Acknowledgments: In studying the meaning and form of open-system properties we benefited from discussions with Necmiye Özay and Scott Livingston.

This work was supported in part by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] N. Wirth, "Program development by stepwise refinement," *CACM*, vol. 14, no. 4, pp. 221–227, 1971.

[2] L. Lamport, "Who builds a house without drawing blueprints?" *CACM*, vol. 58, no. 4, pp. 38–41, 2015.

[3] C. B. Jones, "Specification and design of (parallel) programs," in *Information Processing*, 1983, pp. 321–332.

[4] J. C. Willems, "The behavioral approach to open and interconnected systems: Modeling by tearing, zooming, and linking," *IEEE CSM*, pp. 46–99, Dec 2007.

[5] R. Kurki-Suonio, "Component and interface refinement in closed-system specifications," in *Formal Methods*, 1999, pp. 134–154.

[6] R. Kurshan and L. Lamport, "Verification of a multiplier: 64 bits and beyond," in *CAV*, 1993, pp. 166–179.

[7] L. Lamport, "Composition: A way to make proofs harder," in *COMPOS*, 1997, pp. 402–423.

[8] ——, "Miscellany," 21 April 1991, sent to TLA mailing list. [Online]. Available: http://lamport.azurewebsites.net/tla/notes/91-04-21.txt

[9] T. Bourke, M. Daum, G. Klein, and R. Kolanski, "Challenges and experiences in managing large-scale proofs," in *Intelligent Computer Mathematics (CICM)*, 2012, pp. 32–48.

[10] L. Lamport, *Specifying systems: The TLA+ language and tools for hardware and software engineers*. Addison-Wesley, 2002.

[11] ——, "What good is temporal logic?" in *Information Processing*, 1983, pp. 657–668.

[12] ——, "Specifying concurrent program modules," *TOPLAS*, vol. 5, no. 2, pp. 190–222, 1983.

[13] W. Thomas, "Solution of Church's problem: A tutorial," *New Perspectives on Games and interaction*, vol. 5, 2008.

[14] R. Rosner, "Modular synthesis of reactive systems," Ph.D. dissertation, Weizmann Institute of Science, Rehovot, Israel, 1992.

[15] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," in *VMCAI*, 2006, pp. 364–380.

[16] N. Piterman and A. Pnueli, "Faster solutions of Rabin and Streett games," in *LICS*, 2006, pp. 275–284.

[17] U. Klein, N. Piterman, and A. Pnueli, "Effective synthesis of asynchronous systems from GR(1) specifications," in *VMCAI*, 2012, pp. 283–298.

[18] A. Walker and L. Ryzhyk, "Predicate abstraction for reactive synthesis," in *FMCAD*, 2014, pp. 219–226.

[19] O. Kupferman and M. Vardi, "Safraless decision procedures," in *FOCS*, 2005, pp. 531–540.

[20] M. de Wulf, L. Doyen, and J.-F. Raskin, "A lattice theory for solving games of imperfect information," in *HSCC*, 2006, pp. 153–168.

[21] R. Ehlers and U. Topcu, "Estimator-based reactive synthesis under incomplete information," in *HSCC*, 2015, pp. 249–258.

[22] A. Pnueli and U. Klein, "Synthesis of programs from temporal property specifications," in *MEMOCODE*, 2009, pp. 1–7.

[23] A. Pnueli and R. Rosner, "Distributed reactive systems are hard to synthesize," in *FOCS*, vol. 2, 1990, pp. 746–757.

[24] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *TC*, vol. 35, no. 8, pp. 677–691, 1986.

[25] Y. Kesten, A. Pnueli, and L.-o. Raviv, "Algorithmic verification of linear temporal logic specifications," in *ICALP*, 1998, pp. 1–16.

[26] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[27] O. Coudert, "Two-level logic minimization: An overview," *Integration, the VLSI Journal*, vol. 17, no. 2, pp. 97–140, 1994.

[28] C. B. Jones, "The early search for tractable ways of reasoning about programs," *IEEE Annals of the History of Computing*, vol. 25, no. 2, pp. 26–49, Apr 2003.

[29] D. E. Knuth, "Robert w floyd, In memoriam," *ACM SIGACT News*, vol. 34, no. 4, pp. 3–13, 2003.

[30] H. H. Goldstine and J. von Neumann, *Planning and coding problems for an electronic computing instrument: Report on the mathematical and logical aspects of an electronic computing instrument, Part II*. Princeton, NJ: The Institute for Advanced Study, 1947, vol. 1. [Online]. Available: https://library.ias.edu/files/pdfs/ecp/planningcodingof0103inst.pdf

[31] A. M. Turing, "Checking a large routine," in *Report of a conference on High Speed Automatic Calculating Machines*, Mathematical Laboratory, Cambridge, 24 Jun 1949, pp. 67–69. [Online]. Available: http://www.turingarchive.org/browse.php/B/8

[32] F. L. Morris and C. B. Jones, "An early program proof by Alan Turing," *IEEE Annals of the History of Computing*, vol. 6, no. 2, pp. 139–143, Apr 1984.

[33] C. Hoare, "An axiomatic basis for computer programming," *CACM*, vol. 12, no. 10, pp. 576–580, 1969.

[34] R. W. Floyd, "Assigning meanings to programs," in *Symposia in Applied Mathematics*, ser. Aspects of Computer Science, vol. 19. AMS, 1967, pp. 19–32.

[35] F. B. Schneider, *On concurrent programming*. Springer, 1997.

[36] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *POPL*, 1989, pp. 179–190.

[37] N. Francez and A. Pnueli, "A proof method for cyclic programs," *Acta Informatica*, vol. 9, pp. 133–157, 1978.

[38] A. Pnueli, "In transition from global to modular temporal reasoning about programs," in *Logics and models of concurrent systems*, ser. NATO ASI Series F: Computer and Systems Sciences, 1985, vol. 13, pp. 123–144.

[39] L. Lamport, "The "Hoare logic" of concurrent programs," *Acta Informatica*, vol. 14, pp. 21–37, 1980.

[40] L. Lamport and F. B. Schneider, "The "Hoare Logic" of CSP, and all that," *TOPLAS*, vol. 6, no. 2, pp. 281–296, 1984.

[41] J. Misra and K. Chandy, "Proofs of networks of processes," *TSE*, vol. 7, no. 4, pp. 417–426, 1981.

[42] A. Pnueli, "The temporal logic of programs," in *FOCS*, 1977, pp. 46–57.

[43] L. Lamport, "Proving the correctness of multiprocess programs," *TSE*, vol. 3, no. 2, pp. 125–143, 1977.

[44] S. Owicki and L. Lamport, "Proving liveness properties of concurrent programs," *TOPLAS*, vol. 4, no. 3, pp. 455–495, 1982.

[45] G. Rock, W. Stephan, and A. Wolpers, *Modular reasoning about structured TLA specifications*, ser. Advances in Computing Science. Springer, 1999, pp. 217–229.

[46] A. Wolpers and W. Stephan, "Modular verification of programmable logic controllers with TLA," in *IFAC INCOM*, vol. 31, no. 15, 1998, pp. 159–164.

[47] M. Abadi and L. Lamport, "Open systems in TLA," in *PODC*, 1994, pp. 81–90.

[48] M. Abadi and S. Merz, "On TLA as a logic," in *Proceedings of the NATO Advanced Study Institute on Deductive Program Design*, ser. NATO ASI series F. Springer, 1996, pp. 235–272, held in Marktoberdorf, Germany, July 26–Aug 7, 1994. [Online]. Available: https://members.loria.fr/SMerz/papers/mod94.html

[49] E. W. Stark, "A proof technique for rely/guarantee properties," *Foundations of Software Technology and Theoretical Computer Science*, vol. 206, pp. 369–391, 1985.

[50] K. L. McMillan, "Circular compositional reasoning about liveness," in *CHARME*, 1999, pp. 342–346.

[51] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[52] L. de Alfaro and T. Henzinger, "Interface automata," in *ESEC/FSE*, 2001, pp. 109–120.

[53] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K. Larsen, "Contracts for systems design," INRIA, Tech. Rep. 8147, 2012.

[54] P. Nuzzo, "Compositional design of cyber-physical systems using contracts," Ph.D. dissertation, University of California, Berkeley, Aug 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-189.html

[55] P. Nuzzo, A. Iannopollo, S. Tripakis, and A. Sangiovanni-Vincentelli, "Are interface theories equivalent to contract theories?" in *MEMOCODE*, 2014, pp. 104–113.

[56] A. Cimatti and S. Tonetta, "A property-based proof system for contract-based design," in *EUROMICRO*, 2012, pp. 21–28.

[57] A. Cimatti, M. Dorigatti, and S. Tonetta, "OCRA: A tool for checking the refinement of temporal contracts," in *ASE*, 2013, pp. 702–705.

[58] R. Alur, T. Henzinger, and O. Kupferman, "Alternating-time temporal logic," in *JACM*, 2002, pp. 672–713.

[59] L. de Alfaro, T. A. Henzinger, and F. Y. Mang, "The control of synchronous systems," in *CONCUR*, 2000, pp. 458–473.

[60] L. de Alfaro and M. Faella, "Information flow in concurrent games," in *ICALP*, 2003, pp. 1038–1053.

[61] S. Schewe and B. Finkbeiner, "Synthesis of asynchronous systems," in *LOPSTR*, 2007, pp. 127–142.

[62] O. Kupferman and M. Y. Vardi, "Synthesis with incomplete information," in *Advances in Temporal Logic*. Springer, 2000, pp. 109–127.

[63] B. Finkbeiner and S. Schewe, "Uniform distributed synthesis," in *LICS*, 2005, pp. 321–330.

[64] K. Chatterjee, T. Henzinger, J. Otop, and A. Pavlogiannis, "Distributed synthesis for LTL fragments," in *FMCAD*, 2013, pp. 18–25.

[65] B. Finkbeiner and S. Schewe, "Bounded synthesis," *STTT*, vol. 15, no. 5–6, pp. 519–539, 2013.

[66] K. Chatterjee and T. Henzinger, "Assume-guarantee synthesis," *TACAS*, pp. 261–275, 2007.

[67] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu, "Learning assumptions for compositional verification," in *TACAS*, 2003, pp. 331–346.

[68] W. Nam and R. Alur, "Learning-based symbolic assume-guarantee reasoning with automatic decomposition," in *ATVA*, 2006, pp. 170–185.

[69] K. Chatterjee, T. Henzinger, and B. Jobstmann, "Environment assumptions for synthesis," in *CONCUR*, 2008, pp. 147–161.

[70] M. Abadi and L. Lamport, "Conjoining specifications," *TOPLAS*, vol. 17, no. 3, pp. 507–535, 1995.

[71] R. Könighofer, G. Hofferek, and R. Bloem, "Debugging formal specifications: A practical approach using model-based diagnosis and counterstrategies," *STTT*, vol. 15, no. 5–6, pp. 563–583, 2013.

[72] W. Li, L. Dworkin, and S. A. Seshia, "Mining assumptions for synthesis," in *MEMOCODE*, 2011, pp. 43–50.

[73] R. Alur, S. Moarref, and U. Topcu, "Counter-strategy guided refinement of GR(1) temporal logic specifications," in *FMCAD*, 2013, pp. 26–33.

[74] ——, "Pattern-based refinement of assume-guarantee specifications in reactive synthesis," in *TACAS*, 2015, pp. 501–516.

[75] R. Ehlers, R. Könighofer, and R. Bloem, "Synthesizing cooperative reactive mission plans," in *IROS*, 2015, pp. 3478–3485.

[76] S. Moarref, "Compositional reactive synthesis for multi-agent systems," Ph.D. dissertation, University of Pennsylvania, 2016.

[77] A. Cohen and K. S. Namjoshi, "Local proofs for global safety properties," *FMSD*, vol. 34, no. 2, pp. 104–125, 2009.

[78] J. Fu and U. Topcu, "Integrating active sensing into reactive synthesis with temporal logic constraints under partial observations," in *ACC*, 2015, pp. 2408–2413.

[79] O. Mickelin, N. Ozay, and R. M. Murray, "Synthesis of correct-by-construction control protocols for hybrid systems using partial state information," in *ACC*, 2014, pp. 2305–2311.

[80] N. Ozay, U. Topcu, and R. M. Murray, "Distributed power allocation for vehicle management systems," in *CDC*, 2011, pp. 4841–4848.

[81] T. Mikkonen, "Abstractions and logical layers in specifications of reactive systems," Ph.D. dissertation, Tampere University of Technology, Feb 1999. [Online]. Available: www.cs.tut.fi/~tjm/doc.ps

[82] M. Katara and T. Mikkonen, "Design approach for real-time reactive systems," in *Intl. Work. on Real-Time Constraints*, 1999. [Online]. Available: https://www.cs.ccu.edu.tw/~pahsiung/cp99-rtc/proceedings.html

[83] H. Thimbleby and P. Ladkin, "From logic to manuals again," *IEE Proc.-Soft. Eng.*, vol. 144, no. 3, 1997.

[84] S. Singh and M. D. Wagh, "Robot path planning using intersecting convex shapes: Analysis and simulation," *TRO*, vol. RA-3, no. 2, 1987.

[85] B. Hayes and J. A. Shah, "Improving robot controller transparency through autonomous policy explanation," in *Human-Robot Interaction*, 2017, pp. 303–312.

[86] R. Ehlers and V. Raman, "Low-effort specification debugging and analysis," *EPTCS*, vol. 157, pp. 117–133, 2014.

[87] I. Dillig, T. Dillig, K. L. McMillan, and A. Aiken, "Minimum satisfying assignments for SMT," in *CAV*, 2012, pp. 394–409.

[88] L. Lamport, "The temporal logic of actions," *TOPLAS*, vol. 16, no. 3, pp. 872–923, 1994.

[89] S. Merz, "Rules for abstraction," in *Advances in Computing Science—ASIAN'97*, 1997, pp. 32–45.

[90] H.-D. Ebbinghaus, *Ernst Zermelo: An approach to his life and work*. Springer, 2007.

[91] L. Lamport, "TLA^{+2}: A preliminary guide," Tech. Rep., 15 Jan 2014.

[92] D. Hilbert and P. Bernays, *Grundlagen der Mathematik II*. Springer, 1970.

[93] A. C. Leisenring, *Mathematical Logic and Hilbert's ε-symbol*. MacDonald Technical & Scientific, 1969.

[94] B. Alpern and F. B. Schneider, "Defining liveness," *IPL*, vol. 21, no. 4, pp. 181–185, 1985.

[95] K. Kunen, *Set theory*, ser. Studies in Logic. College Publications, 2013, vol. 34.

[96] I. Filippidis and R. M. Murray, "Formalizing synthesis in TLA^{+}," California Institute of Technology, Tech. Rep. CaltechCDSTR:2016.004, 2016. [Online]. Available: http://resolver.caltech.edu/CaltechCDSTR:2016.004

[97] M. Abadi, L. Lamport, and P. Wolper, "Realizable and unrealizable specifications of reactive systems," in *ICALP*, 1989, pp. 1–17.

[98] A. Pnueli and R. Rosner, "A framework for the synthesis of reactive modules," in *CONCURRENCY*, 1988, pp. 4–17.

[99] Y. Kesten, N. Piterman, and A. Pnueli, "Bridging the gap between fair simulation and trace inclusion," *Inf. Comput.*, vol. 200, no. 1, pp. 35–61, 2005.

[100] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reacive(1) designs," *JCSS*, vol. 78, no. 3, pp. 911–938, 2012.

[101] W. Thomas, "On the synthesis of strategies in infinite games," in *STACS*, 1995, pp. 1–13.

[102] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program repair as a game," in *CAV*, 2005, pp. 226–238.

[103] I. Filippidis, S. Dathathri, S. C. Livingston, N. Ozay, and R. M. Murray, "Control design for hybrid systems with TuLiP: The temporal logic planning toolbox," in *2016 IEEE Conference on Control Applications (CCA)*, 2016, pp. pp.1030–1041.

[104] H. Vanzetto, "Proof automation and type synthesis for set theory in the context of TLA+," Ph.D. dissertation, Université de Lorraine, Dec 2014. [Online]. Available: https://hal.inria.fr/tel-01096518

[105] L. Lamport and L. C. Paulson, "Should your specification language be typed?" *TOPLAS*, vol. 21, no. 3, pp. 502–526, May 1999.

[106] I. Filippidis and R. M. Murray, "Symbolic construction of GR(1) contracts for systems with full information," in *ACC*, 2016, pp. 782–789.

[107] S. C. Kleene, *Introduction to metamathematics*. North-Holland, 1971.

[108] K. Kunen, *The foundations of mathematics*, ser. Studies in Logic. College Publications, 2012, vol. 19.

[109] U. Klein and A. Pnueli, "Revisiting synthesis of GR(1) specifications," in *HVC*, 2010, pp. 161–181.

[110] M. Abadi and S. Merz, "An abstract account of composition," in *Int. Symp. on Mathematical Foundations of Computer Science (MFCS)*, 1995, pp. 499–508.

[111] B. Jonsson and Y.-K. Tsay, "Assumption/guarantee specifications in linear-time temporal logic," *TCS*, vol. 167, no. 1–2, pp. 47–72, 1996.

[112] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *CACM*, vol. 8, no. 9, p. 569, 1965.

[113] M. Abadi and L. Lamport, "The existence of refinement mappings," *TCS*, vol. 82, no. 2, pp. 253–284, 1991.

[114] B. Alpern and F. B. Schneider, "Recognizing safety and liveness," *Distributed Computing*, vol. 2, no. 3, pp. 117–126, 1987.

[115] L. Lamport, "Proving possibility properties," *TCS*, vol. 206, no. 1–2, pp. 341–352, 1998.

[116] I. Filippidis and R. M. Murray, "Symbolic construction of GR(1) contracts for synchronous systems with full information," Caltech, Tech. Rep. arXiv:1508.02705, 2015.

[117] ——, "Hiding variables when decomposing specifications into GR(1) contracts," in *CDC (under review)*, 2017. [Online]. Available: http://www.cds.caltech.edu/~murray/preprints/fm17-cdc_s.pdf

[118] L. Lamport, "How to write a 21st century proof," *Journal of fixed point theory and applications*, vol. 11, no. 1, pp. 43–63, 2012.

[119] R. E. Bryant, "On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication," *TOC*, vol. 40, no. 2, pp. 205–213, 1991.

[120] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem, "ANZU: A tool for property synthesis," in *CAV*, 2007, pp. 258–262.

[121] A. Pnueli, Y. Sa'ar, and L. D. Zuck, "JTLV: A framework for developing verification algorithms," in *CAV*, 2010, pp. 171–174.

[122] T. Villa, T. Kam, R. K. Brayton, and A. Sangiovanni-Vincentelli, *Synthesis of finite state machines: Logic optimization*. Springer, 1997.

[123] G. D. Hachtel and F. Somenzi, *Logic synthesis and verification algorithms*. Kluwer, 1996.

[124] P. C. McGeer, J. V. Sanghavi, R. K. Brayton, and A. L. Sangiovanni-Vicentelli, "ESPRESSO-SIGNATURE: A new exact minimizer for logic functions," *TVLSI*, vol. 1, no. 4, pp. 432–440, 1993.

[125] R. L. Rudell, "Logic synthesis for VLSI design," Ph.D. dissertation, University of California, Berkeley, 1989.

[126] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic minimization algorithms for VLSI synthesis*. Kluwer, 1984.

[127] O. Coudert and J. C. Madre, "New ideas for solving covering problems," in *Design Automation Conference (DAC)*, 1995, pp. 641–646.

[128] O. Coudert, J. C. Madre, H. Fraisse, and H. Touati, "Implicit prime cover computation: An overview," in *Synthesis and Simulation Meeting and International Interchange (SASIMI)*, 1993. [Online]. Available: http://sasimi.jp

[129] O. Coudert, J. C. Madre, and H. Fraisse, "A new viewpoint on two-level logic minimization," in *Design Automation Conference (DAC)*, 1993, pp. 625–630.

[130] O. Coudert and J. C. Madre, "Implicit and incremental computation of primes and essential primes of Boolean functions," in *Design Automation Conference (DAC)*, 1992, pp. 36–39.

[131] I. Filippidis, R. M. Murray, and G. J. Holzmann, "A multi-paradigm language for reactive synthesis," in *4th Work. on Synthesis, SYNT*, 2015, pp. 73–97. [Online]. Available: https://doi.org/10.4204/EPTCS.202.6

[132] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *ICCAD*, 1993, pp. 42–47.

[133] F. Boniol, V. Wiels, Y. Ait Ameur, and K.-D. Schewe, Eds., *ABZ 2014: The landing gear case study*, 2014.

[134] "Contract construction implementation." [Online]. Available: https://github.com/johnyf/contract_maker

APPENDIX

Below is the proof of Theorem 3 on 16. The proof is structured in levels [118], [91].

```
PROOF:
$\langle 1\rangle 1.\ \mathit{IsRealizable}_2(\Box\Diamond\neg D)$
  $\langle 2\rangle 1.\ \mathit{IsRealizable}_2(\Box(D \Rightarrow \Diamond(U \vee \mathit{Out})))$
    $\langle 3\rangle 1.$ DEFINE $Z \triangleq \mathit{Attr}_2(U \vee \mathit{Out})$
    $\langle 3\rangle 2.\ D \Rightarrow Z$
      BY DEF $D$
    $\langle 3\rangle 3.\ \mathit{IsRealizable}_2(\Box(Z \Rightarrow \Diamond(U \vee \mathit{Out})))$
      BY DEF $\mathit{Attr}$
    $\langle 3\rangle 4.$ Q.E.D.
      BY $\langle 3\rangle 2$, $\langle 3\rangle 3$
  $\langle 2\rangle 2.\ (U \vee \mathit{Out}) \Rightarrow \neg D$
    $\langle 3\rangle 1.\ D \Rightarrow \neg \mathit{Out}$
      $\langle 4\rangle 1.\ \mathit{Basin} \Rightarrow \neg \mathit{Out}$
        BY DEF $\mathit{Out}$
      $\langle 4\rangle 2.\ D \Rightarrow \mathit{Basin}$
        BY DEF $D$
      $\langle 4\rangle 3.$ Q.E.D.
        BY $\langle 4\rangle 1$, $\langle 4\rangle 2$
    $\langle 3\rangle 2.\ D \Rightarrow \neg U$
      BY DEF $D$
    $\langle 3\rangle 3.$ Q.E.D.
      BY $\langle 3\rangle 1$, $\langle 3\rangle 2$
  $\langle 2\rangle 3.\ \mathit{IsRealizable}_2(\Box(D \Rightarrow \Diamond\neg D))$
    BY $\langle 2\rangle 1$, $\langle 2\rangle 2$
  $\langle 2\rangle 4.\ \Box(D \Rightarrow \Diamond\neg D) \equiv \Box\Diamond\neg D$
    PROOF:
          $\Box(D \Rightarrow \Diamond\neg D) \equiv \Box \vee \neg D$
                                   $\vee\ D \wedge (D \Rightarrow \Diamond\neg D)$
          $\equiv \Box(\neg D \vee \Diamond\neg D)$
  $\langle 2\rangle 5.$ Q.E.D.
    BY $\langle 2\rangle 3$, $\langle 2\rangle 4$
$\langle 1\rangle 2.\ \mathit{IsRealizable}_1(\Box \vee \neg(T \vee A)$
                          $\vee\ \Diamond A \vee \Diamond\Box T)$
  $\langle 2\rangle 1.$ DEFINE $Z \triangleq \mathit{Trap}_1(\mathit{Stay}, A)$
  $\langle 2\rangle 2.\ \mathit{IsRealizable}_1(\Box(Z \Rightarrow \vee\ \Diamond A$
                               $\vee\ \Diamond\Box(Z \wedge \neg A)))$
    BY DEFS $Z$, $\mathit{Trap}$
  $\langle 2\rangle 3.\ T \equiv Z \wedge \neg A$
    BY DEFS $T$, $Z$
  $\langle 2\rangle 4.\ (T \vee A) \Rightarrow Z$
    $\langle 3\rangle 1.\ (T \vee A) \Rightarrow ((Z \wedge \neg A) \vee A)$
      BY $\langle 2\rangle 3$
    $\langle 3\rangle 2.\ (T \vee A) \Rightarrow (Z \vee A)$
      BY $\langle 3\rangle 1$
    $\langle 3\rangle 3.\ (Z \vee A) \Rightarrow Z$
      $\langle 4\rangle 1.\ A \Rightarrow Z$
        BY DEF $Z$, $\mathit{Trap}$
      $\langle 4\rangle 2.$ Q.E.D.
        BY $\langle 4\rangle 1$
    $\langle 3\rangle 4.$ Q.E.D.
      BY $\langle 3\rangle 2$, $\langle 3\rangle 3$
  $\langle 2\rangle 5.$ Q.E.D.
    BY $\langle 2\rangle 2$, $\langle 2\rangle 3$, $\langle 2\rangle 4$
$\langle 1\rangle 3.\ T \Rightarrow D$
```

$\langle 2\rangle 1.$ $T \equiv \mathit{Trap}_1(\mathit{Stay}, A) \land \neg A$
BY DEF T
$\langle 2\rangle 2.$ $\mathit{Trap}_1(\mathit{Stay}, A) \Rightarrow (\mathit{Stay} \lor A)$
BY DEF Trap
$\langle 2\rangle 3.$ $T \Rightarrow (\mathit{Stay} \land \neg A)$
BY $\langle 2\rangle 1$, $\langle 2\rangle 2$
$\langle 2\rangle 4.$ $\mathit{Stay} \Rightarrow D$
BY DEFS Stay, Obs_1
$\langle 2\rangle 5.$ Q.E.D.
BY $\langle 2\rangle 3$, $\langle 2\rangle 4$
$\langle 1\rangle 4.$ Q.E.D.
BY $\langle 1\rangle 1$, $\langle 1\rangle 2$, $\langle 1\rangle 3$