To appear, Proceedings of the IEEE, 2018
http://www.cds.caltech.edu/~murray/preprints/fm18-pieee.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. We parametrize the information flow between components by introducing parameters that select whether each variable is visible to a component or not. The decomposition algorithm identifies which variables can be hidden while preserving realizability and ensuring correct composition, and these are eliminated from component specifications by quantification and conversion of binary decision diagrams to formulas. 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. This operator simplifies the writing of specifications that are realizable without being vacuous. 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 however, the controller requires 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.

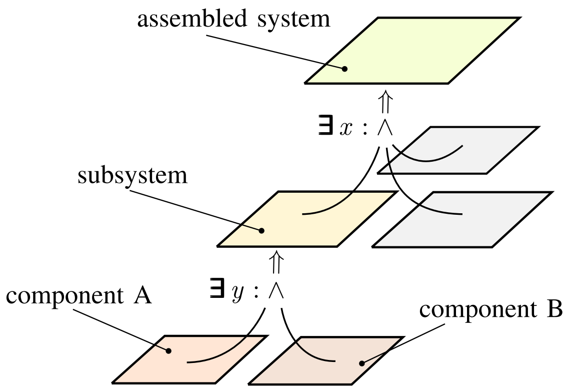


Fig. 1: Anatomy of hierarchical system design in TLA$^+$: composition is represented by conjunction ($\wedge$), hiding of details by (temporal) existential quantification ($\exists$), and refinement by logical implication $\Rightarrow$.

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 with sufficient accuracy to describe the intended system operation without ambiguity [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

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

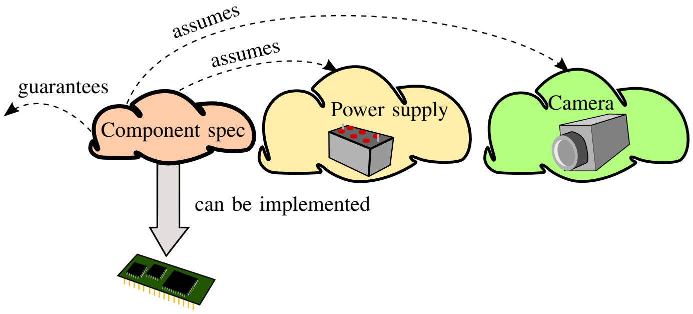


Fig. 2: A component is specified by expressing requirements that the implementation should fulfill under assumptions about other parts of the assembled system.

at the implementation level can then be derived by hand or using (automated) *synthesis*. 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 [13], [14], [15], [16], [17], [18], [19]. 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 is mathematically represented by a collection of variables, whose behavior is described by a temporal logic formula. A component can be studied in the context of an environment, which is represented by other variables, in which case the temporal logic formula describes the desired behavior of the component in the presence of its environment, for example how the component should respond to environmental changes.

In this paper, a contract is a collection of realizable component specifications that combined imply the specification we decomposed. We assume synchronous interaction of components that allows in each step at most one component to change in a non-unique way, described using a scheduler that ensures interleaving changes for other components (see also Remark 14 on page 33). 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. We study the problem of eliminating variables during the decomposition of an overall specification into component specifications that form an assume-guarantee contract (a top-down approach [20]). The number of components, and which variables represent each component are given as problem input. In order to detect which variables, from those that represent other components, 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]. Different interconnection architectures can be associated to each recurrence goal, in which case switching between them is controlled by one component, and can take multiple steps.

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) [14]. 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.

A simple example to demonstrate how decomposition works is the following. Consider a building with secure doors controlled by some central location. We want to be able to repeatedly enter and exit the building (overall system property), but we do not control the doors. In order for the building security controller to open a door, we should both swipe our card near the door we want opened, and stand in front of a camera. The security controller and the persons that enter and exit are the two components. An incorrect decomposition would be to require that if the doors repeatedly open, then we repeatedly enter and exit the building. Another incorrect decomposition would be to assume that swiping our card will lead to the door eventually opening. We need to assume that if we both swipe and stand in front of a door, then eventually the door will be opened (one component specification). The security controller is required to eventually respond to such a request by eventually opening the corresponding door (the other component specification). The decomposition approach we propose constructs liveness specifications of this form that ensure the overall objective, which is to repeatedly cross through doors.

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].

In summary, the contributions of this work are a decomposition algorithm that takes a specification and partition of variables as input and computes implementable specifications for the components represented by these variables, which avoids circular dependence between components, a parametric analysis for finding what variables can be hidden when specifying components that are implementable, an implementation of an algorithm for converting binary decision diagrams to minimal formulas that involve integer-valued variables and its application to specification construction, a formalization of realizability, and an operator for defining assume-guarantee properties from complete-system properties (closed systems).

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], [42, §6 on p. 532]. Stepwise implication in their work constrains the immediate future behavior of a system in case its environment behaved as assumed throughout the past. The increment of time between constraint and assumption enables assembling interdependent components without circular dependence. All properties up to this point were safety, and not expressed in temporal logic [43]. Two developments followed, and the work presented here is based on them.

The first was Lamport's introduction of *proof lattices* [20]. 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 of stepwise implication operators ($\overset{+}{-\!\triangleright}$ and variants) in temporal logic by Lamport [12], and Pnueli [38], 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], [54], [55]. In this context, contracts are used broadly, as an umbrella term that encompasses both interface theories and assume-guarantee contracts [56], [57], [53], [58], 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 [59], as well as a verification tool of contract refinement using an SMT solver [60]. This body of work focuses mainly on using and modifying existing contracts. We are interested in *constructing* contracts.

Decomposition of an assume-guarantee contract for an overall system into assume-guarantee contracts for components has been investigated in an approach that checks whether a candidate decomposition satisfies certain sufficient conditions, and if not amends the contracts in a sound way in search of a correct decomposition [61]. This approach is formulated generically for contract theories whose operators satisfy certain distributivity requirements, and is demonstrated in theories with trace-based and modal transition specifications.

An approach to architectural synthesis based on contracts of components available from a library has been studied in [62], [63]. In that approach, the components are automatically selected from an existing library, with the objective of creating an assembly that satisfies a given specification.

Contract theories in the framework of [53], [55] formulate the notion of contract as a pair of two assertions (properties) that represent a component and its intended environment. In our approach, each component is specified by a single temporal formula, which incorporates assumptions and guarantees implicitly, as a suitable form of implication (stepwise for safety, propositional for liveness) [42], [64], [65]. For example, the formula $\varphi \triangleq \Box\Diamond(a = 1) \Rightarrow \Diamond\Box(b = 2)$, is equivalent to $\Box\Diamond(b \neq 2) \Rightarrow \Diamond\Box(a \neq 1)$. Which one is intended as assumption, $\Box\Diamond(a = 1)$, or $\Box\Diamond(b \neq 2)$? Formally, we cannot distinguish without mentioning a separate formula other than φ. In other words, the formula $A \Rightarrow G$ describes one component, without describing an intended environment. Two formulas A and $A \Rightarrow G$ can describe two components. Our notion of contract refers to a collection of component specifications, and for the case of two specifications corresponds in descriptive capability to a pair of assertions as contract [53], [55], [66]. Also, we view a contract as an agreement that binds multiple components, whereas a pair of assertions in contract theories is an agreement that binds one component. Methodologically, in contract theories one checks that an assumption formula is fulfilled by another component's guarantee [67], whereas in our approach the conjoined component specifications should imply the desired overall specification.

The theory of synchronous relational interfaces [67] is an approach that allows expressing safety contracts, and reasoning about composition, refinement, and component substitutability. The refinement calculus of reactive systems (RCRS) [68] is a framework for describing components using monotonic property transformers that operate on sets of traces, and can describe safety and liveness properties. It is a typed formalism that distinguishes inputs from outputs, and represents constraints on the environment in a way that can be regarded as behavioral typing, supporting non-input-receptive representation of systems and type inference [69], [70]. We use an untyped logic, TLA^{+}, and suitable forms of implication to specify realizable open systems. Our approach is state-based, and how realizability is required, i.e., how quantifiers affect variables, indicates which variables are controlled by each component. Declaration of variables does not annotate them as inputs or outputs. Which variables are communicated to other components is determined by the decomposition algorithm. In our approach, (strictly) causal systems are specified using stepwise implication (in the operators $\stackrel{+}{-\!\!\triangleright}$ and $\mathit{WhilePlusHalf}$). Acausal specifications are unrealizable with our definition of realizability. RCRS is aimed mainly at verification and bottom-up synchronous composition of systems and their contracts from components, whereas in this work we are interested in decomposing specifications of an overall system.

FOCUS is a typed formalism based on stream processing functions [71], [72] which can express assume-guarantee specifications, open and closed systems, and supports reasoning about system composition and refinement.

Reactive modules [73] is another formalism for hierarchical specification and verification of systems, which supports assume-guarantee reasoning for both synchronous and asynchronous systems, temporal refinement, and state hiding.

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 [74], [75], [76]. Inability to observe external state changes makes a game asynchronous [22], [77]. 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 [14]. 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 [78]. To avoid this route alternative methods have been developed, using universal co-Büchi automata [18], or antichains [19].

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 [79], or restrict the specifications to limited fragments of LTL [80]. 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 [81]. Asynchronous distributed synthesis is undecidable [82], [83], [77]. In our approach we find interconnections of components that suffice for implementing the component specifications. Instead of selecting a specific interconnection, which leads to an undecidable problem, we search for an interconnection that suffices. This is a trade-off of decidability for optimality, in that the resulting interconnection can involve more information sharing between the components than may be necessary.

Besides synthesis of a distributed implementation, the more general notion of *assume-guarantee* synthesis [84] 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. Assumption synthesis has been used for the verification of existing modules by eliminating variables to abstract the modules, before reasoning about safety properties of their composition [85].

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 [86], and implemented symbolically [87]. Later work addressed synthesis by separating the construction of assumptions into safety and liveness [88]. The safety assumption is obtained by property *closure*, which also plays a key role in the composition theorem presented in [42]. Our work is based on this separate treatment of safety and liveness. Methods that use opponent strategies [89] to refine the assumptions of a GR(1) specification, searching over syntactic patterns were proposed in [90], [91]. The syntactic approach of [91] was used in [92] 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 [93].

Our work uses parametrization, based on ideas of approximating asynchronous with GR(1) synthesis [22], [16]. Another form of parametrization studied in the context of synthesis is that of safety and reachability goals [94]. Instead of hiding specific variables, an alternative approach in the context of verification [95] 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 [96]. A hierarchical approach where an observer for the continuous state is designed separately from synthesizing a discrete controller from temporal logic specifications [97], and decomposition of properties for synthesizing implementations have been studied in the context of aircraft management systems [98]. Layering as a method for structuring system design has been applied in the context of the DisCo method, which is based on TLA [99], [100], [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 [101], robot path planning among planar rectangles [102] and recently for simplifying enumerated robot controllers [103]. In the context of synthesis, prime implicants (used here for minimal covering) have been used for refining abstractions [17], and have been mentioned in the context of debugging specifications [104]. For theories more general than propositional logic there has been work on deriving prime implicants in the context of SMT solvers [105].

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) [106, §4], [107, 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 [108]. Every entity in TLA$^+$ is a set (also called a *value*). A function f is a set with the property that, for every $x \in \text{DOMAIN}\, f$, we know what value $f[x]$ is. Functions can be defined with the syntax

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

where $e(x)$ is some expression [10, p. 303, p. 71]. If a value f equals the function constructor that maps values in $\text{DOMAIN}\, f$

to the values obtained by function application, then it is a function

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

For any $x \notin \textsf{DOMAIN}\, f$, $f[x]$ 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 LAMBDA [109]. 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 [110], [111]. If $\exists x \,:\, P(x)$, then the expression $\textsf{CHOOSE}\; x \,:\, P(x)$ equals some value that satisfies $P(x)$. Otherwise, this expression is some unspecified value that can differ depending on P.

The operator $\land$ denotes conjunction, $\lor$ disjunction, $\neg$ negation. Conjunctions and disjunctions can be written vertically, for example

$$\begin{array}{ll} \land\; x = 1 & \text{A conjunct} \\ \land\; y > x & \text{another conjunct} \end{array}$$

Vertical lists of this kind can also be nested. Nat denotes the set of natural numbers [10, §18.6, p. 348], and for $i, j \in \mathit{Nat}$ the set of integers between i and j is denoted by

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

A function with domain $1..n$ for some $n \in \mathit{Nat}$ 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 $\textsf{LET}\; x \triangleq e_1 \;\textsf{IN}\;\; e$. A string is a sequence of characters, for example "ab". The expression f["r"] is also written $f.r$, for any string "r".

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 expressions 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". A behavior σ satisfies the formula $\Box f$, denoted $\sigma \models \Box f$, if every "tail" of σ satisfies f. More precisely, if

$$\forall n \in \mathit{Nat} \,:\, \sigma[n..] \models f$$

where $\sigma[n..] \triangleq [i \in \mathit{Nat} \mapsto \sigma[i+n]]$.

For example, if formula f is TRUE in every state of behavior σ, then $\sigma \models \Box f$ is TRUE. The formula $\Diamond f$ is TRUE if f is TRUE at some state of a behavior, i.e., $\Diamond f \triangleq \neg\Box\neg f$. Formal semantics are defined in [10, §16.2.4].

A property is a collection of behaviors described by a temporal formula. An example formula is shown in Fig. 4, which includes actions within temporal operators, and a state predicate as initial condition. 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⁺ the constructs [48, Prop. 2.1]

$$\begin{aligned} [A]_v &\;\triangleq\; A \lor (v' = v), \\ \langle A \rangle_v &\;\triangleq\; \neg[\neg A]_v = A \land (v' \neq v), \end{aligned}$$

where A is an action, are used to ensure stutter-invariance. So, $\Box[x' = x + 1]_x$ is satisfied by a behavior whose each step either increments x by one, or leaves x unchanged. TLA⁺ allows writing $\Box[A]_v$ and $\Diamond\langle A \rangle_v$, but not the above constructs alone, neither an action A outside them. The raw logic does not impose these requirements, so there we can write $\Box A$. We use the constructs $[A]_v$ and $\langle A \rangle_v$ also as shorthands for the formulas that define them. Boolean (and other) operators can

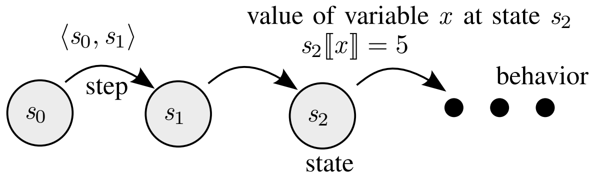


Fig. 3: Semantic concepts of the temporal logic TLA⁺.

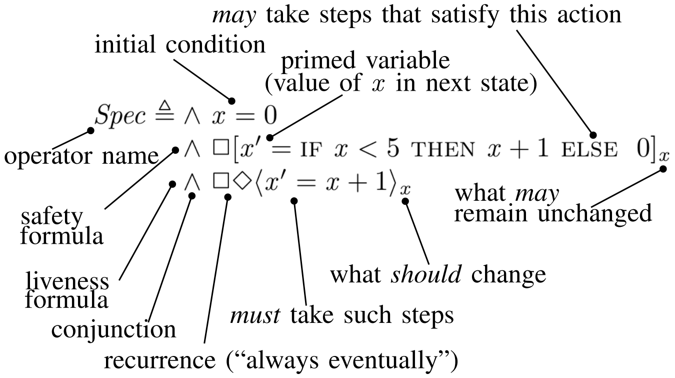


Fig. 4: Elements of formula and operator definition in TLA⁺.

be applied between any expressions, for example if A and B are actions, then the formula $A \land B$ is an action.

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 [112], [35, p. 49]. A property of the form $\Box\Diamond p$ ($\Diamond\Box p$) is called *recurrence* (*persistence*) [113].

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

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

C. Synthesis of implementations

Synthesis is the algorithmic construction of an implementation that satisfies the specification of a component, given as a temporal formula $Phi(x, y)$, where variable y represents the component, and variable x its environment. The purpose of an implementation is to produce the desired behavior for y, for example the output of a circuit with input x. An implementation (or realization) of $Phi(x, y)$ is formally a function f that changes y (so decides y'), depending on the current values of x, y, in a way that satisfies $Phi(x, y)$. In addition, an implementation can include "memory", which is internal state used to store useful information. If the variable mem represents this memory, then the function f depends on mem, and the implementation includes a function g that changes mem (so mem'), depending on x, y, mem. A temporal property $Phi(x, y)$ is called *realizable* if an implementation exists.

The notion of realizability [114], [115], [36] can be formalized [116] as shown in Fig. 5, which is based on a note by Lamport [8] (see also [12, p. 221]). Fig. 5 corresponds to realizability in the literature on synthesis [117, §4, pp. 46–47], [118, §2.3, pp. 914–915].

The operator $Realization$ describes a temporal property satisfied by behaviors where the variables mem, y start from $mem0, y0$ and change according to the functions f (for externally visible behavior) and g (for internal behavior). The expression $IsRealizable(Phi)$ means that the property Phi is implementable (feasible), in that $f, g, y0, mem0$ exist such that f, g have finite domains, and any behavior of x, y that satisfies $Realization$ also satisfies the given property Phi.

Later in our discussion we mention realizability with respect to different sets of variables. Formally this corresponds to writing different versions of Fig. 5, with for example y_1, y_2 and x_1 in one version, y_1, y_2, y_3 and x_2, x_3 in another, etc. Instead, we write $IsRealizable_{x_1;y_1,y_2}(Phi)$ to make explicit the variable names (see also Remark 18).

$$\begin{array}{lll} IsAFiniteFcn(f) & \triangleq & \land\, IsAFunction(f) \\ & & \land\, IsFiniteSet(\text{DOMAIN}\, f) \end{array}$$

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

$$\begin{array}{l} IsRealizable(Phi(\_,\, \_)) \;\triangleq \\ \quad \exists\, f,\, g,\, y0,\, mem0 : \\ \qquad \land\, IsAFiniteFcn(f) \land IsAFiniteFcn(g) \\ \qquad \land\, \text{LET}\;\; R(u,\, v) \;\triangleq\; Realization(\\ \qquad\qquad\qquad\qquad u,\, v,\, f,\, g,\, y0,\, mem0) \\ \qquad\quad \text{IN} \quad \boldsymbol{\forall}\, x,\, y :\; R(x,\, y) \Rightarrow Phi(x,\, y) \end{array}$$

Fig. 5: A definition of realizability. The operator $IsFiniteSet$ requires finite cardinality [10, p. 341].

Tractable liveness: A formula described by the schema

$$StreettPair \;\triangleq\; \bigvee\nolimits_{j\in 1..m}\Diamond\Box P_j \lor \bigwedge\nolimits_{i\in 1..n}\Box\Diamond R_i$$

defines a liveness property categorized as generalized Streett(1), or GR(1) [14]. A formula of this form is useful for expressing the dependence of a component on its environment. Rewriting the above as

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

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], [113]. Synthesis of a controller that implements a GR(k) property has computational complexity factorial in the number of Streett pairs k [15]. 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 is the number of states (in the worst case exponential in the number of variables) [118], [119], [120]. 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 [121], so exponentially more expensive than GR(1) synthesis [14]. For this reason GR(1) synthesis algorithms unconditionally add a memory variable that ranges over $1..n$.

D. Elements of synthesis algorithms

Fixpoint operators: Reasoning about open systems involves computing from which states a controller exists that can guide the system to a desired set of states (destination). The set of states from where such a controller exists is called an *attractor*, and results from iteratively solving a "one-step" control problem [75]. The "one-step" control problem involves finding from which states the system can reach a *Target* in one step that satisfies the action *SysNext*, assuming that the environment satisfies the action *EnvNext*. With x (y) a variable representing the environment (controller), the one-step control problem is described by the controllable step operator

$$\begin{aligned}&\mathit{Step}(x,\ y,\ \mathit{Target}(\_,\ \_)) \triangleq \exists\, y' :\\ &\quad \land\ \mathit{SysNext}(x,\ y,\ y')\\ &\quad \land\ \forall\, x' :\ \ \mathit{EnvNext}(x,\ y,\ x') \Rightarrow \mathit{Target}(x',\ y')\end{aligned}$$

(commonly denoted as *CPre*, see also Remark 9 on page 33). A state satisfies *Step* if x, y take values in that state such that the controller 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. An operator for the general case is given in Remark 13 on page 33.

The set of states (attractor) from where a controller exists that can guide the system to some state that satisfies $\mathit{Goal}(x, y)$ (destination) in at most k steps results by applying the *Step* operator k times. This computation is formalized as the operator *kStepAttractor* in Fig. 6 [120], [119], where m "counts" down from k to 0, and the applications of *Step* are "chained" together. The fixpoint that results from applying *Step* is formalized as the operator *Attractor* in Fig. 6.

Another computation we use later is finding the largest subset of a set of states *Stay* such that the component can either keep the behavior forever within *Stay*, or eventually let the behavior enter the set of states *Escape*. This computation is formalized as the operator *Trap* in Fig. 6, and can be thought of as finding a "waiting" area from where the environment can "service" the component's request.

Symbolic implementation: 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 [122]. This information is declared as type hints [123] 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.

Remark 1: The presentation here is in terms of TLA⁺. The approach and results are applicable also to other frameworks, for example other logics with linear-time semantics, with suitable adaptation. □

```
kStepAttractor(x, y, Goal(_, _), k) ≜
  LET RECURSIVE F(_, _, _)
      F(u, v, m) ≜ IF m = 0
          THEN Goal(u, v)
          ELSE LET Target(a, b) ≜ F(a, b, m − 1)
               IN  ∨ Target(u, v)
                   ∨ Step(u, v, Target)
  IN  F(x, y, k)

Attractor(x, y, Goal(_, _)) ≜
    LET
        Attr(u, v, n) ≜ kStepAttractor(u, v, Goal, n)
        r ≜ CHOOSE k ∈ Nat :  ∀ u, v :
            Attr(u, v, k) ≡ Attr(u, v, k + 1)
    IN
        Attr(x, y, r)

kStepSafe(x, y, Stay(_, _), Escape(_, _), k) ≜
  LET RECURSIVE F(_, _, _)
      F(u, v, m) ≜ IF m = 0
          THEN TRUE
          ELSE LET Safe(a, b) ≜ F(a, b, m − 1)
               IN  ∨ Escape(u, v)
                   ∨ ∧ Stay(u, v)
                     ∧ Step(u, v, Safe)
  IN  F(x, y, k)

Trap(x, y, Stay(_, _), Escape(_, _)) ≜
  LET
      Safe(u, v, n) ≜ kStepSafe(u, v, Stay, Escape, n)
      r ≜ CHOOSE k ∈ Nat :  ∀ u, v :
              Safe(u, v, k) ≡ Safe(u, v, k + 1)
  IN
      Safe(x, y, r)
```

Fig. 6: Operators used for solving games to implement a specification that involves interaction with an environment.

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.

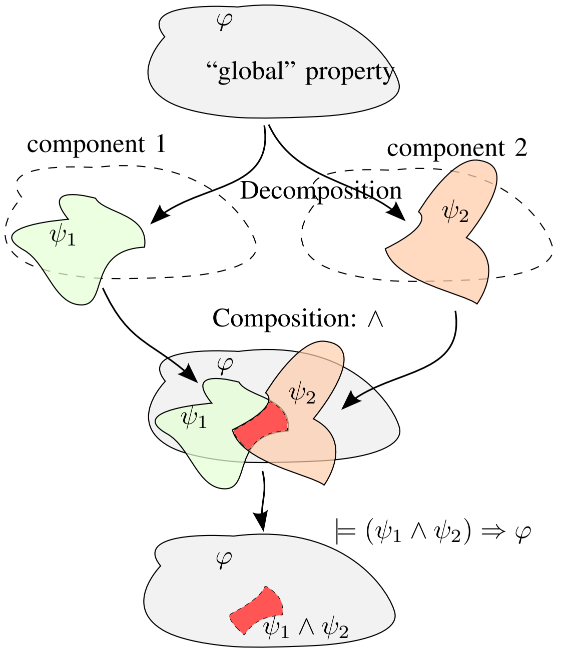


Fig. 7: Each component is specified by a property that may allow behaviors that violate the desired global property. These undesired behaviors would be caused by arbitrary behavior of other components. Nonetheless, the conjoined component specifications imply the global property, because of mutual fulfillment of assumptions between components.

These requirements are formalized with the following definition. A *contract* [124] for implementing a property *Phi* is a collection of n temporal properties, described by the operators

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

and a partition of variables $x, \dots, z$ among n components, such that

$$\begin{aligned} &\land\ \mathit{IsRealizable}_1(A) \land \dots \land \mathit{IsRealizable}_n(W) \\ &\land\ \big(A(x, \dots) \land \dots \land W(\dots, z)\big) \Rightarrow \mathit{Phi}(x, \dots, z) \end{aligned}$$

where $\mathit{IsRealizable}_i$ refers to a rewriting of Fig. 5 according to what variables represent component i (Remark 18 on page 33), the symbol _ represents operator arity [10, §17.1.1] (Remark 19). The properties $A(x, \dots), \dots, W(\dots, z)$ can depend on different subsets of variables. In other words, a contract is a collection of realizable properties that conjoined imply the desired behavior for the system assembled from components that implement these properties. Each property $(A, \dots)$ is described by a temporal formula that incorporates an assumption and a guarantee in a suitable implication (logical implication for the liveness part, stepwise implication for the safety part). Thus, each property is of an assume-guarantee form. For example, if the formula $(\Box\Diamond P) \Rightarrow \Box\Diamond Q$ specifies component F and $\Box\Diamond P$ component H, then we can informally call $\Box\Diamond P$ a guarantee of H (towards F), and an assumption of F about H. The notion of composition of properties is illustrated in Fig. 7.

```
EXTENDS Integers
VARIABLES spot_1, spot_2, free_x, free_y, free,
          req, pos_x, pos_y, occ, turn
station_vars ≜ ⟨spot_1, spot_2, free_x, free_y, free⟩
robot_vars ≜ ⟨req, pos_x, pos_y⟩
vars ≜ ⟨station_vars, robot_vars, occ, turn⟩
StationStep ≜ ∧ turn = 1
              ∧ (req = 0) ⇒ (free′ = 0)
StationNext ≜
    ∧ spot_1 ∈ 0 . . 1 ∧ spot_2 ∈ 0 . . 1 ∧ free ∈ 0 . . 1
    ∧ free_x ∈ 0 . . 18 ∧ free_y ∈ 0 . . 18
    ∧ ∨ (free = 0)
      ∨ (free_x = 1 ∧ free_y = 1 ∧ spot_1 = 0)
      ∨ (free_x = 2 ∧ free_y = 1 ∧ spot_2 = 0)
    ∧ (free = 1) ⇒ ∧ spot_1 = 0  ⇒  occ ≠ 1
                   ∧ spot_2 = 0  ⇒  occ ≠ 2
    ∧ StationStep ∨ UNCHANGED station_vars

RobotNext ≜
    ∧ pos_x ∈ 1 . . 15 ∧ pos_y ∈ 1 . . 15 ∧ req ∈ 0 . . 1
    ∧ ((req = 1 ∧ req′ = 0) ⇒ ∧ free = 1
                               ∧ free_x = pos_x′
                               ∧ free_y = pos_y′)
    ∧ (turn = 2) ∨ UNCHANGED robot_vars

OthersNext ≜ (occ ∈ 1 . . 3) ∧ (occ′ = occ)
SchedulerNext ≜ ∧ turn ∈ 1 . . 2
                ∧ (turn = 1) ⇒ (turn′ = 2)
                ∧ (turn = 2) ⇒ (turn′ = 1)

Env ≜ ∧ turn ∈ 1 . . 2  ∧ occ ∈ 1 . . 3
      ∧ □[OthersNext ∧ SchedulerNext]_vars
      ∧ □◇⟨SchedulerNext⟩_turn
Init ≜ ∧ spot_1 = 0 ∧ spot_2 = 0
       ∧ free_x = 0 ∧ free_y = 0 ∧ free = 0
       ∧ pos_x = 1 ∧ pos_y = 1 ∧ req = 0
Next ≜ StationNext ∧ RobotNext
L ≜ □◇(req = 0) ∧ □◇(req = 1)
Assembly ≜ Init ∧ □[Next]_vars ∧ L
Phi ≜ Env ⇒ Assembly
```

Fig. 8: Assembled-system specification for the charging station example (i.e., before decomposition).

Remark 2: The above notion of contract describes the obligations that bind each component (in analogy to an agreement among them). The above notion relates in two ways to a notion of contract as a pair of an assumption and a guarantee [55], [53], [54], [66]. The case of two properties above, e.g., A, B, can be thought of as describing an environment and a component, separately, and so to correspond to a pair of properties. On the other hand, each property above $(A, \dots)$ incorporates an assumption and a guarantee using a suitable form of implication, thus it has the nature of an agreement that binds one component. □

Example 1: As an example used throughout the paper, consider a charging station for mobile robots that has two charging spots, and a robot that requests a spot for charging. The robot is represented by its coordinates on the plane pos_x, pos_y. The charging station has two charging spots, with coordinates 1,1 and 2,1.

The charging station keeps track of which spots are taken (variables $spot\_1, spot\_2$, 0 if a spot is free), and sets the variables $free\_x, free\_y$ to the coordinates of the spot that becomes available for docking, and notifies the robot by setting the variable $free = 1$. The robot can request docking by setting the variable req. When $free = 0$, the variables $free\_x, free\_y$ do not communicate information and can take other values.

Not all spots are free. One other spot can be occupied by another robot, which forms part of the environment of the charging station and the robot. This spot is indicated by the variable $occ \in 1..3$ (3 means that the other robot is away from charging spots). 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 spot this is. Otherwise, assumptions about how occ can change need to be included, for example to prevent occ from changing to the same spot that the station has assigned to the first robot. The variable $turn$ is used to express the assumption that in each step either variables that represent the robot change and variables that represent the station remain unchanged, or vice versa. The specification of the entire system is shown in Fig. 8. This specification can be extended by adding more spots at different coordinates, for example a third spot with coordinates 10,12.

We wrote the property Phi 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 $Env \land Assembly$ (a closed system). In that case, we would have to consider components for the scheduler and other robots. Note that the property $Assembly$ defines synchronous and interleaving changes to the components.

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

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

We write the operators $PhiS, PhiR$ with arguments to indicate the dependence on variables that we will obtain in Sec. VI-D (see also Remark 20 on page 34).

Considering the liveness part of $PhiS$ and $PhiR$, one choice would be $\Box\Diamond((req = 1) \Rightarrow (free = 1 \land turn = 2))$ for $PhiS$, and $\Diamond\Box(free = 0 \land req = 1) \lor \big(\Box\Diamond(req = 0) \land \Box\Diamond(req = 1)\big)$ for $PhiR$. Examples of the properties $PhiS$ and $PhiR$ are given in later parts of this running example, as we incrementally describe the stages of automated decomposition. □

$$
\begin{array}{l}
\mathit{WhilePlusHalf}(A(\_,\,\_),\; G(\_,\,\_),\; x,\; y) \;\triangleq \\
\;\; \forall b : \;\; \lor \neg \land b \in \mathrm{BOOLEAN} \;\; \land \Box[b' = \mathrm{FALSE}]_b \\
\qquad\qquad\quad \land \exists u,\, v : \;\; \land A(u,\, v) \\
\qquad\qquad\qquad\qquad\qquad\; \land \Box \lor b \neq \mathrm{TRUE} \\
\qquad\qquad\qquad\qquad\qquad\qquad \lor \langle u,\, v\rangle = \langle x,\, y\rangle \\
\qquad\quad\; \lor \exists u,\, v : \;\; \land G(u,\, v) \land (v = y) \\
\qquad\qquad\qquad\qquad \land \Box \lor b \neq \mathrm{TRUE} \\
\qquad\qquad\qquad\qquad\quad \lor \langle u,\, v\rangle = \langle x,\, y\rangle \\
\qquad\qquad\qquad\qquad \land \Box[(b = \mathrm{TRUE}) \Rightarrow (v' = y')]_{\langle b,\, v,\, y\rangle}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Unzip}(P(\_,\,\_),\; x,\; y) \;\triangleq \\
\quad \mathrm{LET}\; Q(u,\, v) \;\triangleq\; P(v,\, u) \quad \text{swap back to } x, y \\
\qquad\quad\;\; A(u,\, v) \;\triangleq\; \mathit{WhilePlusHalf}(Q,\, Q,\, v,\, u) \quad \text{swap to } y, x \\
\quad \mathrm{IN} \quad \mathit{WhilePlusHalf}(A,\, P,\, x,\, y)
\end{array}
$$

Fig. 9: The operator $Unzip$ defines an open system from a closed system. The operator $WhilePlusHalf$ describes stepwise implication for the safety part of A, G, and implication for the liveness part.

B. Open systems

1) Defining an open system: Usually the components we build rely on their environment in order to operate as intended. Such a component that interacts with an environment is called an open system [47], [117, §3.1], [106, §9.5.3]. For example, a laptop should be able to connect 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 an open system can be defined mathematically by whether arbitrary behavior is allowed for any of the variables. Let x be a variable and $P(x)$ a temporal property. If $P(x)$ implies that x remains in some fixed set S throughout a behavior, then $P(x)$ describes a closed system, i.e.,

$$
\begin{array}{l}
\mathit{IsAClosedSystem}(P(\_)) \;\triangleq\; \exists S : \forall v : \;\; P(v) \Rightarrow \Box(v \in S) \\
\mathit{IsAnOpenSystem}(P(\_)) \;\triangleq\; \neg \mathit{IsAClosedSystem}(P)
\end{array}
$$

Note that $IsAnOpenSystem(P)$ is equivalent to $\forall S : \exists v : P(v) \land \Diamond(v \notin S)$. So the possibility of *diverging behavior* characterizes a system as open. In other words, a property P defines a closed system if it implies a type invariant that bounds all the variables that occur in P. Therefore, closed systems can be defined using Δ_0 formulas [125, p. 161]. Diverging behavior is also the main concept in how initial conditions affect realizability [116, Lemma 6, p. 12].

2) Specifying interaction with an environment: A component's specification should not constrain its environment, but

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

Fig. 10: Expressing the *WhilePlusHalf* operator in raw TLA$^+$ with past operators.

constrain the system as long as the environment behaves as assumed in the intended application [41], [12], [42], [64], [65]. This form of requirement is expressible with a formula that spreads implication incrementally over a behavior [42], [65], [126], [48], [64], [41] (stepwise implication). The operator *WhilePlusHalf* in Fig. 9 takes two temporal operators A, G. The property $A(x, y)$ that can be thought of as describing what we assume about the environment (assumption), and $G(x, y)$ what we require of the system (guarantee). *WhilePlusHalf* 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 (see also Remark 16). Specifying liveness using G, with A describing a safety property is not restrictive, similarly to $\overset{+}{-}\!\!\triangleright$ [42, §5.1].

We can use *WhilePlusHalf* to specify how an open system should behave. Writing a closed-system specification P for how component and environment should behave when assembled is typically easier than reasoning about how to split this into two properties A, G. One reason is avoiding realizability due to an unintended reason, a form of vacuity. Vacuity can arise if x is not constrained by G as much as it is in A. Another motivation for writing a closed-system specification is that we can then utilize liveness to simplify the specification (writing specifications that are not machine-closed [127]).

The operator *Unzip* serves this purpose, by taking a closed system described by P, and using it as G, whereas it passes as A a (safety) property weaker than G. So *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 (P), the component's next step should not definitely block the assembly, and the assembly should not have been blocked in the past.

Blocking the assembly means violating the safety part of P, by changing the values of some variables in a way that P can no longer be satisfied. When using more variables, we write $\mathit{Unzip}_{x_1, x_2; y_1}$ to signify changes analogous with those for $\mathit{IsRealizable}_{x_1, x_2; y_1}(\mathit{Phi})$ (Sec. III-C).

3) Synthesis from open system properties: For the synthesis of implementations for properties specified using *Unzip*, relating this operator to existing results about synthesis from GR(1) properties is useful [118]. This is possible by turning temporal quantification ($\boldsymbol{\exists}$) to rigid quantification ($\exists$), using the operator *RawWhilePlusHalf* defined in Fig. 10. 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 and the action *Next* in that step and later ones. The operator *Earlier* abbreviates the composition of the past LTL operators *WeakPrevious* and *Historically* [128].

It can be shown that if $P \equiv \mathit{Init} \land \Box[\mathit{Next}]_{\langle x, y\rangle} \land L$, where L a GR(1) liveness property, and the pair of properties $\mathit{Init} \land \Box[\mathit{Next}]_{\langle x, y\rangle}, L$ is machine-closed (meaning that L does not constrain the safety in the property $\mathit{Init} \land \Box[\mathit{Next}]_{\langle x, y\rangle}$ [42]), then an implementation synthesized for the property

$$\mathit{RawWhilePlusHalf}(\mathit{Init}, [\exists y' : \mathit{Next}]_x, [\exists x' : \mathit{Next}]_y, [\mathit{Next}]_{\langle x, y\rangle}, L, x, y)$$

using existing algorithms [118] also realizes $\mathit{Unzip}(P, x, y)$.

Remark 3: Open systems can be defined also in other ways, for example using game graphs together with liveness formulas [117], game structures, alternating-time temporal logic formulas [74], or modules [129]. The property P corresponds to the graph and liveness in a game graph description. □

Remark 4 (Related operators): 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]). How *Unzip* is defined is reminiscent of how $\overset{+}{-}\!\!\triangleright$ is defined for safety properties in terms of $-\!\!\triangleright$ [48, p. 262], [126, Prop. 1, p. 501]. The operator *RawWhilePlusHalf* is a modification of [65] to avoid circularity [124], [64, §5, $\triangleright$ on p. 59]. □

Remark 5 (Symmetry): Dijkstra requires symmetry from solutions to the mutual exclusion problem [130, item (a)]. The approach we follow asserts that all components are implemented as "Moore machines" (the functions f and g in Fig. 5 are independent of primed variable values). Alternatives are possible where one component is Mealy and its environment Moore [118], [89]. 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 [75]. □

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 $P \ \triangleq\ \mathit{Env} \land \mathit{Assembly}$ 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. □

The concept of interleaving means that in each step the state of at most one component changes [10, p. 137]. If there are only two components, represented by the variables x and y, then property $\mathit{Phi}(x, y)$ is interleaving if it implies that x and

y do not both change at once [42, p. 514], [48], [116, §5, p. 22] (in TLA$^+$ $\mathit{Phi}(x, y) \Rightarrow \Box[(x' = x) \vee (y' = y)]_{\langle x,y\rangle}$)

$$\mathit{Phi}(x, y) \Rightarrow \Box((x' = x) \vee (y' = y))$$

We can relax interleaving to allow multiple components to change their state in one step, but multiple changes be possible for at most one. This relaxed notion is useful for including a scheduler that changes in a unique way in each (nonstuttering) step. We consider specifications that are interleaving in this way, allowing in each step non-unique changes for at most one component. 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].

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.

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.

Let $P(x_1, \dots, y_1, \dots)$ be a closed-system property of the form $\mathit{Init} \wedge \Box[\mathit{Next}]_{vrs} \wedge \mathit{Liveness}$, where $\mathit{Liveness}$ is a GR(1) property, and $\varphi(x_1, \dots, y_1, \dots) \triangleq \mathit{Unzip}_{x_1,\dots;y_1,\dots}(P, x_1, \dots, y_1, \dots)$. We are interested in finite-state specifications, i.e., P allows $x_1, \dots, y_1, \dots$ to take values from a finite set. So φ is an open-system GR(1) property. We assume that the specification includes a scheduler that ensures turn-based interleaving changes for other components, as discussed earlier, and that the variable that represents the scheduler remains visible to the components.

Problem 1 (Hiding variables): Assume that φ (defined above) is realizable by an implementation that can read the variables $x_1, \dots$, i.e., $\mathit{IsRealizable}_{x_1,\dots;y_1,\dots}(\varphi)$. Find those subsets of variables $x_a, x_b, \dots$ such that φ remain realizable by an implementation that can read only $x_a, x_b, \dots, y_1, \dots$, i.e., $\mathit{IsRealizable}_{x_a,x_b,\dots;y_1,\dots}(\varphi)$.

Exactly solving Problem 1 is computationally hard, because it requires reasoning about realizability of implementations with partial information about their environment. For this reason, in this section we develop a sound approach for selecting which variables to hide, i.e., we may select more $x_a, x_b, \dots$ than the minimal mumber necessary. We *parametrize the selection* of which variables to hide, by modifying the controllable step operator, which is used in later sections to *construct* a property φ that is realizable.

Synthesis of component implementations with partial information is computationally hard, so we eliminate the hidden variables from φ, using universal quantification, and express the component specifications with formulas in which hidden variables do not occur.

Problem 2 (Expressibility): Given a temporal operator φ as in Problem 1, the set of variables $x_a, \dots$ from $x_1, \dots$, and the remaining of those variables as $x_p, \dots$, express the formula $\forall x_p, \dots : \varphi(x_1, \dots)$ as a quantifier-free formula.

This step requires quantifier elimination. Realizability remains unchanged, due to how $x_a, \dots$ are selected above. The resulting specification ψ is realizable with full information because it mentions variables $x_a, \dots$ and does not mention $x_p, \dots$.

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 of the form

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

where the conjunct $\mathit{Liveness}$ is a conjunction of recurrence properties, for example $\Box\Diamond \mathit{Goal}_1 \wedge \Box\Diamond \mathit{Goal}_2$. The property $\mathit{Liveness}$ can impose safety constraints when conjoined to the property $\mathit{SM} \triangleq \mathit{Init} \wedge \Box[\mathit{Next}]_{vrs}$. If this is not the case, then the pair of properties $\mathit{SM}, \mathit{Liveness}$ is called *machine-closed* [131, p. 261], [42, p. 519]. We decompose the safety and liveness parts of the specification separately. After decomposition, each component specification will contain only "pieces" of liveness constraints. So the safety part should be strong enough to prevent any component from "straying away" to an extent that would violate the property $\mathit{Liveness}$.

The property SM may be too permissive to ensure that $\mathit{Liveness}$ 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 $\mathit{Assembly}$, i.e., such that

$$\models \mathit{Assembly} \Rightarrow W.$$

The property W is known as *closure* of the property $\mathit{Assembly}$ [132, p. 120], [42, p. 518], [48, pp. 261–262], [133, Fig. 2] due to topological considerations [112] (see also Remark 17 on page 33). If W is written in the form

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

then the invariant Inv defines the largest set of states that can occur in any behavior that satisfies the property $\mathit{Assembly}$. The weakest invariant yields also the (unique) weakest safety assumption necessary in turn-based games with full information (set of cooperatively winning states) [88], [124, §III-A].

Computing Inv for recurrence properties is straightforward [25], [26] and shown for completeness in Algo. 11. The case of GR(1) liveness is similar [14]. Let $vars$ be a set of variable names. Let $init, next$ be BDDs that represent the state predicate I and the action N, which depend on only primed and unprimed $vars$. Let $goals$ be a tuple of BDDs for the state predicates $G_1, \dots, G_n$. Let $Goals \triangleq \bigwedge_j \Box\Diamond G_j$. Let $R \triangleq Init \land \Box N \land Goals$. The procedure LIVESTATES computes the set of states from where the recurrence goals can be repeatedly visited. The procedure REACHABLESTATES computes the set of states that can be reached from the initial condition. Their intersection yields the set of states that are both reachable from initial conditions and from where the recurrence goals can be repeatedly visited. The procedure PRIMEIN$(vars, u)$ substitutes primed for unprimed identifiers in u, for each (unprimed) identifier in $vars$. The procedure UNPRIMEIN performs the reverse operation. The procedure PRIME$(vars)$ returns the set resulting from priming each (unprimed) identifier in $vars$. The procedure EXIST$(vars, u)$ existentially quantifies the variables $vars$ in the BDD u.

Theorem 3 (Closure): ASSUME : Let Inv be the state predicate returned (as a BDD) from the call CLOSURE$(init, next, goals, vars)$ of the procedure CLOSURE defined in Algo. 11. PROVE : The closure of the property R is equivalent to the property $Q \triangleq Init \land \Box Inv \land \Box N$.

In the case of the property $Assembly$ from above, the action N is $[Next]_{vrs}$.

Example 3: Fig. 12 illustrates the state predicate Inv that results from closure with respect to the liveness goal $\Box\Diamond(s_3 \lor s_7)$, and with node s_1 as initial condition. The recurrence goal is attainable from the nodes s_1, s_2, s_3, s_6, s_7, but not from the nodes s_4, s_5 [133, Fig. 2]. So s_4, s_5 are not included in Inv. Even though $s_3 \lor s_7$ can be repeatedly visited starting from nodes s_6, s_7, these nodes are unreachable from the initial condition, thus not included in Inv. □

Example 4: For the charging station example, the assembly invariant Inv (when Env holds) is

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

This invariant was symbolically computed, and the resulting BDD was then converted to a minimal formula in disjunctive normal form, with constraints on integer variables as conjuncts. □

Algo. 11: Computing the closure of a temporal logic formula of the form $I \land \Box N \land \bigwedge_j \Box\Diamond G_j$. lambda y can be thought of as an "anonymous" procedure that takes y as argument.

```
def CLOSURE($init, next, goals, vars$) :
    $r$ := REACHABLESTATES($init, next, vars$)
    $z$ := LIVESTATES($next, goals, vars$)
    return $z \land r$

def LIVESTATES($next, goals, vars$) :
    $operator$ := lambda $y$ : PREIMAGE($y, next, vars$)
    $z$ := TRUE
    $zold$ := CHOOSE $r$ : $r \neq z$
    while $z \neq zold$ :
        $zold$ := $z$
        $zpre$ := $operator(zold)$
        for $goal \in goals$ :
            $target$ := $zpre \land goal$
            $z$ := $z \land$ LEASTFIXPOINT($operator, target$)
    return $z$

def REACHABLESTATES($init, next, vars$) :
    $operator$ := lambda $y$ : IMAGE($y, next, vars$)
    return LEASTFIXPOINT($operator, init$)

def LEASTFIXPOINT($operator, target$) :
    $y$ := $target$
    $yold$ := CHOOSE $r$ : $r \neq y$
    while $y \neq yold$ :
        $yold$ := $y$
        $y$ := $y \lor operator(y)$
    return $y$

def IMAGE($source, action, vars$) :
    $u$ := $source \land action$
    $u$ := EXIST($vars, u$)
    return UNPRIMEIN($vars, u$)

def PREIMAGE($target, action, vars$) :
    $primed\_target$ := PRIMEIN($vars, target$)
    $u$ := $action \land primed\_target$
    $qvars$ := PRIME($vars$)
    return EXIST($qvars, u$)
```

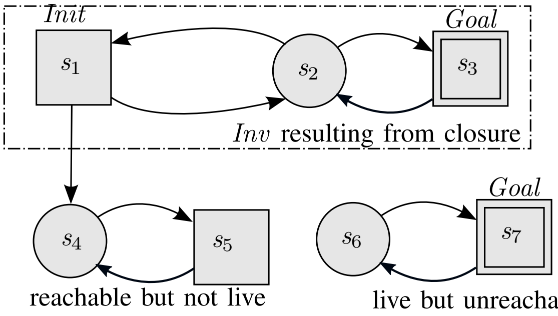


Fig. 12: An example of the state predicate Inv that results from closure with respect to the recurrence goal $\Box\Diamond(s_3 \lor s_7)$.

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 $\mathit{WInit} \land \Box[\mathit{WNext}]_{vrs}$, where [106, by INV2, Fig. 5, p. 888]

$$\begin{aligned} \mathit{WInit} &\triangleq \mathit{Inv} \land \mathit{Init} \\ \mathit{WNext} &\triangleq \mathit{Inv} \land \mathit{Next} \land \mathit{Inv}'. \end{aligned}$$

The property $\mathit{Unzip}(W, x, y)$ is defined using WNext and the following actions as arguments of the operator $\mathit{RawWhilePlusHalf}$

$$\begin{aligned} \mathit{EnvNext}(x,\, y,\, x') &\triangleq [\exists\, y' :\ \mathit{WNext}]_x \\ \mathit{SysNext}(x,\, y,\, y') &\triangleq [\exists\, x' :\ \mathit{WNext}]_y \end{aligned}$$

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 WNext is satisfied by each step. The reason is that in each step, at most one component can change in a non-unique way.

Remark 6: 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 $\Box\Diamond\langle \mathit{SchedulerNext}\rangle_{turn}$ 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

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

We will use $\mathit{Inv}(h, x, y)$ as a bound on h. It will be the case that $\models \mathit{Bound}(h, x, y) \Rightarrow \exists y' : P(h, x, y, y')$. 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)

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

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

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

Algebraic manipulation yields

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

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

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

By defining

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

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

we obtain

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

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. This soundly solves Problem 2. 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 5: Consider a 3 by 2 grid, with cell coordinates given by $y \in 1..3$ and $h \in 1..2$, where the component controls y and the environment h, and these variables can change in an interleaving way (in one step y may change, in the next step h may change, and so on, as determined by variable turn, similarly to Fig. 8). The actions are $\mathit{SysNext} \triangleq (\mathit{turn} = 1) \land (y \in 1..3) \land (y' \in 1..3) \land (y - 1 \leq y') \land (y' \leq y + 1)$ and $\mathit{EnvNext} \triangleq (\mathit{turn} = 2) \land (h \in 1..2) \land \big((h' = h) \lor (h' = 2)\big)$.

If the objective is $\Box\Diamond(y = 3)$, then the component does not need to know the value of variable h. During the attractor computation for reasoning about realizability, the controllable step operator will have $y = 3$ as first target, then $y \in 2..3$, and the fixpoint will be $y \in 1..3$. In this case, the variable h can be hidden from the component.

In contrast, the objective $\Box\Diamond(y = h)$ is unrealizable without variable h being visible to the component, because the choice of y' depends on the value of h. □

Example 6: 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}
\land\, \mathit{turn} = 2 \land \mathit{turn}' = 1 \land \mathit{free} \in 0 \mathrel{..} 1 \land \mathit{free\_x} \in 0 \mathrel{..} 18 \\
\land\, \mathit{free\_y} \in 0 \mathrel{..} 18 \land \mathit{occ} \in 1 \mathrel{..} 3 \land \mathit{occ}' \in 1 \mathrel{..} 3 \\
\land\, \mathit{pos\_x} \in 1 \mathrel{..} 15 \land \mathit{pos\_x}' \in 1 \mathrel{..} 15 \land \mathit{pos\_y} \in 1 \mathrel{..} 15 \\
\land\, \mathit{pos\_y}' \in 1 \mathrel{..} 15 \land \mathit{req} \in 0 \mathrel{..} 1 \land \mathit{req}' \in 0 \mathrel{..} 1 \\
\land\, \mathit{spot\_1} \in 0 \mathrel{..} 1 \land \mathit{spot\_2} \in 0 \mathrel{..} 1 \\
\land \quad \lor \quad \land\, (\mathit{free} = 1) \land (\mathit{free\_x} \in 0 \mathrel{..} 1) \land (\mathit{occ} = 3) \\
\qquad\qquad \land\, (\mathit{occ}' = 3) \land (\mathit{pos\_x}' = 1) \land (\mathit{pos\_y}' = 1) \\
\qquad \lor \quad \land\, (\mathit{free} = 1) \land (\mathit{free\_x} \in 2 \mathrel{..} 18) \land (\mathit{occ} \in 2 \mathrel{..} 3) \\
\qquad\qquad \land\, (\mathit{occ}' = 3) \land (\mathit{pos\_x}' = 2) \land (\mathit{pos\_y}' = 1) \\
\qquad \lor \quad \land\, (\mathit{free} = 1) \land (\mathit{occ} \in 1 \mathrel{..} 2) \land (\mathit{occ}' = 1) \\
\qquad\qquad \land\, (\mathit{pos\_x}' = 2) \land (\mathit{pos\_y}' = 1) \land (\mathit{spot\_2} = 0) \\
\qquad \lor \quad \land\, (\mathit{free} = 1) \land (\mathit{occ} \in 1 \mathrel{..} 2) \land (\mathit{occ}' = 2) \\
\qquad\qquad \land\, (\mathit{pos\_x}' = 1) \land (\mathit{pos\_y}' = 1) \land (\mathit{spot\_1} = 0) \\
\qquad \lor\, (\mathit{occ} = 1) \land (\mathit{occ}' = 1) \land (\mathit{req} = 0) \\
\qquad \lor\, (\mathit{occ} = 1) \land (\mathit{occ}' = 1) \land (\mathit{req}' = 1) \\
\qquad \lor\, (\mathit{occ} = 2) \land (\mathit{occ}' = 2) \land (\mathit{req} = 0) \\
\qquad \lor\, (\mathit{occ} = 2) \land (\mathit{occ}' = 2) \land (\mathit{req}' = 1) \\
\qquad \lor\, (\mathit{occ} = 3) \land (\mathit{occ}' = 3) \land (\mathit{req} = 0) \\
\qquad \lor\, (\mathit{occ} = 3) \land (\mathit{occ}' = 3) \land (\mathit{req}' = 1) \\
\land\, \mathit{InvH}
\end{array}
$$

After hiding the robot's coordinates $\mathit{pos\_x}, \mathit{pos\_y}$, the environment action is simplified to

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

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

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

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

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

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 $m = \textsf{FALSE}$, meaning h visible)

$$
\begin{array}{l}
\mathit{MaskedInv}(h,\, v,\, x,\, y,\, m) \triangleq \textsf{LET } r \triangleq \mathit{Mask}(m,\, v,\, h) \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \textsf{IN } \quad \mathit{Inv}(r,\, x,\, y) \\
\mathit{PrmInv}(v,\, x,\, y,\, m) \triangleq \exists\, h : \ \mathit{MaskedInv}(h,\, v,\, x,\, y,\, m) \\
R(v,\, x,\, y,\, m) \triangleq \\
\qquad \exists\, y' : \forall\, x',\, v' : \ \forall\, h : \\
\qquad\quad \textsf{LET } r \triangleq \mathit{Mask}(m,\, v,\, h) \\
\qquad\quad \textsf{IN } \quad \lor\, \neg \mathit{Inv}(r,\, x,\, y) \\
\qquad\qquad\quad\ \ \lor\, \land\, \mathit{SysNext}(r,\, x,\, y,\, y') \\
\qquad\qquad\qquad\quad \land\, \lor\, \neg \mathit{EnvNext}(r,\, x,\, y,\, v',\, x') \\
\qquad\qquad\qquad\qquad \lor\, \mathit{Target}(v',\, x',\, y',\, m) \\
\mathit{PrmStep}(v,\, x,\, y,\, m) \triangleq \\
\qquad \land \quad \mathit{PrmInv}(v,\, x,\, y,\, m) \\
\qquad \land \quad R(v,\, x,\, y,\, m)
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{PrmStep}(v,\, x,\, y,\, m) \equiv \\
\quad \textsf{LET} \\
\qquad \mathit{MskInv}(h) \triangleq \\
\qquad\quad \textsf{LET } r \quad \triangleq \mathit{Mask}(m,\, v,\, h) \\
\qquad\quad \textsf{IN } \quad \mathit{Inv}(r,\, x,\, y) \\
\qquad \mathit{PrmInv} \triangleq \exists\, h : \ \mathit{MskInv}(h) \\
\\
\qquad \mathit{MskSysNext}(h,\, y') \triangleq \\
\qquad\quad \textsf{LET } r \triangleq \mathit{Mask}(m,\, v,\, h)
\end{array}
$$

$$\begin{array}{l}
\qquad \textsf{IN} \quad \mathit{SysNext}(r,\, x,\, y,\, y') \\
\quad \mathit{PrmSysNext}(y') \;\triangleq \\
\qquad \land\; \mathit{PrmInv} \\
\qquad \land\; \forall\, h :\; \mathit{MskInv}(h) \Rightarrow \mathit{MskSysNext}(h,\, y') \\
\\
\quad \mathit{MskEnvNext}(h,\, v',\, x') \;\triangleq \\
\qquad \textsf{LET}\; r \;\triangleq\; \mathit{Mask}(m,\, v,\, h) \\
\qquad \textsf{IN} \quad \mathit{EnvNext}(r,\, x,\, y,\, v',\, x') \\
\quad \mathit{PrmEnvNext}(v',\, x') \;\triangleq \\
\qquad \exists\, h :\; \mathit{MskInv}(h) \land \mathit{MskEnvNext}(h,\, v',\, x') \\
\textsf{IN} \\
\quad \mathit{PrmStep}(v,\, x,\, y,\, m) \;\triangleq\; \exists\, y' :\; \forall\, x',\, v' : \\
\qquad \land \quad \mathit{PrmSysNext}(y') \\
\qquad \land \quad \mathit{PrmEnvNext}(v',\, x') \Rightarrow \mathit{Target}(v',\, x',\, y',\, m)
\end{array}$$

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 *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 that 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 above approach soundly solves Problem 1.

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 [94]. 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.

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. Each component is represented by a collection of variables, whose behavior is specified by a temporal property that can mention also other variables that represent the component's environment. 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

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

where the chaining is established by the implications

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

Conjoining the above specifications, we can deduce the recurrence properties

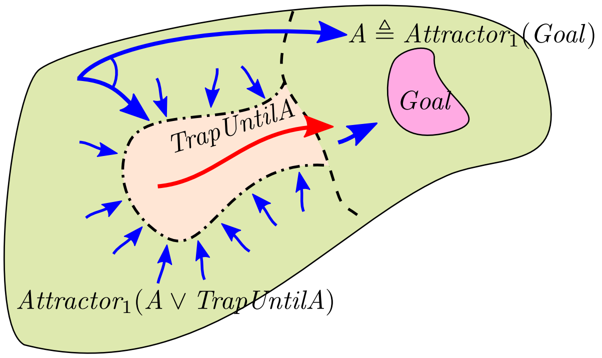
$$L \;\triangleq\; \Box\Diamond R_1 \land \Box\Diamond R_2 \land \Box\Diamond R_3.$$

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" [42], [126]. Liveness "assumptions" are meaningful in the context of multiple components, specified by multiple temporal properties, a situation similar to possibility properties [134], [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 $F \overset{+}{\rhd} G$ is a safety property [48, §5.2, p. 261]).

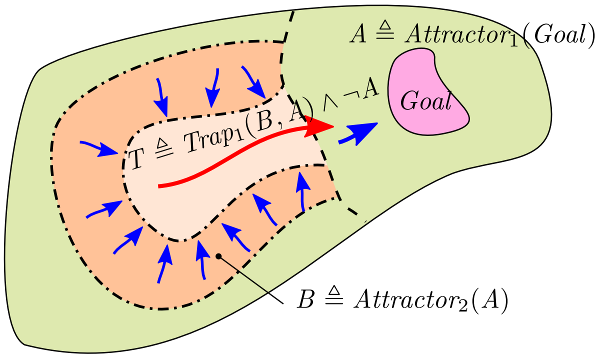
A simple but necessary property of the specifications L_1, L_2, L_3 is the acyclic arrangement of the reasoning that derives L [38], forming a *proof lattice* [44]. Mutual dependence of safety properties is admissible due to how implication is spread in a "stepwise" fashion over a behavior, as with the operator *WhilePlusHalf* (Fig. 9). 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].

All the discussion that follows focuses on liveness and omits the safety part of specifications. Safety is addressed by closure



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

Fig. 13: The basic idea of the approach.



$A \triangleq \mathit{Attractor}_1(\mathit{Goal})$
Goal
$T \triangleq \mathit{Trap}_1(B, A) \wedge \neg A$
$B \triangleq \mathit{Attractor}_2(A)$

Fig. 14: How traps are constructed (simple case).

and computation of component actions as described in Sec. V (see also Appendix B). 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. 13. 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 $\mathit{TrapUntilA}$ from where component 2 can reach A *and* component 1 can keep the assembly inside $\mathit{TrapUntilA}$ until A is reached. We can then write the liveness specifications

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

Property L_2 is realizable by component 2 (because it can reach A, which is outside $\mathit{TrapUntilA}$). 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 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) [124]. 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

$$B \triangleq \mathit{Attractor}_2(A).$$

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

$$C \triangleq \mathit{Trap}_1(B, A).$$

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

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

These sets are illustrated in Fig. 14. Letting $\mathit{TrapUntilA} \triangleq T$, 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 basic decomposition problem we are interested in, in the presence of full information, is the following (we use two variables for brevity, the statement generalizes to multiple variables). A more detailed statement that includes safety and an environment is given in Appendix B.

Problem 4: Let variable x represent component 1, and variable y component 2. Let $P(x, y)$ be a (satisfiable) closed-system property with $\Box\Diamond \mathit{Goal}(x, y)$ as liveness. Assume that P in each step allows non-unique changes to at most one of x, y.

Algo. 15: Basic algorithm for decomposing a recurrence goal, in the presence of full information. Components 1 and 2 are *Player*, *Team*, respectively.

```
def DecomposeGoal($G$, $Player$, $Team$) :
    $Traps := list()$
    $Y := G$
    $Yold :=$ CHOOSE $r : r \neq Y$
    while $Y \neq Yold$ :
        $Yold := Y$
        $A, T :=$ MakeAssumption($Y$, $Player$, $Team$)
        $Traps.append(T)$
        $Y := A \vee T$
    return $Y$, $Traps$

def MakeAssumption($Goal$, $Player$, $Team$) :
    $A := Attr(Goal, Player)$
    $B := Attr(A, Team)$
    $T := Trap(B, A, Player) \wedge \neg A$
    return $A$, $T$
```

Find temporal properties ψ_1, ψ_2 with GR(1) liveness such that $IsRealizable_{y;x}(\psi_1) \wedge IsRealizable_{x;y}(\psi_2)$ and $\big(\psi_1(x,y) \wedge \psi_2(x,y)\big) \Rightarrow P(x,y)$.

Theorem 5: ASSUME : Algo. 15 returns a Y such that $Inv \Rightarrow Y$, and Inv is satisfiable, where Inv as in Sec. V.

PROVE : The properties $\psi_1(x,y), \psi_2(x,y)$, with liveness $(\Box\Diamond Goal(x,y)) \vee \bigvee_i \Diamond\Box T_i(x,y)$ and $\bigwedge_i \Box\Diamond\neg T_i(x,y)$, respectively, solve Problem 4, where T_i are the traps returned by Algo. 15.

The above result can be applied also to other approaches (e.g., based on raw TLA$^+$ or LTL), with suitable changes to *IsRealizable* and the safety part of specifications.

This algorithm derives from an earlier version for the case without hidden variables [124], [135]. Covering all initial conditions of the assembly is not possible in general [124, §III-B], unless either safety is restricted [136, §V], or a syntactic fragment larger than GR(1) is used [124, §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 [124, Prop. 6] indicate that it is better for the specifier to reconsider the specification.

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 7: Consider the graph shown in Fig. 16. Component 1 chooses the next node when at a disk, and component 2 when at a box. A trap is found for Fig. 16, because component 2 can reach A from both nodes s_3 and s_4. Fig. 17 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

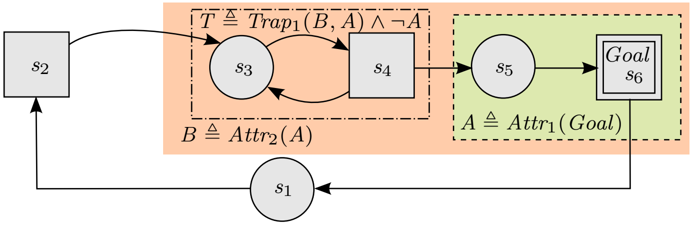

s_2
$T \triangleq Trap_1(B, A) \wedge \neg A$
s_3
s_4
s_5
$Goal$
s_6
$B \triangleq Attr_2(A)$
$A \triangleq Attr_1(Goal)$
s_1

Fig. 16: An example where a trap is found.

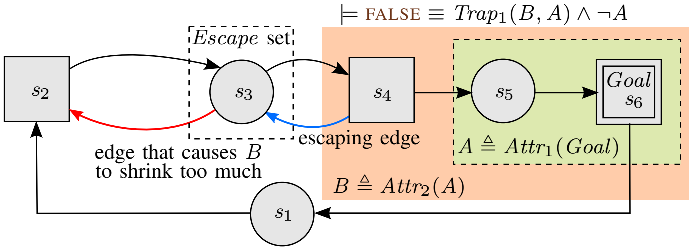

$\models$ FALSE $\equiv Trap_1(B, A) \wedge \neg A$
$Escape$ set
s_2
s_3
s_4
s_5
$Goal$
s_6
escaping edge
edge that causes B to shrink too much
$A \triangleq Attr_1(Goal)$
$B \triangleq Attr_2(A)$
s_1

Fig. 17: The simple approach cannot find a trap in this example. Compared to Fig. 16, the failure is due to the edge $\langle s_4, s_3\rangle$.

2 can move "backwards" from s_4 to s_3. So a larger B is needed, but why did B shrink compared to Fig. 16?

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).

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. 18. Let the state predicate $Escape$ mean that the current node is s_3. Define

$$Basin \triangleq B \vee Escape$$

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

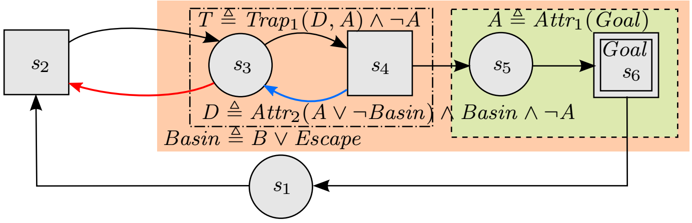
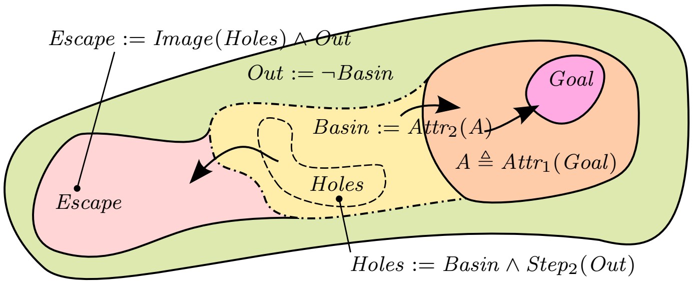

$T \triangleq Trap_1(D, A) \wedge \neg A$
$A \triangleq Attr_1(Goal)$
s_2
s_3
s_4
s_5
$Goal$
s_6
$D \triangleq Attr_2(A \vee \neg Basin) \wedge Basin \wedge \neg A$
$Basin \triangleq B \vee Escape$
s_1

Fig. 18: Including the states where component 2 can escape allows finding the trap suggested by the specifier's intuition.



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

Fig. 19: Collecting escapes that can cause a trap set to not form.

implication

$$T \Rightarrow Basin$$

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

$$(\neg Basin) \Rightarrow \neg T$$

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

$$D \quad \triangleq \quad Attr_2(A \lor \neg Basin) \land \neg(A \lor \neg Basin)$$

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. 18. The larger attractor D enables a trap to form; the set of states

$$T \quad \triangleq \quad Trap_1(D, A) \land \neg A$$

is nonempty. Moreover, $Attr_1(T \lor A)$ covers all nodes. So the components can realize the properties

$$\begin{aligned} \text{Component 2}: \quad & L_2 \quad \triangleq \quad \Box\Diamond\neg D \\ \text{Component 1}: \quad & L_1 \quad \triangleq \quad \Diamond\Box T \lor \Box\Diamond Goal \end{aligned}$$

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. 19 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)

$$Basin \ := Attr_2(A).$$

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

$$Holes \ := Basin \land Step_2(\neg Basin).$$

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

$$Escape \ := Out \land Image(Holes)$$

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

Algo. 20: Extended algorithm that finds decompositions in more cases. The procedure DECOMPOSEGOAL is the same with Algo. 15 and is not shown here.

```
def MakeAssumption($Goal, Player, Team$) :
    $A \ := Attr(Goal, Player)$
    $Basin \ := Attr(A, Team)$
    $Escape \ := \text{TRUE}$
    while $(\neg \models Escape \equiv \text{FALSE})$ :
        $Out \ := \neg Basin$
        $Holes \ := Basin \land Step(Out, Team)$
        $Escape \ := Out \land Image(Holes \land Inv, Team)$
        $Basin \ := Basin \lor Escape$
        $TeamGoal \ := A \lor \neg Basin$
        $D \ := Attr(TeamGoal, Team) \land Basin$
        $T \ := Trap(D, A, Player) \land \neg A$
    return $A, T$
```

$$\begin{aligned} & Image(x,\ y,\ Source(\_,\ \_),\ Next(\_,\ \_,\ \_,\ \_)) \ \triangleq \\ & \quad \exists\, p,\ q:\ Source(p,\ q) \land Next(p,\ q,\ x,\ y) \end{aligned}$$

The resulting $Basin$ is used for computing $D := Attr_2(A \lor \neg Basin) \land Basin \land \neg A$ and $Trap_1(D, A) \land \neg A$. 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.

Theorem 6: Algo. 20 solves Problem 4, similarly to Theorem 5, in more cases than Algo. 15.

Fig. 18 is a problem where Algo. 20 returns a decomposition, whereas Algo. 15 does not.

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. The operator $Maybe$ takes a state predicate $P(r, x, y)$ and if r is hidden, then $Maybe$ is true at states that could possibly satisfy P (if r is visible, then $Maybe$ is equivalent to P). The parameter m determines whether r is visible or not. The operator $Observable$ describes states that could possibly satisfy the state predicate Inv (with respect to hidden variables), and at which it is possible to observe whether we are inside P provided that we are inside R.

$$\begin{aligned} & Maybe(v,\ x,\ y,\ m,\ P(\_,\ \_,\ \_)) \ \triangleq \\ & \quad \exists\, h:\ \text{LET}\ r \ \triangleq \ Mask(m,\ v,\ h) \\ & \qquad\quad \text{IN} \quad P(r,\ x,\ y) \end{aligned}$$

$$\begin{aligned} & Observable(v,\ x,\ y,\ m,\ P(\_,\ \_,\ \_), \\ & \qquad\qquad R(\_,\ \_,\ \_),\ Inv(\_,\ \_,\ \_)) \ \triangleq \\ & \ \land Maybe(v,\ x,\ y,\ m,\ Inv) \\ & \ \land \forall\, h:\ \text{LET}\ r \ \triangleq \ Mask(m,\ v,\ h) \\ & \qquad\quad \text{IN} \quad R(r,\ x,\ y) \Rightarrow P(r,\ x,\ y) \quad \text{P is observable within R} \end{aligned}$$

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 *Attractor* and *Trap* operators. The sets of states when observability is taken into account are shown in Fig. 21. 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 $G \triangleq Obs_1(Goal)$ for computing A
- A is replaced by $U \triangleq Obs_2(A)$ for computing D
- D is replaced by $Stay \triangleq Obs_1(D)$ 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 7 (Soundness): ASSUME : The sets of states D and T are computed as in Fig. 21. PROVE : The property

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

is realizable by component 2. The property

$$Q \triangleq \Box \vee \neg(T \vee A) \vee (\Diamond\Box T) \vee \Diamond A$$

is realizable by component 1. The implication holds

$$\models (T \wedge Inv) \Rightarrow D$$

A detailed proof can be found in the appendix.

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 $Stay$, $(Stay \wedge Inv) \Rightarrow D$. Thus, $(T \wedge Inv) \Rightarrow D$. QED

Theorem 7 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 7 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

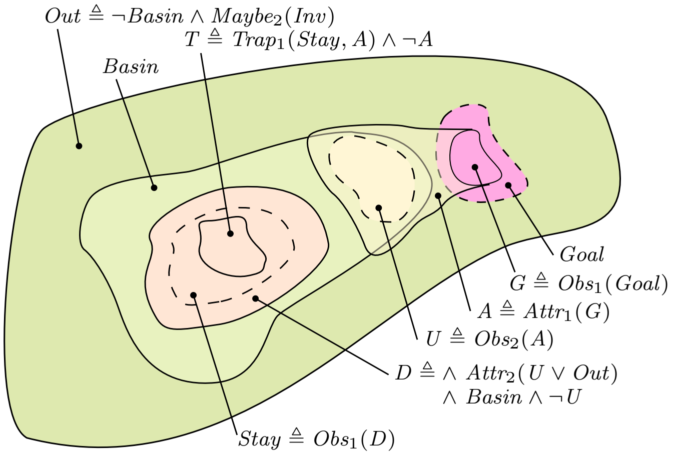

$Out \triangleq \neg Basin \wedge Maybe_2(Inv)$
$T \triangleq Trap_1(Stay, A) \wedge \neg A$
$Basin$
$Goal$
$G \triangleq Obs_1(Goal)$
$A \triangleq Attr_1(G)$
$U \triangleq Obs_2(A)$
$D \triangleq \wedge\ Attr_2(U \vee Out)$
$\wedge\ Basin \wedge \neg U$
$Stay \triangleq Obs_1(D)$

Fig. 21: Accounting for observability when computing assumptions.

```
$Y\ := Observable_1(Goal)$
$Yold\ :=$ CHOOSE $r\ :\ r \neq Y$
while $Y \neq Yold$ :
    $Yold\ := Y$
    $A, T, D\ :=$ MAKEPINFOASSUMPTION$(Y, \ldots)$
    (* ...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 [118].

Problem 8: Let variables $x_1, \ldots$ represent component 1 and variables $y_1, \ldots$ represent component 2, and *turn* a scheduler. Let $P(x_1, \ldots, y_1, \ldots, turn)$ be a (satisfiable) closed-system property, with $\Box\Diamond Goal(x_1, \ldots, y_1, \ldots, turn)$ as liveness. Assume that P in each step allows non-unique changes to at most one of the components, in a turn-based way via the scheduler.

Find those subsets of variables $x_a, \ldots$ and $y_p, \ldots$, and properties $\psi_1(x_1, \ldots, y_p, \ldots, turn), \psi_2(x_a, \ldots, y_1, \ldots, turn)$ with GR(1) liveness that depend on only the variables shown, such that $IsRealizable_{x_1,\ldots,turn;y_p,\ldots}(\psi_1) \wedge IsRealizable_{x_a,\ldots,turn;y_1,\ldots}(\psi_2)$, and $\big(\psi_1(x_1, \ldots, y_p, \ldots, turn) \wedge \psi_2(x_a, \ldots, y_1, \ldots, turn)\big) \Rightarrow P(x_1, \ldots, y_1, \ldots, turn)$.

Algo. 22 soundly solves Problem 8, as follows.

Theorem 9: Let the parameters m take values that correspond to making variables $x_a, \ldots$ visible to component 2, and $y_p, \ldots$ to component 1. If Algo. 22 returns a Y such that for those m values $Maybe_1(Inv) \Rightarrow Y$, and Inv is satisfiable, then the properties ψ_1, ψ_2 with liveness

$$\Box\Diamond G(x_1, \ldots, y_p, \ldots, turn) \vee \bigvee_i \Diamond\Box T_i(x_1, \ldots, y_p, \ldots, turn)$$

and

$$\bigwedge_i \Box\Diamond\neg D_i(x_a, \ldots, y_1, \ldots, turn),$$

respectively (and safety as in Sec. V) solve Problem 8.

Example 8: 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

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

The goal $\Diamond\Box T \lor \Box\Diamond(req = 0)$ 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 $\Diamond\Box T \lor \Box\Diamond(req = 0)$ 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 $req, occ, turn$

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

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 $spot\_1, spot\_2$). 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 $\Diamond\Box P_1 \lor \Box\Diamond R_1$ and $\Diamond\Box P_2 \lor \Box\Diamond R_2$. 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 $\Box\Diamond\neg P_1 \land \Box\Diamond\neg P_2$ 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 10 (Relaxation): ASSUME : TEMPORAL P, Q PROVE : $(IsRealizable(P) \land (P \Rightarrow Q)) \Rightarrow IsRealizable(Q)$.

For what we are interested in, let

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

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 11: Let

$$
\begin{array}{l}
Q \;\triangleq\; \lor\ \Diamond\Box T_1 \lor \Diamond\Box T_2 \\
\qquad\quad \lor\ \Box\Diamond R_1 \land \Box\Diamond R_2 \\
\text{ASSUME} : IsRealizable_1(Q) \land \land\ \neg D_1 \Rightarrow \neg T_1 \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ \neg D_2 \Rightarrow \neg T_2 \\
\text{PROVE} \ : \forall f : \lor \neg \land\ IsARealization_1(f, Q) \\
\qquad\qquad\qquad\quad \land\ Realization_1(f, Q) \\
\qquad\qquad\qquad\quad \land\ \Box\Diamond\neg D_1 \land \Box\Diamond\neg D_2 \\
\qquad\qquad\quad \lor\ \Box\Diamond R_1 \land \Box\Diamond R_2
\end{array}
$$

where $IsARealization$ is the modification of $IsRealizable$ 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 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. 22. 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 $TeamGoal$ shrinks, so $Attr(TeamGoal, Team)$ may shrink (not necessarily), but $Basin \land \neg TeamGoal$ 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

Algo. 22: Algorithm for constructing contracts of recurrence-persistence pairs.

```
def MakePinfoAssumption($Goal, Player, Team$) :
        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.
    $G := Observable(Goal, Player)$
    $A := Attr(G, Player)$
    $TeamGoal := Observable(A, Inv, Inv, Team)$
    $Basin := Attr(TeamGoal, Team)$
    $Escape :=$ TRUE
    $Converged :=$ FALSE
L1  while ($\neg \models Escape \equiv$ FALSE) :
            Complement within team state space.
L2      $Out := \neg Basin \wedge Maybe(Inv, Team)$
        $Holes := Basin \wedge Step(Out, Team)$
        $Escape := Out \wedge Image(Holes \wedge Inv, Team)$
L3      $Escape := Maybe(Escape, Team)$
                $\wedge Out \wedge \neg Converged$
L4      $Basin := Basin \vee Escape$
        $\eta_{player}, \eta_{team} :=$ MakePair(
            $A, Basin, Player, Team$)
        $Converged := Converged$
                $\vee NonEmptySlices(\eta_{player})$
    return $A, \eta_{player}, \eta_{team}$

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

Fig. 17. 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. 23. 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. 24. The variable *Coverged* is used for this purpose. The operator $NonEmptySlices(\eta_{player}) \triangleq \exists vars : \eta_{player}$ 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 12 (Termination): ASSUME : A finite number of states satisfies the global invariant Inv. PROVE : Algo. 22 terminates in a finite number of iterations.

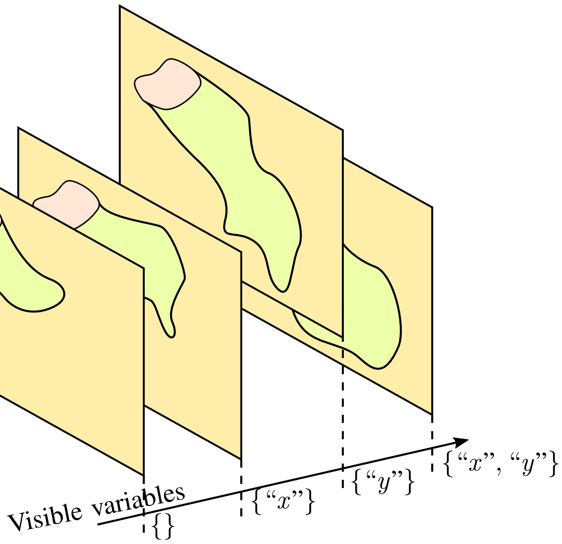


Fig. 23: Slices of the state space that correspond to different assignments of values to the parameters.

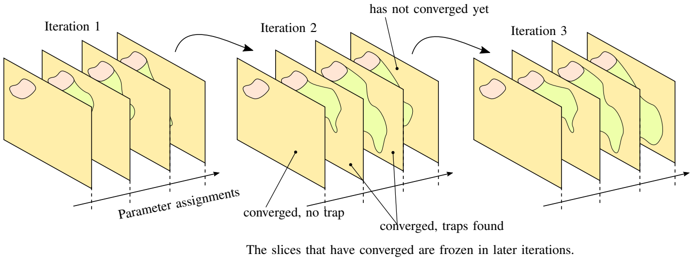


Fig. 24: In iterations of non-monotonic operators that depend on parameters, when a solution is found for some parameter values (a slice), then no further iterations should occur for those values.

A proof is given in Appendix C.

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 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 [94, 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 [106, 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).

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

$$\models Maybe_i(Inv) \Rightarrow Y,$$

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 7 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. 22 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 $\neg Goal \wedge Inv$. This condition can be applied to the algorithm in order to improve observability. This modification is obtained by the replacement

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

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 $\neg Within$ 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, which is component 1 at the first stage of decomposition. In each interconnection mode, different variables are communicated between components. The root component switches between interconnections when it observes that the current recurrence goal has been reached. Each interconnection is signified by the value of a variable controlled by the root component (an additional field in the record discussed below). If this variable is visible to all other components, then switching occurs in a single step. Otherwise, the change of mode is propagated from component to component along a fixed spanning tree over the components (by copying the value of the mode variable), starting from the root component, and takes multiple steps to complete. In configurations that occur intermediately while transitioning from one interconnection to another, if a component has insufficient information to change its state (because its neighboring components have not yet all switched to the new mode), then the component takes a stuttering step, which is allowed by its specification, as assumed earlier.

Each component is required to not violate the shared invariant Inv (via closure of the overall specification), so no change of another component would lead outside that invariant. The decomposition algorithm ensures that for each interconnection, the components can satisfy the corresponding liveness objective from within this invariant.

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. For example, let $cnct \in 1..2$ represent the current interconnection mode. If in mode 1 decomposition yields the liveness objective $\Diamond\Box P \vee \Box\Diamond G$ for a component, then this becomes $\Diamond\Box(P \wedge (cnct = 1)) \vee \Box\Diamond((cnct = 1) \Rightarrow G)$. This can be regarded as a component assuming that if it provides enough information to its environment, then it can request reactions that become feasible for the environment when that information is available. While information availability does not match any interconnection mode, a component can stutter, because its liveness requirements are "turned off".

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 finite set of strings as domain [10, p. 49]. 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 $vars_1 \in [\text{SUBSET}\ \{\text{"x"}, \text{"z"}\} \rightarrow Val]$ 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.

VII. Generating Minimal Specifications

We use binary decision diagrams [24], [137] 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 [138], [139] (e.g., as circuits), and in electronic design automation [140], [141]. 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. 25. 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 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 [142], [143], [144], [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.

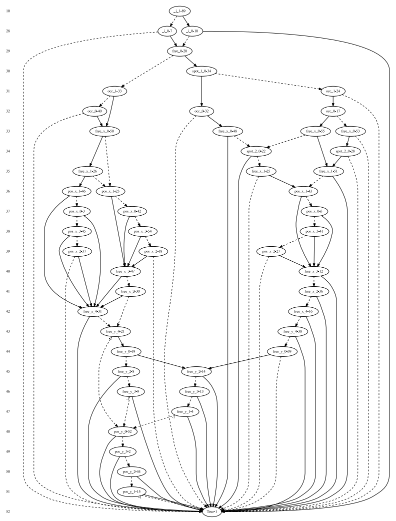


Fig. 25: The binary decision diagram from which the formula of the global invariant was generated for the charging station example in Sec. V-B. The variable names shown are the "bits" that are used to represent the integer-valued variables. A BDD is not very suitable to help a human understand what Boolean expression it represents.

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 [145], [146], [147], [148]. 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` [149]. Briefly, the problem of finding a minimal DNF formula of the form we described can be expressed as in Fig. 26, 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.

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,

$$
\begin{array}{l}
\text{EXTENDS } \mathit{FiniteSets},\ \mathit{Integers} \\
\text{CONSTANTS } \mathit{Variables},\ \mathit{Domain},\ \mathit{CareSet} \\
\mathit{Assignments} \ \triangleq\ [\mathit{Variables} \to \mathit{Int}] \\
\text{ASSUME} \\
\quad \land (\mathit{Domain} \subseteq \mathit{Assignments}) \land (\mathit{CareSet} \subseteq \mathit{Domain}) \\
\quad \land \mathit{IsFiniteSet}(\mathit{Domain}) \land \mathit{IsFiniteSet}(\mathit{Variables}) \\
\quad \land (\mathit{CareSet} \neq \{\}) \land f \in [\mathit{Domain} \to \text{BOOLEAN}\,] \\
\mathit{EndPoint}(k) \ \triangleq\ [1 \,..\, k \to \mathit{Domain}] \\
\mathit{IsInOrthotope}(x,\ a,\ b) \ \triangleq\ \forall\, \mathit{var} \in \mathit{Variables} : \\
\quad (a[\mathit{var}] \leq x[\mathit{var}]) \land (x[\mathit{var}] \leq b[\mathit{var}]) \\
\mathit{IsInRegion}(x,\ p,\ q) \ \triangleq\ \exists\, i \in \text{DOMAIN } p : \\
\quad \mathit{IsInOrthotope}(x,\ p[i],\ q[i]) \\
\mathit{SameOver}(f,\ p,\ q,\ S) \ \triangleq\ \forall\, x \in S : \\
\quad f[x] \equiv \mathit{IsInRegion}(x,\ p,\ q) \\
\text{p, q define a cover that contains k orthotopes} \\
\mathit{IsMinDNF}(k,\ p,\ q,\ f) \ \triangleq \\
\quad \land \{p,\ q\} \subseteq \mathit{EndPoint}(k) \\
\quad \land \mathit{SameOver}(f,\ p,\ q,\ \mathit{CareSet}) \\
\quad \land \forall\, r \in \mathit{Nat} :\ \forall\, u,\ v \in \mathit{EndPoint}(r) : \\
\qquad \lor \neg \mathit{SameOver}(f,\ u,\ v,\ \mathit{CareSet}) \quad \text{not a cover, or} \\
\qquad \lor r \geq k \quad \text{u, v has at least as many disjuncts as p, q}
\end{array}
$$

Fig. 26: The roblem of finding a minimal DNF formula.

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. Minimization is performed with respect to a given care predicate.

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 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 9: We show the usefulness of decompiling BDDs by revisiting the charging station example from Sec. V-B. Fig. 25 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 [150], 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 following minimal DNF formula. The meaning of this formula are all those states from where the robot can repeatedly request a spot for charging, and the station can respond by communicating which spot it has reserved for the robot to use.

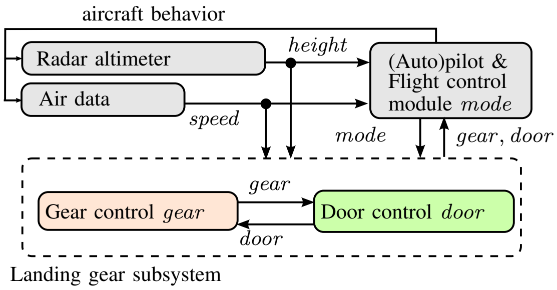


Fig. 27: Landing gear avionics. Arrows that enter the dashed boundary are connected to both modules inside it.

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

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 [124], [151]. Three modules (components) are involved, as shown in Fig. 27. 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 input to the decomposition algorithm is the overall system specification described below, together with which variables represent each component. The output are the specifications for each component and the interconnections between components, as described below.

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 in Fig. 28 in TLA$^+$.

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 $vars$-nonstuttering step, So the scheduler changes in a noninterleaving way with respect to the other components. As in Remark 6, the scheduler is required to take infinitely many nonstuttering steps, i.e., $\Box\Diamond\langle SchedulerNext\rangle_{turn}$, which allows fixpoint algorithms where the variable $turn$ changes in each step. The specification has constant parameters $max\_height, \ldots$ 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

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

The two conjuncts below were used as care predicate.

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

$$\begin{array}{l}
\text{EXTENDS } Integers \\
\text{VARIABLES } mode, height, speed, door, gear, turn \\
\text{CONSTANTS } max\_height, max\_speed, door\_down, \\
\qquad gear\_down, threshold\_height, threshold\_speed
\end{array}$$

mode 0, 2, 1 used below

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

Fig. 28: Assembled-system specification for the landing gear example.

The action $\mathit{AutopilotStep}$ applies to steps that change the autopilot. The action that constrains the autopilot is

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

Note that only variables that represent the autopilot appear primed in the action $\mathit{AutopilotStep}$.

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

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

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

- door corresponds to h
- $\mathit{gear}, \mathit{turn}$ to x
- $\mathit{height}, \mathit{mode}, \mathit{speed}$ to y

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

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

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 $\mathit{InvH} \equiv \mathit{InvWithDoorHidden}$. 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

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

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 $\mathit{SimplerAutopilotStep}$ defines the autopilot action by letting

$$
\begin{array}{l}
\mathit{SimplerAutopilotNext}(\mathit{gear}, \mathit{turn}, \mathit{height}, \mathit{mode}, \mathit{speed}, \\
\qquad\qquad\qquad\qquad \mathit{height}', \mathit{mode}', \mathit{speed}') \triangleq \\
\quad \land\ \mathit{InvH} \\
\quad \land\ \lor (\mathit{turn} = 1) \land \mathit{SimplerAutopilotStep} \\
\qquad \lor\ \text{UNCHANGED}\ \langle \mathit{height}, \mathit{mode}, \mathit{speed} \rangle
\end{array}
$$

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 $\mathit{turn} = 1$. The environment action $\mathit{SimplerEnvNext}$ is obtained after existential quantification of door and door' from the environment action. The scheduler remains the same, changing turn in each step. In the gear module's turn ($\mathit{turn} = 3$), the action is

$$
\begin{array}{l}
\mathit{SimplerGearModuleNext} \triangleq \\
\quad \land\ \mathit{gear} \in 0 \,..\, 5 \land \mathit{gear}' \in 0 \,..\, 5 \\
\quad \land\ \mathit{height} \in 0 \,..\, 100 \land \mathit{mode} \in 0 \,..\, 2 \\
\quad \land\ \mathit{speed} \in 0 \,..\, 40 \\
\quad \land\ \ \lor (\mathit{gear} \in 0 \,..\, 4) \land (\mathit{gear}' = 0) \\
\qquad\ \lor (\mathit{gear} \in 1 \,..\, 5) \land (\mathit{gear}' = 5) \\
\qquad\ \lor\ \land (\mathit{height} \in 76 \,..\, 100) \land (\mathit{mode} = 2) \\
\qquad\qquad \land (\mathit{speed} \in 0 \,..\, 30) \\
\quad \land\ \text{LET}\ \mathit{turn} \triangleq 3\ \ \text{IN}\ \ \mathit{InvH} \\
\quad \land\ (\text{LET}\ \mathit{turn} \triangleq 1 \\
\qquad \text{IN}\ \ \exists\, \mathit{height}, \mathit{mode}, \mathit{speed} :\ \mathit{InvH})'
\end{array}
$$

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 $\mathit{threshold\_speed}$. 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

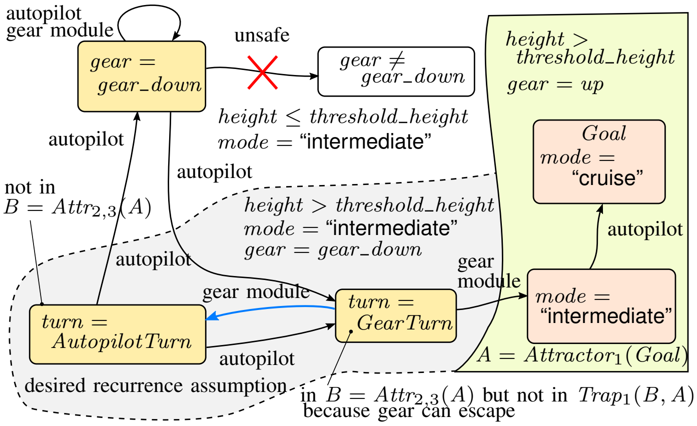


Fig. 29: The reason why the algorithm of Sec. VI-C is useful in the landing gear example. This description is simplified, in that it corresponds to a specification with the constraint $(mode = \text{"cruise"}) \Rightarrow (gear = 0)$ in $AutopilotNext$.

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. 29, 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 node on the bottom left (each node corresponds to several states). In that node, 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 node 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 node, 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 node. The autopilot has a choice to not go backwards, thus it can keep the behavior within the two bottom left nodes, until the gear does retract.

Algo. 22 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. 22. 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

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

and the resulting persistence objective $\Diamond\Box(AutopilotTrap \land (cnct = 0))$. 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 $\Box\Diamond((cnct \neq 0) \lor \neg DTeam)$ for the door-gear subsystem is defined by

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

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

$$\begin{array}{l}
Gear\_Trap \triangleq \\
\land\ turn \in 1\,..\,3 \land door \in 0\,..\,5 \\
\land\ gear \in 0\,..\,5 \land height \in 0\,..\,100 \land mode \in 0\,..\,2 \\
\land\ \quad \lor \land (turn = 2) \land (gear = 0)
\end{array}$$

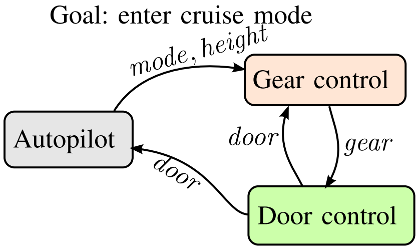


Fig. 30: Communicated variables when goal is cruise mode.

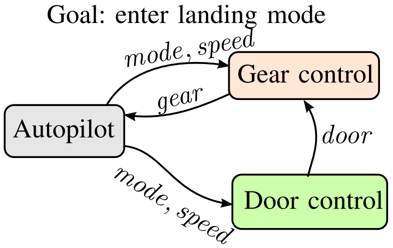


Fig. 31: Communicated variables when goal is landing mode.

$$
\begin{aligned}
&\quad \land (\mathit{height} \in 76\,..\,100) \land (\mathit{mode} = 2)\\
\lor &\quad \land (\mathit{door} \in 1\,..\,5) \land (\mathit{gear} = 0)\\
&\quad \land (\mathit{height} \in 76\,..\,100) \land (\mathit{mode} = 2)
\end{aligned}
$$

and for the door module

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

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 $\mathit{gear} = 0$). 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. 30. For the doors, the requirement is that if the gear has retracted ($\mathit{gear} = 0$), then the doors should not be open ($\mathit{door} \in 1..5$).

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. 31 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. 31 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 $\mathit{Stay} := \mathit{Observable}(D, \mathit{Inv}, \mathit{Inv}, \mathit{Player})$ in Algo. 22 to $\mathit{Stay} := \mathit{Observable}(D, \mathit{Within} \land \mathit{Inv}, \mathit{Inv}, \mathit{Player})$, 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 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 [152].

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, 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 (about the *turn* variable), 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] R. Rosner, "Modular synthesis of reactive systems," Ph.D. dissertation, Weizmann Institute of Science, Rehovot, Israel, 1992.

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

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

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

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

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

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

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

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

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

[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.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. A. Henzinger, and K. G. Larsen, "Contracts for systems design," INRIA, Tech. Rep. 8147, 2012.

[54] ——, "Contracts for system design," *Foundations and Trends® in Electronic Design Automation*, vol. 12, no. 2–3, pp. 124–400, 2018.

[55] A. Sangiovanni-Vincetelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-based design for cyber-physical systems," *European Journal of Control (EJC)*, vol. 18, no. 3, pp. 217–238, 2012.

[56] 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

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

[58] P. Nuzzo, H. Xu, N. Ozay, J. B. Finn, A. L. Sangiovanni-Vincentelli, R. M. Murray, A. Donzé, and S. A. Seshia, "A contract-based methodology for aircraft electric power system design," *IEEE Access*, vol. 2, pp. 1–25, 2014.

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

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

[61] T. T. H. Le, R. Passerone, U. Fahrenberg, and A. Legay, "Contract-based requirement modularization via synthesis of correct decompositions," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 15, no. 2, pp. 33:1–33:26, 2016.

[62] A. Iannopollo, P. Nuzzo, S. Tripakis, and A. Sangiovanni-Vincentelli, "Library-based scalable refinement checking for contract-based design," in *DATE*, 2014, pp. 1–6.

[63] A. Iannopollo, S. Tripakis, and A. Sangiovanni-Vincentelli, "Constrained synthesis from component libraries," in *FACS*, 2016, pp. 1–6.

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

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

[66] S. S. Bauer, A. David, R. Hennicker, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski, "Moving from specifications to contracts in component-based design," in *FASE*, 2012, pp. 43–58.

[67] S. Tripakis, B. Lickly, T. A. Henzinger, and E. A. Lee, "A theory of synchronous relational interfaces," *TOPLAS*, pp. 14:1–14:41, 2011.

[68] V. Preoteasa and S. Tripakis, "Refinement calculus of reactive systems," in *EMSOFT*, 2014, pp. 2:1–2:10.

[69] ——, "Towards compositional feedback in non-deterministic and non-input-receptive systems," in *LICS*, 2016, pp. 768–777.

[70] V. Preoteasa, I. Dragomir, and S. Tripakis, "Type inference of Simulink hierarchical block diagrams in Isabelle," in *FORTE*, 2017, pp. 194–209.

[71] M. Broy, F. Dederich, C. Dendorfer, M. Fuchs, T. Gritzner, and R. Weber, "The design of distributed systems—An introduction to FOCUS," Technische Universität München, Tech. Rep., 1992.

[72] M. Broy and K. Stølen, *Specification and Development of Interactive Systems*. Springer, 2001.

[73] R. Alur and T. A. Henzinger, "Reactive modules," *FMSD*, vol. 15, no. 1, pp. 7–48, 1999.

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

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

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

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

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

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

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

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

[82] H. Lamouchi and J. Thistle, "Effective control synthesis for DES under partial observations," in *CDC*, vol. 1, 2000, pp. 22–28.

[83] S. Tripakis, "Undecidable problems of decentralized observation and control," in *CDC*, vol. 5, 2001, pp. 4104–4109.

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

[85] R. Alur, L. de Alfaro, T. A. Henzinger, and F. Y. Mang, "Automating modular verification," in *CONCUR*, 1999, pp. 82–97.

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

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

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

[89] 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.

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

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

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

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

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

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

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

[97] 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.

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

[99] 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

[100] 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

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

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

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

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

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

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

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

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

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

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

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

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

[113] Z. Manna and A. Pnueli, "A hierarchy of temporal properties," in *PODC*, 1990, pp. 377–410.

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

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

[116] 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

[117] 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.

[118] 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.

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

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

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

[122] 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

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

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

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

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

[127] M. Abadi, B. Alpern, K. R. Apt, N. Francez, S. Katz, L. Lamport, and F. B. Schneider, "Preserving liveness: Comments on safety and liveness from a methodological point of view," *IPL*, vol. 40, no. 3, pp. 141–142, 1991.

[128] O. Lichtenstein, A. Pnueli, and L. Zuck, "The glory of the past," in *Conference on logics of programs*, 1985, vol. 193, pp. 196–218.

[129] M. Y. Vardi, "Verification of open systems," in *FSTTCS*, 1997, pp. 250–266.

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

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

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

[133] M. Abadi and L. Lamport, "Composing specifications," *TOPLAS*, vol. 15, no. 1, pp. 73–132, 1993.

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

[135] 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.

[136] I. Filippidis, "Decomposing formal specifications into assume-guarantee contracts for hierarchical system design," Ph.D. dissertation, California Institute of Technology, 2018. [Online]. Available: http://resolver.caltech.edu/CaltechTHESIS:12172017-171304566

[137] 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.

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

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

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

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

[142] 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.

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

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

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

[146] 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

[147] 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.

[148] 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.

[149] 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

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

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

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

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

[154] L. Lamport, "A temporal logic of actions," Systems Research Center of Digital Equipment Corporation, Palo Alto, CA, Tech. Rep. Research Report 47, Apr 1990.

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

[156] 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. 1030–1041.

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

Ioannis Filippidis graduated with a Diploma in mechanical engineering from the National Technical University of Athens, Greece, in 2011. He received a Ph.D. in control and dynamical systems from the California Institute of Technology, Pasadena, CA, USA, in 2018.

During the summers of 2013 and 2014 he was an intern at the Laboratory for Reliable Software, Jet Propulsion Laboratory, California Institute of Technology. He is a Postdoctoral Scholar with the Networked Control Systems Group in Control & Dynamical Systems, Computing and Mathematical Sciences Department, California Institute of Technology.

Richard M. Murray received the B.S. degree in Electrical Engineering from California Institute of Technology in 1985 and the M.S. and Ph.D. degrees in Electrical Engineering and Computer Sciences from the University of California, Berkeley, in 1988 and 1991, respectively. He is currently the Thomas E. and Doris Everhart Professor of Control & Dynamical Systems and Bioengineering at Caltech. Murray's research is in the application of feedback and control to networked systems, with applications in biology and autonomy. Current projects include analysis and design of biomolecular feedback circuits, synthesis of discrete decision-making protocols for reactive systems, and design of highly resilient architectures for autonomous systems.

Appendix

A. General remarks

Remark 7: When we refer to “sets” of variables, these sets are not sets in the object language (here TLA⁺), but in the metatheory [153]. □

Remark 8: It is possible to define the operator *Earlier* (Sec. IV-B3) by using a modified satisfaction relation $\models$ in raw TLA⁺, but we will omit this definition here. □

Remark 9: 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 (see also [154, Sec. 2.2.2 on p. 6]). □

Remark 10 (Recursive operator definitions): Recursive operator definitions (as those in Fig. 6) are part of TLA^{+2} [109]. □

Remark 11 (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 [108]. A collection that is not a set is called a proper class [155, 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 [116, 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. □

Remark 12: Realizability can be defined without design of initial conditions for the component ($y0$ in Fig. 5). In that case the component specification should not constrain the initial condition [116, §3.3, pp. 14–16]. Initial conditions then become part of an antecedent. In consequence, a conjunction of realizable component specifications would not imply a closed-system property. The choice between these different definitions is a matter of specification style. □

Remark 13: The operator *Step* defined in Sec. III-D corresponds to 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

$$
\begin{array}{l}
\mathit{GeneralStep}(x,\ y,\ \mathit{Target}(\_,\ \_)) \triangleq \exists\, y' : \\
\quad \land\ \mathit{SysNext}(x,\ y,\ y') \\
\quad \land\ \forall\, x' : \ \ \lor \neg \mathit{EnvNext}(x,\ y,\ x') \\
\qquad\qquad\quad\ \lor\ \land\ \mathit{AssemblyNext}(x,\ y,\ x',\ y') \\
\qquad\qquad\qquad\ \land\ \mathit{Target}(x',\ y')
\end{array}
$$

The general case is mentioned for completeness. □

Remark 14: In the full information case, the assumption that in each step at most one component can change in a non-unique way ensures determinacy [119], which allows for decomposition with an algorithm that extends Algo. 15 [136, Ch. 10], [124]. The algorithms we described can be extended also to a setting without the assumption about non-unique changes, by using the operator *GeneralStep* in fixpoint computations [136, §9.5]. □

Remark 15: In Fig. 6, *Goal* is a first-order operator. An attractor definition with *Goal* being a set is possible too [156, §IV-A]. □

Remark 16 (Comparison of WhilePlusHalf to $\overset{+}{\rhd}$): Only v is constrained in the first state of the suffix, thus the “half” in the name of *WhilePlusHalf*. In contrast, the operator $\overset{+}{\rhd}$ 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 [116, §5.2.4, pp. 26–27]. Relaxing G can lead to underspecification, which motivates defining *WhilePlusHalf*. □

Remark 17 (Defining closure): The closure of a temporal property can be defined as follows [8], [48, Sec. 5.3], [64, Sec. 2.1 on p. 52]

$$
\begin{array}{l}
\mathit{MustUnstep}(b) \ \triangleq\ \ \land\ b = \text{TRUE} \\
\qquad\qquad\qquad\qquad\ \land\ \Box[b' = \text{FALSE}]_b \\
\qquad\qquad\qquad\qquad\ \land\ \Diamond(b = \text{FALSE}) \\
\mathit{SamePrefix}(b,\ u,\ x) \ \triangleq\ \Box(b \Rightarrow (u = x)) \\
\mathit{Front}(P(\_),\ x,\ b) \ \triangleq\ \exists\, u :\ \ P(u) \land \mathit{SamePrefix}(b,\ u,\ x) \\
\\
\mathit{Cl}(P(\_),\ x) \ \triangleq\ \forall\, b :\ \ \mathit{MustUnstep}(b) \Rightarrow \mathit{Front}(P,\ x,\ b)
\end{array}
$$

□

Remark 18: Let $\mathit{Phi}(x_1, \ldots, y_1, \ldots)$ be a temporal property. The notation $\mathit{IsRealizable}_{x_a, x_b, \ldots; y_1, \ldots}(\mathit{Phi})$ is a shorthand for

$$
\begin{array}{l}
\mathit{Realization}(x_a, x_b, \ldots, y_1, \ldots, f_1, \ldots, g, \\
\qquad\qquad y0_1, \ldots, \mathit{mem}0) \ \triangleq \\
\quad \exists\, \mathit{mem} : \\
\qquad \text{LET } v \ \triangleq\ \langle \mathit{mem}, x_a, x_b, \ldots, y_1, \ldots \rangle \\
\qquad\quad\ \ A \ \triangleq\ \land\ y_1' = f_1[v] \\
\qquad\qquad\qquad\ \land\ y_2' = f_2[v] \\
\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad\ \land\ \mathit{mem}' = g[v] \\
\qquad \text{IN}\ \ \land\ \langle \mathit{mem}, y_1, \ldots \rangle = \langle \mathit{mem}0, y0_1, \ldots \rangle \\
\qquad\qquad \land\ \Box[A]_v \land \text{WF}_{\langle \mathit{mem}, y_1, \ldots \rangle}(A) \\
\mathit{IsRealizable}(\mathit{Phi}(\_, \ldots)) \ \triangleq \\
\quad \exists f_1, \ldots, g, y0_1, \ldots, \mathit{mem}0 : \\
\qquad \land\ \mathit{IsAFiniteFcn}(f_1) \land \cdots \land \mathit{IsAFiniteFcn}(g) \\
\qquad \land\ \text{LET } R(u_a, u_b, \ldots, v_1, \ldots) \ \triangleq\ \mathit{Realization}(\\
\qquad\qquad\qquad u_a, u_b, \ldots,\ v_1, \ldots,\ f_1, \ldots, g, \\
\qquad\qquad\qquad y0_1, \ldots, \mathit{mem}0) \\
\qquad\quad \text{IN}\ \ \forall\, x_1, \ldots, y_1, \ldots : \\
\qquad\qquad R(x_a, x_b, \ldots, y_1, \ldots) \Rightarrow \mathit{Phi}(x_1, \ldots, y_1, \ldots)
\end{array}
$$

□

Remark 19: In the definition of a contract (Sec. IV-A), we use operators $A(\_, \ldots), \ldots$ to specify components, and refer to variables $x, \ldots, w$. Formally, a statement of the form VARIABLES $x, \ldots, w$ declares these variables in the current context [10, §3.1, §17.3, §17.5.2].

Alternatively, we can use temporal quantification, as follows

$$\begin{aligned}&\land\ \mathit{IsRealizable}_1(A) \land \cdots \land \mathit{IsRealizable}_n(W)\\&\land\ \boldsymbol{\forall}\, x, \dots, z\ :\ \big(A(x, \dots) \land \dots \land W(\dots, z)\big) \Rightarrow \mathit{Phi}(x, \dots)\end{aligned}$$

The effect of a temporal quantifier includes declaring the variables that it bounds [10, p. 41, p. 110]. The position of each operator argument determines the role of each variable. We could rename the bound variables (including those bound by $\boldsymbol{\forall}$ within *IsRealizable*) without changing the formula's meaning. Instead of distinguishing arguments, values can be used as identifiers [10, §10.2], with *IsRealizable* defined accordingly. □

Remark 20: In a formula of the form (as in Example 1)

$$\begin{aligned}&\lor\ \neg \land\ \mathit{PhiS}(\dots)\\&\qquad\ \land\ \mathit{PhiR}(\dots)\\&\lor\ \mathit{Phi},\end{aligned}$$

the negation applies to the entire conjunction list [10, §15.2.2 on p. 286], and the disjunction is equivalent to the implication $(\mathit{PhiS}(\dots) \land \mathit{PhiR}(\dots)) \Rightarrow \mathit{Phi}$, provided the operators $\mathit{PhiS}, \mathit{PhiR}, \mathit{Phi}$ are Boolean-valued [10, §16.1.3]. □

Remark 21 (Quantifier elimination and operators): In Sec. V-C we defined the operator *SimplerEnvNext* as

$$\begin{aligned}&\mathit{SimplerEnvNext}(x,\ y,\ x') \ \triangleq\\&\quad \exists\, h,\ h' :\ \ \mathit{Inv}(h,\ x,\ y) \land \mathit{EnvNext}(h,\ x,\ y,\ h',\ x')\end{aligned}$$

When we compute a quantifier-free formula equivalent to the above formula, by eliminating quantifiers, we should define a new operator, for example *Foo*. We cannot reuse the name *SimplerEnvNext*, even though we can prove that $\mathit{Foo} \equiv \mathit{SimplerEnvNext}$ (equivalent).

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 ×, 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 [153, §74, Vol. 1, p. 405]. □

B. Detailed problem and theorem statements

Below are more detailed versions of Problem 4 and Theorem 5, including an environment and details about safety. The statement below generalizes to multiple variables.

Problem 13: Let variable x represent component 1, variable y component 2, and variable z their environment. Let $P(x, y, z)$ be a (finite-state) closed-system property of the form $I \land \Box[N]_{\langle x,y,z\rangle} \land \Box\Diamond \mathit{Goal}(x, y, z)$ that in each step allows non-unique changes to at most one of x, y, z. Let $\varphi(x, y, z) \triangleq \mathit{Unzip}_{z;x,y}(P, z, x, y)$ (z env) and assume that $\mathit{IsRealizable}_{z;x,y}(\varphi)$.

Find temporal properties ψ_1, ψ_2 with GR(1) liveness such that $\mathit{IsRealizable}_{z,y;x}(\psi_1) \land \mathit{IsRealizable}_{z,x;y}(\psi_2)$ and $\big(\psi_1(x, y, z) \land \psi_2(x, y, z)\big) \Rightarrow \varphi(x, y, z)$.

Theorem 14: ASSUME : Algo. 15 returns a Y such that $\mathit{Inv} \Rightarrow Y$ (*Inv* is satisfiable by realizability of φ). Pick x_0, y_0 such that $(\exists w : \mathit{WInit}(x_0, y_0, w)) \land \forall w : (\exists u, v : \mathit{WInit}(u, v, w)) \Rightarrow \mathit{WInit}(x_0, y_0, w)$. Let $Q(u, v, w) \triangleq (u = x_0) \land (v = y_0) \land W(u, v, w)$, where $\mathit{Inv}, W, \mathit{WInit}$ as in Sec. V.

PROVE : The specifications

$$\begin{aligned}\psi_1(x, y, z) \triangleq \text{LET } & P_1(w, v, u) \triangleq\\&\quad \land\ Q(u, v, w)\\&\quad \land\ \Box\Diamond \mathit{Goal}(u, v, w) \lor \textstyle\bigvee_i \Diamond\Box T_i(u, v, w)\\ \text{IN } & \ \mathit{Unzip}_{z,y;x}(P_1, z, y, x)\end{aligned}$$

and

$$\begin{aligned}\psi_2(x, y, z) \triangleq \text{LET } & P_2(w, u, v) \triangleq\\&\quad \land\ Q(u, v, w)\\&\quad \land\ \textstyle\bigwedge_i \Box\Diamond \neg T_i(u, v, w)\\ \text{IN } & \ \mathit{Unzip}_{z,x;y}(P_2, z, x, y)\end{aligned}$$

solve Problem 4, where T_i are the traps returned by Algo. 15.

Initial conditions x_0, y_0 are selected above to ensure that the system starts from a state that satisfies *WInit*, which is possible due to realizability of φ, the stepwise form of *Unzip*, and the assumption $\mathit{Inv} \Rightarrow Y$.

Remark 22: The above result can be applied also to other approaches (e.g., based on raw TLA+ or LTL), with suitable changes to *IsRealizable* and *Unzip*, or using *RawWhilePlusHalf*. □

C. A detailed proof of Theorem 12

A structured proof style is used [157], [109].

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

D. A detailed proof of Theorem 7

Below is the proof of Theorem 7 on 20. The proof is structured in levels [157], [109].

```
PROOF:
⟨1⟩1. $\mathit{IsRealizable}_2(\Box\Diamond\neg D)$
  ⟨2⟩1. $\mathit{IsRealizable}_2(\Box(D \Rightarrow \Diamond(U \lor \mathit{Out})))$
    ⟨3⟩1. DEFINE $Z \triangleq \mathit{Attr}_2(U \lor \mathit{Out})$
    ⟨3⟩2. $D \Rightarrow Z$
      BY DEF $D$
    ⟨3⟩3. $\mathit{IsRealizable}_2(\Box(Z \Rightarrow \Diamond(U \lor \mathit{Out})))$
      BY DEF $\mathit{Attr}$
    ⟨3⟩4. QED
      BY ⟨3⟩2, ⟨3⟩3
  ⟨2⟩2. $(U \lor \mathit{Out}) \Rightarrow \neg D$
    ⟨3⟩1. $D \Rightarrow \neg \mathit{Out}$
      ⟨4⟩1. $\mathit{Basin} \Rightarrow \neg \mathit{Out}$
        BY DEF $\mathit{Out}$
      ⟨4⟩2. $D \Rightarrow \mathit{Basin}$
        BY DEF $D$
      ⟨4⟩3. QED
        BY ⟨4⟩1, ⟨4⟩2
    ⟨3⟩2. $D \Rightarrow \neg U$
      BY DEF $D$
    ⟨3⟩3. QED
      BY ⟨3⟩1, ⟨3⟩2
  ⟨2⟩3. $\mathit{IsRealizable}_2(\Box(D \Rightarrow \Diamond\neg D))$
    BY ⟨2⟩1, ⟨2⟩2
  ⟨2⟩4. $\Box(D \Rightarrow \Diamond\neg D) \equiv \Box\Diamond\neg D$
    PROOF:

          $\Box(D \Rightarrow \Diamond\neg D) \equiv \Box \lor \neg D$
                                   $\lor\ D \land (D \Rightarrow \Diamond\neg D)$
          $\equiv \Box(\neg D \lor \Diamond\neg D)$

  ⟨2⟩5. QED
    BY ⟨2⟩3, ⟨2⟩4
⟨1⟩2. $\mathit{IsRealizable}_1(\Box \lor \neg(T \lor A)$
                    $\lor\ \Diamond A \lor \Diamond\Box T)$
  ⟨2⟩1. DEFINE $Z \triangleq \mathit{Trap}_1(\mathit{Stay}, A)$
  ⟨2⟩2. $\mathit{IsRealizable}_1(\Box(Z \Rightarrow \lor\ \Diamond A$
                         $\lor\ \Diamond\Box(Z \land \neg A)))$
    BY DEFS $Z, \mathit{Trap}$
  ⟨2⟩3. $T \equiv Z \land \neg A$
    BY DEFS $T, Z$
  ⟨2⟩4. $(T \lor A) \Rightarrow Z$
    ⟨3⟩1. $(T \lor A) \Rightarrow ((Z \land \neg A) \lor A)$
      BY ⟨2⟩3
    ⟨3⟩2. $(T \lor A) \Rightarrow (Z \lor A)$
      BY ⟨3⟩1
    ⟨3⟩3. $(Z \lor A) \Rightarrow Z$
      ⟨4⟩1. $A \Rightarrow Z$
        BY DEF $Z, \mathit{Trap}$
      ⟨4⟩2. QED
        BY ⟨4⟩1
    ⟨3⟩4. QED
      BY ⟨3⟩2, ⟨3⟩3
  ⟨2⟩5. QED
    BY ⟨2⟩2, ⟨2⟩3, ⟨2⟩4
⟨1⟩3. $(\mathit{Inv} \land T) \Rightarrow D$
  ⟨2⟩1. $T \equiv \mathit{Trap}_1(\mathit{Stay}, A) \land \neg A$
    BY DEF $T$
  ⟨2⟩2. $\mathit{Trap}_1(\mathit{Stay}, A) \Rightarrow (\mathit{Stay} \lor A)$
    BY DEF $\mathit{Trap}$
  ⟨2⟩3. $T \Rightarrow (\mathit{Stay} \land \neg A)$
    BY ⟨2⟩1, ⟨2⟩2
  ⟨2⟩4. $(\mathit{Inv} \land \mathit{Stay}) \Rightarrow D$
    BY DEFS $\mathit{Stay}, \mathit{Obs}_1$
  ⟨2⟩5. QED
    BY ⟨2⟩3, ⟨2⟩4
⟨1⟩4. QED
  BY ⟨1⟩1, ⟨1⟩2, ⟨1⟩3
```