Decomposing formal specifications into assume-guarantee contracts for hierarchical system design

Thesis by
Ioannis Filippidis

In Partial Fulfillment of the Requirements for the
degree of
Doctor of Philosophy

CALIFORNIA INSTITUTE OF TECHNOLOGY
Pasadena, California

2019
Defended November 30, 2017

Ioannis Filippidis
ORCID: 0000-0003-4704-3334

ACKNOWLEDGEMENTS

I would like to thank my advisor Richard Murray for advice and support through my PhD studies, the freedom to pursue a range of different research directions, and for the invaluable experience that I gained from our collaboration. I would like to thank Gerard Holzmann for insightful observations, his advice, and guidance during my internships at the Jet Propulsion Laboratory. I would like to thank the members of my thesis committee Joel Burdick and Mani Chandy for their helpful comments.

I would like to thank Necmiye Özay and Scott Livingston for discussions over the years that significantly helped in the research of this thesis. Discussions with Necmiye motivated Section 9.1, Section 9.2, and Section 9.6, and Necmiye pointed out the relevance of well-separation. Choices of quantification for initial conditions were formulated by Scott, and discussions with Scott motivated examining when an assume-guarantee specification should be regarded as vacuous. Section 9.3.3 was motivated by a discussion with Necmiye and Scott.

Thanks to the designers of TLA$^+$ for a useful and elegant language.

This work was supported in part by the Powell Foundation, the JPL Graduate Fellowship Program, and 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.

ABSTRACT

Specifications for complex engineering systems are typically decomposed into specifications for individual subsystems in a way that ensures they are implementable and simpler to develop further. We describe a method to algorithmically construct specifications for components that should implement a given specification when assembled. By eliminating variables that are irrelevant to realizability of each component, we simplify the specifications and reduce the amount of information necessary for operation. To identify these variables, we parametrize the information flow between components.

The specifications are written in TLA$^+$, with liveness properties restricted to an implication of conjoined recurrence properties, known as GR(1). We study whether GR(1) contracts exist in the presence of full information, and prove that memoryless GR(1) contracts that preserve safety do not always exist, whereas contracts in GR(1) with history-determined variables added do exist. We observe that timed stutter-invariant specifications of open-systems in general require GR(2) liveness properties for expressing them.

We formalize a definition of realizability in TLA$^+$, and define an operator for forming open-systems from closed-systems, based on a variant of the "while-plus" operator. The resulting open-system properties are realizable when expected to be. We compare stepwise implication operators from the literature, and establish relations between them, and examine the arity required for expressing these operators. We examine which symmetric combinations of stepwise implication and implementation kind avoid circular dependence, and show that only Moore components specified by strictly causal stepwise implication avoid circular dependence.

The proposed approach relies on symbolic algorithms for computing specifications. To convert the generated specifications from binary decision diagrams to readable formulas over integer variables, we symbolically solve a minimal covering problem. We implemented an algorithm for minimal covering over lattices originally proposed for two-level logic minimization. We formalized the computation of essential elements and cyclic core that is part of this algorithm, and machine-checked the proofs of safety properties using a proof assistant. Proofs supporting the thesis are organized as TLA$^+$ modules in appendices.

PUBLISHED CONTENT AND CONTRIBUTIONS

[Contrib1] I. Filippidis, R. M. Murray, and G. J. Holzmann, "A multi-paradigm language for reactive synthesis," in *4th Workshop on Synthesis (SYNT)*, 2015, pp. 73–97. DOI: 10.4204/EPTCS.202.6
Ioannis conceived and developed the proposed approach.

[Contrib2] I. Filippidis and R. M. Murray, "Symbolic construction of GR(1) contracts for systems with full information," in *2016 American Control Conference (ACC)*, 2016, pp. 782–789. DOI: 10.1109/ACC.2016.7525009
Ioannis conceived and developed the proposed approach.

[Contrib3] I. Filippidis and R. M. Murray, "Symbolic construction of GR(1) contracts for synchronous systems with full information," California Institute of Technology, Technical Report arXiv:1508.02705, 2015,
Ioannis conceived and developed the proposed approach.

[Contrib4] I. Filippidis and R. M. Murray, "Formalizing synthesis in TLA$^+$," California Institute of Technology, Technical Report CaltechCDSTR:2016.004, 2016. Available at: http://resolver.caltech.edu/CaltechCDSTR:2016.004
Ioannis conceived and developed the proposed approach.

[Contrib5] 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. DOI: 10.1109/CCA.2016.7587949
Ioannis contributed substantially code to the software toolbox and to its design, developed some elements of the approach, and wrote sections of the paper.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

Chapter 1

INTRODUCTION

1.1 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. 1.1. 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. 1.2. The maximum manipulator speed that the controller can safely command is limited by the camera frame rate. Depending on power supply and type of operation, the manipulator controller can request a lower frame rate from the camera, in order to economize on power. During grasping operations though, the controller does require high fidelity and frequent frames. Based on the available power supply, the controller may decide to decline a request for grasping, due to insufficient power for completing the operation. Such an issue could arise in rovers that depend on solar energy, because their power supply is contigent on environmental conditions.

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 [191]. The design activity should be captured accurately [122, 86]. 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

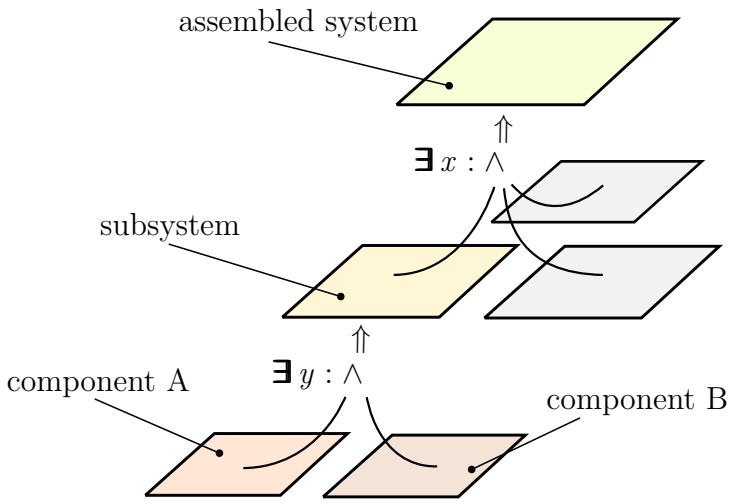


Figure 1.1: Anatomy of hierarchical system design in TLA$^+$: composition is represented by conjunction ($\wedge$), hiding of details by (temporal) existential quantification ($\exists$), and refinement by logical implication $\Rightarrow$.

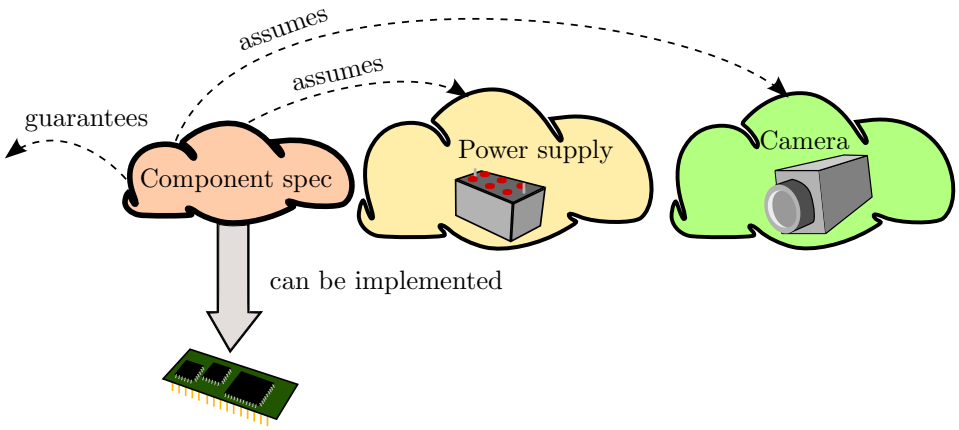


Figure 1.2: A component is specified by expressing requirements that the implementation should fulfill under assumptions about other parts of the assembled system.

creating interfaces between them [190, 105]. 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 [106]. 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 [115, 113]. But it may be the only scalable way of approaching the design of a large system [26], [115, pp. 421–422], [117, p. 168].

A system can be described at different levels of detail [111], [110, p. 192]. A

description that corresponds closely to available physical elements is directly implementable [191]. However, writing specifications at this level of detail is often more difficult than specifying behavior at a higher level. A specification at the implementation level can then be derived by hand or using (automated) *synthesis*, which relies on notions from the theory of games [179]. Synthesis has attracted considerable interest in the past two decades, and advances both in theory and implementation have been made, as described in Section 2.2 [167, 153, 152, 95, 189, 102, 52]. 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.

1.2 Proposed approach

A component mathematically means a collection of variables. A contract is a collection of realizable component specifications that combined imply the specification we decomposed (Section 4.1). 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.

Parametric analysis for hiding variables In this thesis, we study the problem of eliminating variables during the decomposition of specifications into assume-guarantee contracts between components. In order to detect which variables each component needs to know about, we use parameters that prescribe whether each variable is hidden or not (Chapter 5). 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, and propose a decomposition algorithm (Chapter 6). Thus, information is hidden without producing component specifications that would be computationally expensive or intractable to implement [59, 158, 159].

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) [153]. The polynomial

computational complexity of implementing open-system GR(1) specifications motivates this choice, discussed more in Section 3.3. The fulfillment of liveness requirements between components is acyclic, in order to avoid circular dependencies. An example case study is discussed in Chapter 8.

Generating readable specifications from BDDs 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 [30, 92, 39]. 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 (Chapter 7). The covering problem is solved exactly with a symbolic branch and bound algorithm originally proposed for two-level logic minimization [44].

Stepwise implication operators We study the properties of stepwise implication operators from the temporal logic of actions (TLA$^+$) and linear temporal logic (LTL), their relation, and avoidance of circular dependence between components. We propose an operator (*WhilePlusHalf*) for defining open-system specifications that are realizable when expected to be (Sections 4.2 and 9.3). Using this operator, we define an operaor (*Unzip*) for forming open from closed systems (Section 9.4.2).

We show that the "while-plus" operator ($\overset{+}{\dashrightarrow}$) of TLA$^+$ is expressible in raw TLA$^+$, as a strictly causal variant of strict implication used for LTL games. We show that the opposite is not true, unless both the environment and component are specified with machine-closed pairs of formulas (Section 9.1). We investigate how machine closure affects the transfer of liveness subformulas from environment to component descriptions and vice versa, and relate to notions of well-separation (Section 9.2).

We distinguish between the assume-guarantee structure and the liveness part of temporal properties. The term "GR(1)" refers to a form of liveness formulas

(one generalized Streett pair), and is orthogonal to what form of stepwise implication is specified (Section 9.8).

Circularity We investigate how circularity arises depending on what assume-guarantee operator is used and what kind of implementation is assumed, Mealy or Moore. We compare the associated controllable step operators, and show that the only symmetric case that avoids circular dependence is composition of Moore components (Section 9.6).

Existence of GR(1) contracts We study whether GR(1) contracts exist in the presence of full information, and prove that memoryless contracts in GR(1) do not always exist, whereas contracts in GR(1) with history-determined variables added do exist (Chapter 10).

History-determined variables and realizability Using raw TLA$^+$, we show that both hiding and unhiding of history-determined variables preserve realizability (Section 9.7), thus formalizing remarks about deterministic Büchi automata in the literature [153].

Stutter-invariant open-systems with time constraints We consider timed stutter-invariant specifications of open-systems, and show that in general GR(2) liveness properties are required for expressing them (Chapter 11).

A short literature review is given in Chapter 2. Chapter 3 introduces the mathematical language we use, a formalized notion of implementability, and some elements from algorithmic game theory. The algorithms are implemented in Python using the package `omega` [63], and the binary decision diagram package `dd` [62], which were written in the context of this thesis.

Where are the proofs? Specifications, theorems, and proofs are organized as TLA$^+$ modules in separate documents. These are described in Appendix A.

Chapter 2

PREVIOUS WORK

2.1 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 [86]. The assumption-commitment paradigm about reactive systems is an evolved instance of reasoning about conditions before and after a terminating behavior. Early formulations [87, pp. 26–29], [97, p. 4] were the assertion boxes used by Goldstine and von Neumann [74], and the tabulated assertions used by Turing [184, 143]. A formalism for reasoning using triples of a *precondition*, a program, and a *postcondition* was introduced by Hoare [80], following the work of Floyd [71], [97, pp. 3–4] on proving properties of elements in a flowchart, based on ideas by Perlis and Gorn [172, p. 122], [87, 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 [155]. Francez and Pnueli [72] 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 [110] observed that such a style of specification is essential to reason about complex systems in a modular way [157, p. 131]. Lamport and Schneider [109, 124] 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 [141], [7, §6 on p. 532], [78, §2.5 on pp. 247–248]. 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 [156]. Two developments followed, and the work presented here is based on them.

The first was Lamport's introduction of *proof lattices* [108]. 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 [149] 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 ($\stackrel{+}{\rightarrow}$ and variants) in temporal logic by Lamport [110], and by Pnueli [157], 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 [165, 192]. This prevents circular existential reasoning about the future, i.e., circular dependencies of liveness properties [6, §2.2, p. 512], [11, §5.4, p. 264], [175], [15, Prop. 14, p. 45]. 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 [175] proposed a proof rule for assume-guarantee reasoning about a non-circular collection of liveness properties. McMillan [133] 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 (Chapter 6).

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 [110, p. 191]. Meyer [137] 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* [50] 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 [22, 21, 170]. In this context, contracts are used broadly, as an umbrella term that encompasses both interface theories and assume-guarantee contracts [146, 147, 22, 148], 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 [38], as well as a verification tool of contract refinement using an SMT solver [37]. 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 [125]. 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 [81, 82]. In that approach, the components are automatically selected from an existing library, with the objective of creating an assembly that satisfies a given specification.

An algorithm for decomposing an LTL contract by partitioning its variables into subsets that define projections whose composition refines the given contract is studied in [83]. The algorithm progressively collects each subset of variables by detecting variable dependencies using a model checker. The resulting projections are used to decompose the synthesis of a composition of components from a library to smaller problems of synthesis from the library.

Contract theories in the framework of [22, 170] 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) [7, 88, 96]. 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 [22, 170, 20]. 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 [182], whereas in our approach the conjoined component specifications should imply the desired overall specification.

The theory of synchronous relational interfaces [182] is an approach that allows expressing safety contracts, and reasoning about composition, refinement, and component substitutability. The refinement calculus of reactive systems (RCRS) [162] 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 [163, 161]. 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 $\stackrel{+}{\rightarrow}$ and *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 [28, 29] which can express assume-guarantee specifications, open and closed systems, and supports reasoning about system composition and refinement.

Reactive modules [15] 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. A methodology for decomposing refinement proofs using assume-guarantee reasoning, abstraction of implementation details, and witness modules for instantiating internal state of the specification has been described in [77, 78].

2.2 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 [16, 51, 49]. Inability to observe external state changes makes a game asynchronous [158, 171]. 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 [155], 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 [153]. Asynchronous centralized synthesis does not yield to such a reduction [158]. Partial information games pose a challenge similar to full LTL properties, due to the need for a powerset-like construction [104]. To avoid this route alternative methods have been developed, using universal co-Büchi automata [102], or antichains [52].

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 [159], unless we restrict the communication architecture to avoid information forks [69], or restrict the specifications to limited fragments of LTL [34]. 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 [70]. Asynchronous distributed synthesis is undecidable [107, 181, 171].

Besides synthesis of a distributed implementation, the more general notion of *assume-guarantee* synthesis [32] 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 [14].

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 [40], and implemented symbolically [144]. Later work addressed synthesis by separating the construction of assumptions into safety and liveness [33]. The safety assumption is obtained by property *closure*, which also plays a key role in the composition theorem presented in [7]. Our work is based on this separate treatment of safety and liveness. Methods that use opponent strategies [99] to refine the assumptions of a GR(1) specification, searching over syntactic patterns, were proposed in [127, 17]. The syntactic approach of [17] was used in [18] to refine assume-guarantee specifications of coupled modules. However, that work cannot handle circularly connected modules, and 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 [57].

Our work uses parametrization, based on ideas of approximating asynchronous with GR(1) synthesis [158, 95, 94]. Another form of parametrization studied in the context of synthesis is that of safety and reachability goals [142]. Instead of hiding specific variables, an alternative approach in the context of verification [41] 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 [59]. Identifying what variables provide information essential for realizability (Chapter 5) relates to work on synthesizing probabilistic sensing strategies [73]. A hierarchical approach where an observer for the continuous state is designed separately from synthesizing a discrete controller from temporal logic specifications [138], and decomposition of properties for synthesizing implementations have been studied in the context of aircraft management systems [150]. Layering as a method for structuring system design has been applied in the context of the DisCo method, which is based on TLA [139, 89, 105].

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 [177], robot path planning among planar rectangles [173] and recently for simplifying enumerated robot controllers [76]. In the context of synthesis, prime implicants (used here for minimal covering)

have been used for refining abstractions [189], and have been mentioned in the context of debugging specifications [58]. For theories more general than propositional logic there has been work on deriving prime implicants in the context of SMT solvers [55].

Chapter 3

PRELIMINARIES

3.1 Predicate Logic and Set Theory

We use the temporal logic of actions (TLA$^+$) [117], 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) [114, §4], [135, 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 [117, p. 300], which is regarded as a foundation for mathematics [56]. 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 [117, p. 303, p. 71]. If a value f equals the function constructor that maps values in DOMAIN f to the values obtained by function application, then it is a function

$$IsAFunction(f) \triangleq f = [x \in \text{DOMAIN } f \mapsto f[x]].$$

For any $x \notin \text{DOMAIN } f$, $f[x]$ 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 [121]. A first-order operator takes as arguments operators

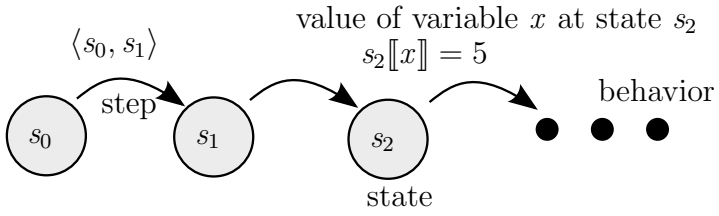


Figure 3.1: Semantic concepts of the temporal logic TLA$^+$.

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 [117, §17.1.1]. TLA$^+$ includes [117, §16.1.2] Hilbert's choice operator [79, 126]. If $\exists x : P(x)$, then the expression CHOOSE $x : P(x)$ equals some value that satisfies $P(x)$. Otherwise, this expression is some unspecified value that can differ depending on P.

The operator $\wedge$ denotes conjunction, $\vee$ disjunction, $\neg$ negation. *Nat* denotes the set of natural numbers [117, §18.6, p. 348], and for $i, j \in Nat$ the set of integers between i and j is denoted by

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

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

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

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

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

3.2 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, as illustrated in Fig. 3.1.

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. A state is a function from variable identifiers to values. In the metatheory of TLA$^+$, each identifier is a string "x", so the value $s[\![x]\!] \triangleq s[\text{"}x\text{"}]$.

We will use the temporal operators: $\Box$ "always" and $\Diamond$ "eventually". If formula f is TRUE in every (some) state of behavior σ, then $\sigma \models \Box f$ ($\sigma \models \Diamond f$) is TRUE. Formal semantics are defined in [117, §16.2.4].

A property is a collection of behaviors described by a temporal formula. If a property φ cannot distinguish between two behaviors that differ only by repetition of states, then φ is called *stutter-invariant*. Stutter-invariance is useful for refining systems by adding lower-level details [111]. In TLA$^+$ stutter-invariance is ensured by the constructs [11, Prop. 2.1]

$$\begin{aligned} [A]_v &\triangleq A \vee (v' = v), \\ \langle A \rangle_v &\triangleq \neg[\neg A]_v = A \wedge (v' \neq v). \end{aligned}$$

So, $(x = 0) \wedge \Box[x' = x + 1]_x$ is satisfied by a behavior whose each step either increments x by one, or leaves x unchanged. The main syntactic elements of TLA$^+$ are illustrated in Fig. 3.2. The notion of stuttering is illustrated in Fig. 3.3. As mentioned above, stutter-sensitive properties can be described in *raw* TLA$^+$ (RTLA$^+$). In addition, in later sections we use raw TLA$^+$ with past temporal operators [128]. We abbreviate this logic as PastRTLA$^+$. The relation between these different flavors of temporal logic is summarized with examples in Table 3.1.

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 [12] [172, p. 49]. A property of the form $\Box\Diamond p$ ($\Diamond\Box p$) is called *recurrence* (*persistence*) [130].

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 [117, Ch.16.2].

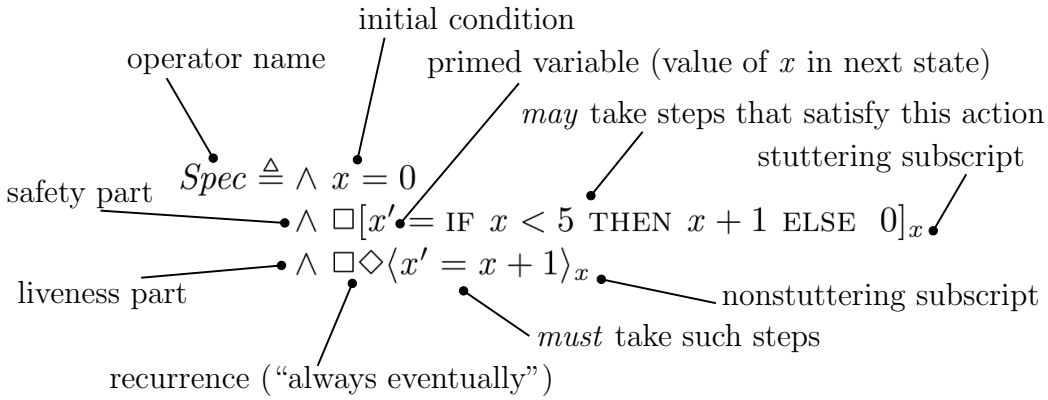


Figure 3.2: Formula anatomy.

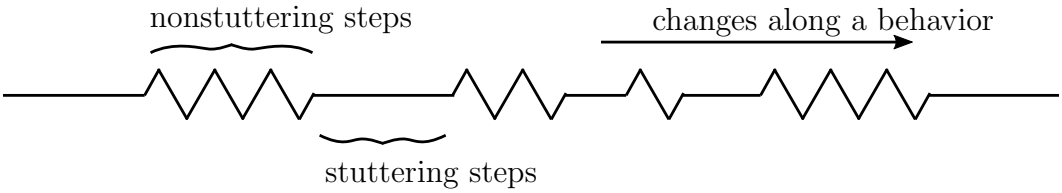


Figure 3.3: Stuttering and nonstuttering steps.

Table 3.1: Distinguishing TLA from raw TLA, TLA from TLA$^+$, with and without past operators. The superscript + signifies the presence of set theory.

	TLA	TLA$^+$
Raw[1]	$\Box(x' = x + 1)$	$\Box(x' \in S)$
Normal[2]	$\Box[x' = x + 1]_x$	$\Box[x' \in S]_{\langle x, S\rangle}$
Raw with past[3]	$\Box(\mathit{Earlier}(x = 1)$ $\Rightarrow (y = 1))$	$\Box(\mathit{Earlier}(x \in S)$ $\Rightarrow (y = 2))$

[1] Stutter-sensitive properties.
[2] Stutter-invariant properties.
[3] Recalling the past relies on indexing, i.e., $\sigma, i \models \varphi$

The thick existential quantifier $\exists\!\!\!\exists$ 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 $\exists\!\!\!\exists x : P(x)$ is satisfied by a behavior if, by introducing stuttering steps, and overwriting the values of the variable x, we can obtain a behavior that satisfies the property $P(x)$. This meaning is illustrated in Fig. 3.4.

In addition to the temporal quantifier $\exists\!\!\!\exists$, which results in a stutter-invariant property when P is stutter-invariant, we denote stutter-sensitive existential

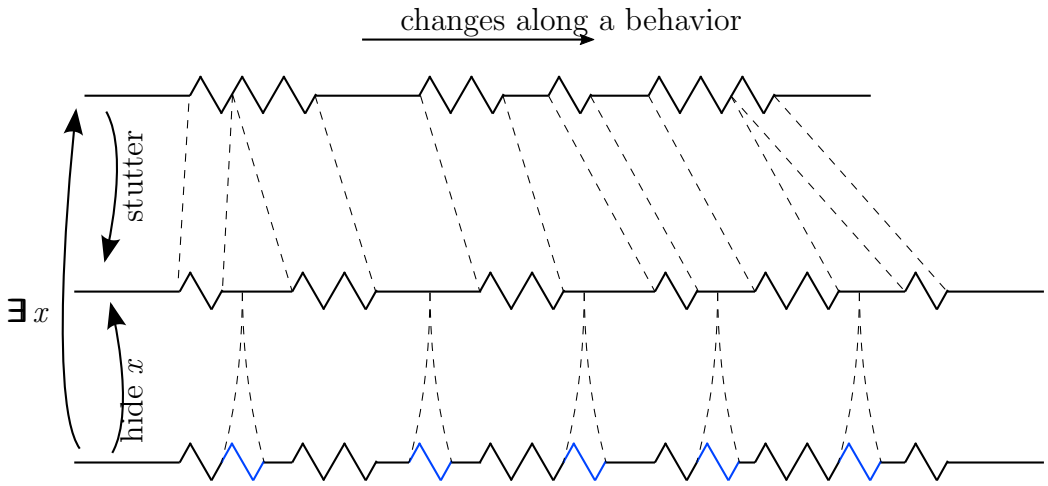


Figure 3.4: Stuttering refinement visualized in the definition of the temporal quantifier $\exists\!\!\exists$.

Table 3.2: Flavors of temporal quantification.

Type	Rigid	Temporal	
		Inv[1]	Raw[2]
Existential	$\exists$	$\exists\!\!\exists$	$\exists\!\!\exists\!\!\mid$
Universal	$\forall$	$\forall\!\!\forall$	$\forall\!\!\forall\!\!\mid$

[1] Stutter-preserving quantification.
[2] Stutter-sensitive quantification.

temporal quantification [95, 91] with the symbol $\exists\!\!\exists\!\mid$.[1] The expression $\exists\!\!\exists\!\mid x : P(x)$ is satisfied by a behavior if by overwriting the values of the variable x, we can obtain a behavior that satisfies the property $P(x)$. This is in contrast to the meaning of $\exists\!\!\exists$ [114]. The temporal quantifier $\exists\!\!\exists\!\mid$ can occur in raw TLA$^+$ expressions, but not in TLA$^+$expressions. Table 3.2 summarizes the different kinds of temporal quantifiers.

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

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

Collection versus set Not every statement in ZF defines a set. Some statements describe collections that are too large to be sets [117, p. 66]. In naive

[1] Stutter-sensitive quantification $\exists\!\!\exists\!\mid$ is defined in the module *TemporalLogic*.

set theory this phenomenon gives rise to Rusell's and other paradoxes [56]. A collection that is not a set is called a proper class [101, 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 [66, p. 65] (within the theory that the semantics is discussed). So to accommodate 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.

3.3 Synthesis of implementations

Synthesis is the algorithmic construction of an implementation that satifies the specification of a component. Let y be a variable that represents the component, and x a variable representing the component's environment. Consider a property described by the temporal formula $Phi(x, y)$. If an implementation of $Phi(x, y)$ exists, then the property is called *realizable*. The notion of realizability [8, 154, 155, 134] can be formalized [66] as follows:

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

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

$$
\begin{array}{l}
IsAFiniteFcn(f) \ \triangleq \ \ \wedge\ IsAFunction(f) \\
\qquad\qquad\qquad\qquad\ \ \wedge\ IsFiniteSet(\text{DOMAIN } f) \\
IsRealizable(Phi(\_,\ \_),\ e(\_,\ \_)) \ \triangleq
\end{array}
$$

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

The operator *IsFiniteSet* requires finite cardinality [117, p. 341]. The expression $IsRealizable(Phi, e)$ means that property *Phi* can be implemented with resources e. The above definition is based on a note by Lamport [113], (see also [110, p. 221]), and formalizes realizability as described in the literature on synthesis [90, §4, pp. 46–47], [23, §2.3, pp. 914–915], with a difference regarding initial conditions.

Tractable liveness A formula described by the schema

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

defines a liveness property categorized as generalized Streett(1), or GR(1) [153]. A formula of this form is useful for expressing the dependence of a component on its environment. Rewriting the above as

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

emphasizes this use case. Usually, the formulas $\neg P_j$ express recurrence properties that the component requires from its environment in order to be able to realize the properties R_i. If the environment lets the behavior satisfy $\Diamond\Box P_j$, then the component cannot and so is not required to satisfy the properties $\Box\Diamond R_i$. As a specification for the gear subsystem of an aircraft, a simple example is $\Box\Diamond(DoorsOpen) \Rightarrow \Box\Diamond(ExtensionRequest \Rightarrow GearExtended)$. This property requires that the gear respond to any request to extend under suitable conditions. A more complete example is described in Chapter 8, and there are several examples of GR(1) specifications for practical applications in the literature [193, 129, 164, 151, 150, 194, 23].

Conjoining k Streett pairs yields a liveness property called GR(k), which can be regarded as a modal conjunctive normal form [158, 130]. Synthesis of a controller that implements a GR(k) property has computational complexity factorial in the number of Streett pairs k [152]. 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 Section 3.4) $m \times n \times S^3$ times, where S the number of states (usually exponential in the number of variables) [23, 178, 179]. 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 [85], so exponentially more expensive than GR(1) synthesis [153]. For this reason GR(1) synthesis algorithms unconditionally add a memory variable that ranges over $1..n$.

3.4 Elements of synthesis algorithms

Reasoning about open systems involves computing from which states a controller exists that can guide the system to a desired set of states. Given a set of states as destination, an *attractor* is the set of states from where such a controller exists. Computing an attractor can be viewed as solving a "multi-step" control problem that iteratively solves a "one-step" control problem [51]. The "one-step" control problem is described by the controllable step operator *Step* (commonly denoted as *CPre*), which is defined as follows:

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

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

$$
\begin{aligned}
&\mathit{GeneralStep}(x,\ y,\ \mathit{Target}(\_,\ \_)) \ \triangleq\ \exists\, y' :\\
&\quad \wedge\ \mathit{SysNext}(x,\ y,\ y')\\
&\quad \wedge\ \forall\, x' :\ \ \vee\ \neg \mathit{EnvNext}(x,\ y,\ x')\\
&\qquad\qquad \vee\ \wedge\ \mathit{AssemblyNext}(x,\ y,\ x',\ y')\\
&\qquad\qquad\quad \wedge\ \mathit{Target}(x',\ y')
\end{aligned}
$$

Remark 1. The expression $\exists y'$ is ungrammatical in TLA$^+$ [117, 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 [112, Sec. 2.2.2 on p. 6]). □

Assume that the state predicate $Goal(\_, \_)$ describes the destination. The states from where a controller exists that can guide the system in at most k steps to some state that satisfies the predicate $Goal(x, y)$ are those that satisfy the following operator [179, 178]:

```
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)
```

Recursive definitions as the above are part of TLA^{+2} [121]. The attractor of $Goal$ is the fixpoint of the k-step attractor operator:

```
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)
```

In this definition $Goal$ is a first-order operator. An attractor definition with $Goal$ being a set is possible too [64, §IV-A]. The above operators are simpler cases of those that we define in later sections. Dual to the attractor is the trap operator, defined as follows:

```
kStepSafe(x, y, Stay(_, _), Escape(_, _), k) ≜
  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)
```

Game solving involves reasoning about sets of states. Symbolic methods using binary decision diagrams (BDDs) [30] 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 [185]. 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.

Chapter 4

CONTRACTS

4.1 Assume-guarantee contracts between components

The purpose of a contract is to represent the assumptions that each component in an assembly makes about other components, and the guarantees that it provides when these assumptions are satisfied. The assignment of obligations to components should be balanced. It is unreasonable to specify an assumption by one component that is infeasible by any other component. So the specifications should suffice for ensuring that the assembly behaves as desired, and also not overconstrain any of the components. We can view these requirements as placing a lower and an upper bound on component specifications. The lower bound ensures that each component is implementable, and the upper bound ensures that the assembled system operates correctly. These requirements are formalized with the following definition. A *contract* [67] is a partition of variables among n components, defined by n actions

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

and a collection of temporal properties

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

that satisfy the following theorem schema:

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

In other words, a contract is a collection of assume-guarantee properties for each component that are realizable and conjoined imply the desired behavior for the system assembled from those components. The notion of composition of properties is illustrated in Fig. 4.1.

Remark 2. Two alternative definitions are possible. Instead of using the letters $A, \ldots, W$ we can use an index j, with the understanding that j is part

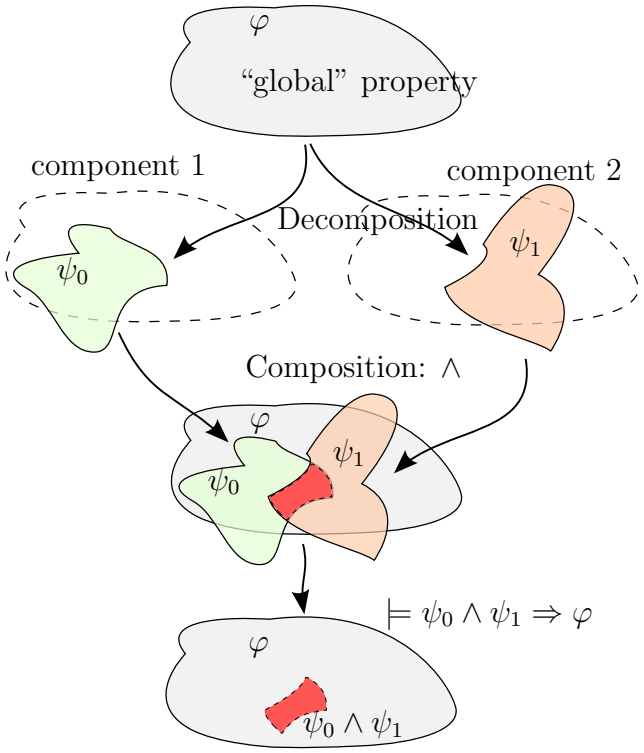


Figure 4.1: Each component is specified by a property that may allow behaviors that violate the desired global property. These undesired behaviors would be caused by arbitrary behavior of other components. Nonetheless, when we conjoin the component specifications of the contract, the result implies the global property, because of mutual fulfillment of assumptions between components.

of the identifier, not a value in the object language (here TLA$^+$) [93]. Another approach is to incorporate the index within the object language. This can be done by defining a single property parametrized by an index, for example $Prop(\_, \_, \_)$ and $eR(\_, \_, \_)$. The schema can then be written as

$$
\begin{array}{lll}
\wedge\, \forall\, j \in 1\,..\,n: & \text{LET} & R(u,\, v) \triangleq Prop(u,\, v,\, j) \\
 & & e(u,\, v) \triangleq eR(u,\, v,\, j) \\
 & \text{IN} & IsRealizable(R,\, e) \\
\wedge\, \mathbf{\forall}\, x,\, y: & & \\
\quad (\forall\, j \in 1\,..\,n: & & Prop(x[j],\, y[j],\, j)) \Rightarrow Phi(x,\, y)
\end{array}
$$

□

Remark 3. 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 [170, 22, 21, 20]. 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 $(A, \ldots)$ 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 thesis, consider a charging station for mobile robots that has two spots, and a robot that requests a spot for charging. The charging station keeps track of which spots are taken (variables $spot\_1, spot\_2$), as well as the spot number that becomes available for the robot to dock ($free\_x, free\_y$), when the variable $free = 1$. The robot can request docking by setting the variable req, and is represented by its coordinates on the plane pos_x, pos_y. Not all spots are free. One other spot is occupied by another robot, which forms part of the environment of the charging station and the robot. This spot is indicated by the variable occ, and to keep the example small, occ remains unchanged through time, so the occupied spot does not change, but neither the station nor the robot control which this spot is. The specification of the entire system is the following:

$$
\begin{array}{l}
\textsc{extends}\ \mathit{Integers} \\
\textsc{variables}\ \mathit{spot\_1},\ \mathit{spot\_2},\ \mathit{free\_x},\ \mathit{free\_y},\ \mathit{free}, \\
\qquad\qquad \mathit{req},\ \mathit{pos\_x},\ \mathit{pos\_y},\ \mathit{occ},\ \mathit{turn} \\
\mathit{station\_vars} \triangleq \langle \mathit{spot\_1},\ \mathit{spot\_2},\ \mathit{free\_x},\ \mathit{free\_y},\ \mathit{free} \rangle \\
\mathit{robot\_vars} \triangleq \langle \mathit{req},\ \mathit{pos\_x},\ \mathit{pos\_y} \rangle \\
\mathit{vars} \triangleq \langle \mathit{station\_vars},\ \mathit{robot\_vars},\ \mathit{occ},\ \mathit{turn} \rangle \\
\mathit{StationStep} \triangleq \ \land\, \mathit{turn} = 1 \\
\qquad\qquad\qquad\quad \land\, (\mathit{req} = 0) \Rightarrow (\mathit{free}' = 0) \\
\mathit{StationNext} \triangleq \\
\quad \land\, \mathit{spot\_1} \in 0 \,..\, 1 \land \mathit{spot\_2} \in 0 \,..\, 1 \land \mathit{free} \in 0 \,..\, 1 \\
\quad \land\, \mathit{free\_x} \in 0 \,..\, 18 \land \mathit{free\_y} \in 0 \,..\, 18 \\
\quad \land\ \lor (\mathit{free} = 0) \\
\qquad \lor (\mathit{free\_x} = 1 \land \mathit{free\_y} = 1 \land \mathit{spot\_1} = 0) \\
\qquad \lor (\mathit{free\_x} = 2 \land \mathit{free\_y} = 1 \land \mathit{spot\_2} = 0) \\
\quad \land (\mathit{free} = 1) \Rightarrow \ \land\, \mathit{spot\_1} = 0 \ \Rightarrow\ \mathit{occ} \neq 1 \\
\qquad\qquad\qquad\quad \land\, \mathit{spot\_2} = 0 \ \Rightarrow\ \mathit{occ} \neq 2
\end{array}
$$

$\land\ StationStep \lor \text{UNCHANGED}\ station\_vars$

$RobotNext \triangleq$
$\land\ pos\_x \in 1 \,..\, 15 \land pos\_y \in 1 \,..\, 15 \land req \in 0 \,..\, 1$
$\land\ ((req = 1 \land req' = 0) \Rightarrow \land\ free = 1$
$\land\ free\_x = pos\_x'$
$\land\ free\_y = pos\_y')$
$\land\ (turn = 2) \lor \text{UNCHANGED}\ robot\_vars$

$OthersNext \triangleq (occ \in 1 \,..\, 3) \land (occ' = occ)$
$SchedulerNext \triangleq \land\ turn \in 1 \,..\, 2$
$\land\ (turn = 1) \Rightarrow (turn' = 2)$
$\land\ (turn = 2) \Rightarrow (turn' = 1)$

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

We wrote the property *Assembly* using implication. In general, the operator *Unzip* (defined in Section 4.2.2) 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 Section 4.2). 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. Let the actions

$eStation(turn) \triangleq (turn' \neq turn)$
$eRobot(turn) \triangleq (turn' \neq turn)$

define when each implementation takes nonstuttering steps. Note that the property *Assembly* defines synchronous and interleaving changes to the components. This is the case for the implementations too, due to the presence

of variable *turn* in *eStation* and *eRobot*, together with the antecedent *Env* in formula *Phi*.

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

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

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

$$
\begin{array}{l}
IsRealizable(PhiR(\_,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_),\ eR(\_)) \ \triangleq \\
\quad \exists\, f\_req,\ f\_pos\_x,\ f\_pos\_y,\ g, \\
\qquad req0,\ pos\_x0,\ pos\_y0,\ mem0 : \\
\quad\ \land\ IsAFiniteFcn(g) \land IsAFiniteFcn(f\_req) \\
\quad\ \land\ IsAFiniteFcn(f\_pos\_x) \land IsAFiniteFcn(f\_pos\_y) \\
\quad\ \land\ \mathbf{\forall}\, req,\ pos\_x,\ pos\_y,\ free\_x,\ free\_y,\ free,\ turn : \\
\qquad \text{LET}\ R(mem) \ \triangleq \\
\qquad\quad \text{LET}\ v \ \triangleq\ \langle mem,\ req,\ pos\_x,\ pos\_y,\ free\_x, \\
\qquad\qquad\qquad\qquad free\_y,\ free,\ turn\rangle \\
\qquad\qquad\quad N \ \triangleq\ \land\ mem' = g[v] \\
\qquad\qquad\qquad\qquad\ \land\ req' \ \ = f\_req[v] \\
\qquad\qquad\qquad\qquad\ \land\ pos\_x' = f\_pos\_x[v] \\
\qquad\qquad\qquad\qquad\ \land\ pos\_y' = f\_pos\_y[v] \\
\qquad\qquad\quad e \ \triangleq\ eR(turn) \\
\qquad\quad \text{IN}\ \ \land\ mem = mem0 \land req = req0 \\
\qquad\qquad\qquad\ \land\ pos\_x = pos\_x0 \land pos\_y = pos\_y0 \\
\qquad\qquad\qquad\ \land\ \Box[e \Rightarrow N]_v \\
\qquad\qquad\qquad\ \land\ \mathrm{WF}_{\langle mem,\, req,\, pos\_x,\, pos\_y\rangle}(e \land N) \\
\qquad \text{IN}
\end{array}
$$

$$(\exists\, mem :\ R(mem))$$
$$\Rightarrow PhiR(req,\ pos\_x,\ pos\_y,\ free\_x,$$
$$free\_y,\ free,\ turn)$$

□

Remark 4. Realizability can be defined without design of initial conditions for the component (the $\exists req0, pos\_x0, pos\_y0$ above). With such a definition, the property *PhiR* should not constrain the component's initial state, because that would render *PhiR* unrealizable [66, §3.3, pp. 14–16]. So initial conditions within *PhiR* would be part of an antecedent. In consequence, a conjunction of realizable component specifications would not imply any closed-system property. The choice between these different definitions is a matter of specification style. □

4.2 Open systems

4.2.1 Defining an open system

An open system is one constrained with respect to variables it does not control [6], [90, §3.1], [114, §9.5.3]. Usually the components we build rely on their environment in order to operate as intended. For example, a laptop should be able to 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 *open* system can be defined mathematically using the notion of proper class. As remarked earlier, any temporal property defines a proper class of behaviors. So any system defined by a temporal property is modeled by a proper class of behaviors. This indicates that the notion of open system is *relative* to some designated variables. These observations motivate the following definition.

$$IsAClosedSystem(P(\_)) \triangleq$$
$$\exists\, S : \forall\, v : \; P(v) \Rightarrow \Box(v \in S)$$
$$IsAnOpenSystem(P(\_)) \triangleq \neg IsAClosedSystem(P)$$

THEOREM

ASSUME TEMPORAL $P(\_)$

PROVE $IsAnOpenSystem(P) \equiv$

$$\forall\, S : \exists\, v : \; P(v) \wedge \Diamond(v \notin S)$$

Let x be a variable and $P(x)$ a temporal property. The property $P(x)$ defines an *open system* if and only if $IsAnOpenSystem(P)$. So the possibility of *diverging behavior* characterizes a system as open. A property P defines a closed-system if it implies a type invariant that bounds all variables that occur in P. Therefore, closed systems can be defined using Δ_0 formulas [100, p. 161]. Diverging behavior is also the main concept in how initial conditions affect realizability [66, Lemma 6, p. 12].

Remark 5. Thus, cardinality distinguishes an open from a closed system; a criterion applicable even if we decide to restrict our attention to only finitely many states. Considering a system closed if its variables take values from a finite set, an open system is one whose variables take values from an infinite set. □

4.2.2 Specifying interaction with an environment

A component's specification should not constrain its environment. This is expressible with a formula that spreads implication incrementally over a behavior [7, 96, 10, 11, 88, 141]. We define the operator *Unzip* for forming open-systems from closed-systems

$$WhilePlusHalf(A(\_,\_),\, G(\_,\_),\, x,\, y) \triangleq$$
$$\begin{array}{lll}
\forall\, b : & \vee \neg \wedge\, b \in \text{BOOLEAN} & \wedge \Box[b' = \text{FALSE}]_b \\
& \wedge \exists\, u,\, v : & \wedge A(u,\, v) \\
& & \wedge \Box \vee b \neq \text{TRUE} \\
& & \vee \langle u,\, v\rangle = \langle x,\, y\rangle \\
& \vee \exists\, u,\, v : & \wedge G(u,\, v) \wedge (v = y) \\
& & \wedge \Box \vee b \neq \text{TRUE} \\
& & \vee \langle u,\, v\rangle = \langle x,\, y\rangle \\
& & \wedge \Box[(b = \text{TRUE}) \Rightarrow (v' = y')]_{\langle b,\, v,\, y\rangle}
\end{array}$$

$Unzip(P(\_, \_), x, y) \triangleq$

LET $Q(u, v) \triangleq P(v, u)$ swap back to x, y

$A(u, v) \triangleq WhilePlusHalf(Q, Q, v, u)$ swap to y, x

IN $WhilePlusHalf(A, P, x, y)$

The operator *WhilePlusHalf* is a slight variant of how the "while-plus" operator $\overset{+}{\triangleright}$ can be defined within TLA$^+$ [117, p. 337], [11, p. 262] ($\overset{+}{\triangleright}$ is defined by TLA$^+$ semantics [117, p. 316]). If A, G are temporal operators, then $WhilePlusHalf(A, G)$ can be thought of as being true of a behavior σ if every finite prefix of σ that can be extended to a behavior that satisfies A can also be extended, starting with a state that satisfies $v = y$, to a behavior that satisfies G.

Remark 6 (Comparison to $\overset{+}{\triangleright}$)**.** Only v is constrained in the first state of the suffix, thus the "half" in the name. In contrast, the operator $\overset{+}{\triangleright}$ constrains both u and v in the first state of the suffix. For disjoint-state specifications, this additional constraint results in unrealizability, using the definition of synthesis from Section 3.3. To obtain a realizable property, the property G should be sufficiently permissive [66, §5.2.4, pp. 26–27]. However, this leads to possible underspecification of what $\exists\, u, v : G(u, v) \ldots$ means (i.e., the closure of G may allow behavior undetermined by the axioms of the logic). These observations motivate the above modification, which corresponds to definitions of "strict implication" from the literature on games [23]. □

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

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

Writing a closed-system specification is typically easier than reasoning about how to separate it into two properties A, G. More fundamentally, the environment behavior should be mentioned in its entirety within G. Otherwise, the disjunct that contains G can allow the component to behave as if the environment has arbitrary future behavior. How *Unzip* is defined is reminiscent of

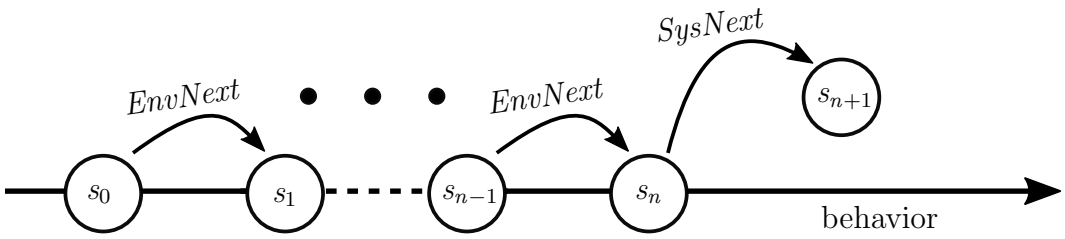


Figure 4.2: The *stepwise principle* of taking one more step.

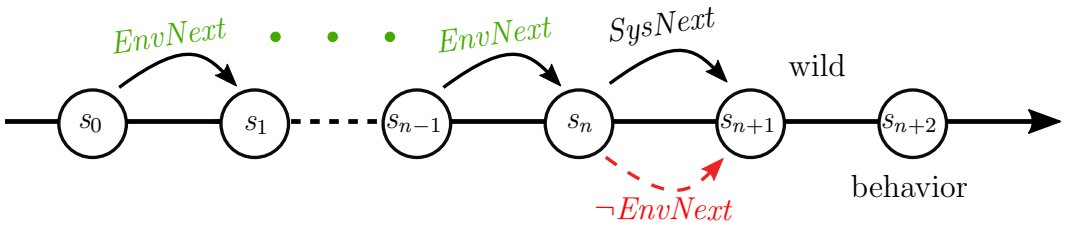


Figure 4.3: The component can behave arbitrarily *after* the environment takes an erroneous step.

how $\stackrel{+}{\triangleright}$ is defined for safety properties in terms of $\rightarrow\!\triangleright$ [11, p. 262], [10, Prop. 1, p. 501].

For the synthesis of implementations for properties specified using *Unzip*, relating this operator to existing results about synthesis from GR(1) properties is useful [23]. This is possible via the following definition in raw TLA$^+$ with past

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

The second conjunct expresses "stepwise implication" (Figs. 4.2 and 4.3), so that if at some step the environment violates the assumed action *EnvNext*, then the system is not obliged to satisfy the action *SysNext* in later steps. The operator *AsmGrt* is a modification of [96] to avoid circularity [67], [88, §5, $\triangleright$ on p. 59]. The operator *Earlier* abbreviates the composition of the past LTL operators *WeakPrevious* and *Historically* (expressible in TLA$^+$ using temporal

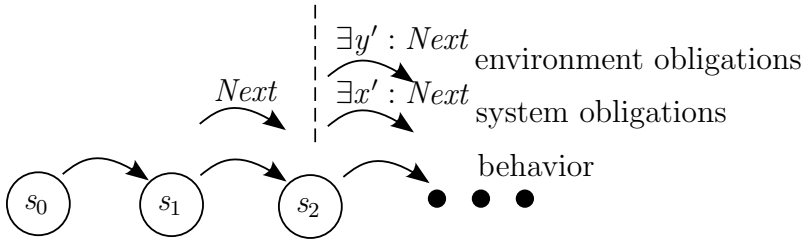


Figure 4.4: Defining an open-system from a closed-system property with action *Next*, using the operator *Unzip*.

quantification and history variables). It is possible to define *Earlier* by using a modified satisfaction relation $\models$ in raw TLA$^+$, but we will omit this definition here.

It is proved in Section 9.4.2 that if $P \equiv \mathit{Init} \wedge \Box[\mathit{Next}]_{\langle x,y\rangle} \wedge L$, where L a GR(1) liveness property, and the pair of properties $\mathit{Init} \wedge \Box[\mathit{Next}]_{\langle x,y\rangle}, L$ is machine-closed (meaning that L does not constrain the safety in the property $\mathit{Init} \wedge \Box[\mathit{Next}]_{\langle x,y\rangle}$ [7]), then an implementation synthesized for the property

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

also realizes $\mathit{Unzip}(P, x, y)$. Existing algorithms can be used to synthesize such an implementation [23]. The above form is illustrated in Fig. 4.4.

Remark 7. Open systems can be defined also in other ways, for example using game graphs together with liveness formulas [90], game structures, alternating-time temporal logic formulas [16], or modules [186]. The property P in the formula $\mathit{Unzip}(P, x, y)$ corresponds to the graph and liveness in a game graph description. □

Remark 8 (Symmetry)**.** Dijkstra requires symmetry from solutions to the mutual exclusion problem [54, item (a)]. The approach we follow asserts that all components are implemented as "Moore machines" (f and g are independent of primed variable values). Alternatives are possible where one component is Mealy and its environment Moore [23, 99]. 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 [51]. □

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 $P \triangleq Env \wedge Assembly$ and let $Unzip(P, spot\_1, \ldots, turn, req, pos\_x, pos\_y)$ 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. □

We consider specifications that are interleaving [117, p. 137] among most of the components involved (the scheduler is an exception), in that they allow variables of only one component to change in each step (a "move"). Components move in a fixed order that repeats, so the resulting interaction can be viewed as a turn-based game between the components. Each variable is controlled by a single component throughout time, and thus the specifications are disjoint-state [117, p. 144]. Let C be a collection of indices that identify components, and e_i be actions that attribute state changes to components (as the e in *Realization*, Section 3.3), and v a tuple of relevant variables. The concept of interleaving can be defined as the following pairwise orthogonality condition [7, p. 514], [11], [66, §5, p. 22]:

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

In words, no step changes the state of more than one component. Let V be a collection of indices of variable identifiers x_k, and $e_{j,k}$ attribute changes of variable x_k to component j. The concept of disjoint state can be defined as follows [66, §5.1.2, p. 24]:

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

In words, all changes to each variable are attributed to no more than one component. These definitions are schematic, in that i, j, k are metatheoretic notation [93].

Chapter 5

PARAMETRIZED HIDING OF VARIABLES

5.1 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 [115]. 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. This objective can be summarized as follows:

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

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

Synthesizing implementations from specifications with partial information is computationally hard. Since the hidden variables are chosen to preserve realizability, and reasoning about hidden behavior at the time of component synthesis is computationally expensive, we eliminate the hidden variables. We

show that the resulting specifications can be written as if only the visible variables were declared to the component, thus as if the component had full information; an objective summarized as follows.

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

In Section 5.3 we discuss the abstraction from the controllable step operator for specific variables, and in Section 5.4 we parameterize the *choice* of which variables to hide. We start by considering the safety part of the specification in Section 5.2.

5.2 Preventing safety violations

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

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

where the conjunct *Liveness* is a conjunction of liveness properties, for example $\Box\Diamond Goal_1 \wedge \Box\Diamond Goal_2$. The property *Liveness* can impose constraints on the safety property $SM \triangleq Init \wedge \Box[Next]_{vrs}$. If this is not the case, then the pair of properties $SM, Liveness$ is called *machine-closed* [2, p. 261], [7, p. 519]. We are about to decompose the property *Assembly*. To avoid circularity, we create the safety and liveness parts of component properties separately, in two stages. Only "pieces" of liveness constraints will end up in each component's specification. This means that we should ensure that the safety part is strong enough to prevent any component from "straying away" to an extent that would violate the assembly's *Liveness* property.

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

$$\models Assembly \Rightarrow W.$$

The property W is known as *closure* of the property *Assembly* [13, p. 120], [7, p. 518], [11, pp. 261–262], due to topological considerations [12]. If W is written in the form

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

then the invariant *Inv* defines the largest set of states that can occur in any behavior that satisfies the property *Assembly*. The weakest invariant also yields the (unique) weakest safety assumption necessary in turn-based games with full information (set of cooperatively winning states) [33], [67, §III-A].

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

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

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

A component's action should constrain the next values of only variables that the component controls. In addition, the component should be constrained to preserve the invariant *Inv*. The property W can be written as $Init \wedge Inv \wedge \Box[WNext]_{vrs}$, where [114, by INV2, Fig.5, p. 888]

$$WNext \ \triangleq \ Inv \wedge Next \wedge Inv'.$$

The property $Unzip(Assembly, x, y)$ can be defined by quantifying the primed variables of other components, using the following operators as arguments of the operator *AsmGrt* (defined in Section 4.2.2):

$$EnvNext(x,\ y,\ x') \ \triangleq$$

$$
\begin{aligned}
&[\exists\, y' :\ \ \wedge \mathit{Inv}(x,\, y) \wedge \mathit{Inv}(x',\, y') \\
&\qquad\qquad \wedge \mathit{Next}(x,\, y,\, x',\, y')]_x \\
&\mathit{SysNext}(x,\, y,\, y') \ \triangleq \\
&\quad [\exists\, x' :\ \ \wedge \mathit{Inv}(x,\, y) \wedge \mathit{Inv}(x',\, y') \\
&\qquad\qquad \wedge \mathit{Next}(x,\, y,\, x',\, y')]_y
\end{aligned}
$$

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 $\mathit{Inv} \wedge \mathit{Next}$ is satisfied by each step. The reason is that in each step, at most one component can change in a non-unique way. See also Section 9.5 for more details.

Remark 9. 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 $\Box\Diamond\langle \mathit{SchedulerNext}\rangle_{\mathit{turn}}$ reduces to safety, because any nonstuttering step of the assembly changes the variable *turn*. □

5.3 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, which leads to the conjunction

$$
\begin{aligned}
&\wedge\, \exists\, h :\ \mathit{Bound}(h,\, x,\, y) \\
&\wedge\, \forall\, h :\ \mathit{Bound}(h,\, x,\, y) \Rightarrow P(h,\, x,\, y,\, y').
\end{aligned}
$$

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 Chapter 3, 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*)

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

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

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

Algebraic manipulation yields

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

If *Target* is independent of h', (which is the case in Chapter 6), then confining universal quantification to the first disjunct yields

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

By defining

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

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

we obtain

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

The resulting operator *StepH* is schematically the same with that for the full information case. So the open-system specification with hidden variables can be rewritten as an open-system specification with no hidden variables, without changing the set of states from where the property is realizable. The eliminated variables do not appear in the component specification, so further work focusing on that component can be carried out in a full information context, including GR(1) synthesis. The action *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) [50], [86, Eq. (4.14), p. 325].

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

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

$\lor\ (occ = 1) \land (occ' = 1) \land (req' = 1)$

$\lor\ (occ = 2) \land (occ' = 2) \land (req = 0)$

$\lor\ (occ = 2) \land (occ' = 2) \land (req' = 1)$

$\lor\ (occ = 3) \land (occ' = 3) \land (req = 0)$

$\lor\ (occ = 3) \land (occ' = 3) \land (req' = 1)$

$\land\ InvH$

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

$\land\ turn = 2 \land turn' = 1 \land free \in 0\,..\,1$

$\land\ free\_x \in 0\,..\,18 \land free\_y \in 0\,..\,18$

$\land\ occ \in 1\,..\,3 \land occ' \in 1\,..\,3$

$\land\ req \in 0\,..\,1 \land req' \in 0\,..\,1$

$\land\ spot\_1 \in 0\,..\,1 \land spot\_2 \in 0\,..\,1$

$\land\ \lor\ (free = 1) \land (occ = 1) \land (occ' = 1)$

$\lor\ (free = 1) \land (occ = 3) \land (occ' = 3)$

$\lor\ \land\ (free = 1) \land (occ \in 1\,..\,2) \land (occ' = 2)$

$\land\ (spot\_1 = 0)$

$\lor\ (occ = 1) \land (occ' = 1) \land (req = 0)$

$\lor\ (occ = 1) \land (occ' = 1) \land (req' = 1)$

$\lor\ (occ = 2) \land (occ' = 2) \land (req = 0)$

$\lor\ (occ = 2) \land (occ' = 2) \land (req' = 1)$

$\lor\ (occ = 3) \land (occ' = 3) \land (req = 0)$

$\lor\ (occ = 3) \land (occ' = 3) \land (req' = 1)$

$\land\ InvH$

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

$InvH \triangleq$

$\land\ turn \in 1\,..\,2 \land free \in 0\,..\,1$

$\land\ free\_x \in 0\,..\,18 \land free\_y \in 0\,..\,18$

$\land\ occ \in 1\,..\,3 \land spot\_1 \in 0\,..\,1 \land spot\_2 \in 0\,..\,1$

$\land\ \lor\ \land\ (free\_x = 1) \land (free\_y = 1) \land (occ \in 2\,..\,3)$

$\land\ (spot\_1 = 0) \land (spot\_2 = 1)$

$$
\begin{array}{ll}
\lor & \land (free\_x = 2) \land (free\_y = 1) \land (occ = 1) \\
& \land (spot\_1 = 1) \land (spot\_2 = 0) \\
\lor & \land (free\_x \in 1 \,..\, 2) \land (free\_y = 1) \land (occ = 3) \\
& \land (spot\_1 = 0) \land (spot\_2 = 0) \\
\lor & (free = 0) \\
\lor & \land (free\_x = 2) \land (free\_y = 1) \land (occ = 3) \\
& \land (spot\_2 = 0)
\end{array}
$$

The concept of a care predicate will be described in Chapter 7. □

5.4 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 Section 5.3, 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

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

The rigid variable m models the availability or lack of information. Following Section 5.3, we replace h with the selector expression to define a controllable step operator with parametrized hiding as follows (where variable v is h for the case that $m = \text{FALSE}$, meaning h visible):

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

$\land \quad R(v,\, x,\, y,\, m)$

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

$PrmStep(v,\, x,\, y,\, m) \equiv$

LET

$MskInv(h) \triangleq$

LET $r \triangleq Mask(m,\, v,\, h)$

IN $Inv(r,\, x,\, y)$

$PrmInv \triangleq \exists\, h :\ MskInv(h)$

$MskSysNext(h,\, y') \triangleq$

LET $r \triangleq Mask(m,\, v,\, h)$

IN $SysNext(r,\, x,\, y,\, y')$

$PrmSysNext(y') \triangleq$

$\land\, PrmInv$

$\land\, \forall\, h :\ MskInv(h) \Rightarrow MskSysNext(h,\, y')$

$MskEnvNext(h,\, v',\, x') \triangleq$

LET $r \triangleq Mask(m,\, v,\, h)$

IN $EnvNext(r,\, x,\, y,\, v',\, x')$

$PrmEnvNext(v',\, x') \triangleq$

$\exists\, h :\ MskInv(h) \land MskEnvNext(h,\, v',\, x')$

IN

$PrmStep(v,\, x,\, y,\, m) \triangleq \exists\, y' :\ \forall\, x',\, v' :$

$\land \quad PrmSysNext(y')$

$\land \quad PrmEnvNext(v',\, x') \Rightarrow Target(v',\, x',\, y',\, m)$

The LET expressions can be implemented either with syntactic substitution of bitvector formulas (provided the variables v and h can take the same values, and compatible type hints are declared for them to aid bitblasting), or existential quantification. We use existential quantification. The operator *PrmStep* can be rearranged to obtain an equivalent result with new actions and the full information *Step*, as in Section 5.3. The assumption that *Target* does not depend on v', which enables the rewriting, holds only for $m = \text{TRUE}$, so this rewriting takes place for specific variables, after the parametrization has been used to select what variables to hide, as described in Chapter 6.

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 [142]. 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.

5.5 Eliminating hidden variables

In Section 5.3 we defined the operator *SimplerEnvNext* as

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

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

These observations arise because we cannot define the same operator twice. A nullary operator stands for the expression on the right hand side of its definition [117, p. 319]. For example, the definition $f \triangleq x^2$ defines the nullary operator f to be the expression x^2. We may define $g \triangleq x \times x$ and prove that $\models (x \in \mathit{Nat}) \Rightarrow (f = g)$ under the usual definitions of superscript and $\times$, but f and g are defined to be different expressions. The act of defining symbols, and how this act relates to declaring symbols as constants and introducing

axioms about those symbols can be understood as extending a formal theory by definitions [93, §74, Vol. 1, p. 405].

Chapter 6

DECOMPOSING A SYSTEM INTO A CONTRACT

6.1 Overview

The decomposition algorithm takes an (open or closed) system specification and produces open-system specifications for designated components. A component means a collection of variables. We assume that the specification allows components to stutter when variables from other components change. So component interaction is synchronous, in that nonstuttering environment steps are noticed by the component implementation, but the components are not required to react immediately to changes. This assumption is useful for transitions between interconnection architectures (Section 6.7.3).

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

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

where the chaining is established by the implications

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

Conjoining the above specifications, we can deduce the recurrence properties

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

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

There is no notion of a "liveness assumption" in the context of a single component specification. Viewing liveness only as a "guarantee" agrees with real world practice: there is no point in a behavior where we can decide that the liveness assumption "has been violated" [7, 10]. Liveness "assumptions" are meaningful in the context of multiple components, specified by multiple temporal properties, a situation similar to possibility properties [116], [117, §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 $F \stackrel{+}{\triangleright} G$ is a safety property [11, §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 [157], forming a *proof lattice* [149]. Mutual dependence of safety properties is admissible because it is possible to spread implication in a "stepwise" fashion over a behavior, as in the operator *WhilePlusHalf*. So what appears circular for safety properties is a well-founded chain of implications crisscrossing between components. Unlike safety properties, liveness properties allow arbitrary deferment of obligations to the future. This deferment is what allows circularity to arise when liveness properties are mutually dependent. Thus, in order to obtain sound conclusions about liveness properties of an assembly, there should be no cycles of dependence among liveness properties guaranteed by different components [175].

All the discussion that follows focuses on liveness and omits the safety part of specifications. Safety is addressed by closure and computation of component actions as described in Chapter 5. In the computations, safety is taken into account in the *Step* operator within the *Attractor* and *Trap* operators.

6.2 The basic algorithm

Consider two components 1 and 2. Suppose that we want their assembly to satisfy the property $L \triangleq \Box\Diamond Goal$. 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. 6.1. For the objective *Goal*, the set

$$A \triangleq Attractor_1(Goal)$$

contains those states from where component 1 can controllably lead the assembly to the *Goal*. Component 1 cannot ensure that *Goal* will be reached

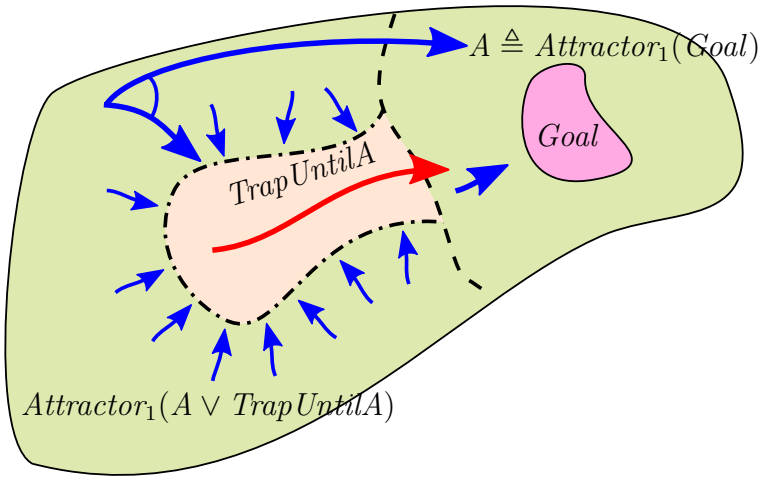


Figure 6.1: The basic idea of the approach.

from outside A. So we need to relax the requirement $\Box\Diamond Goal$ on component 1, by disjoining another liveness property. Suppose that we could find a set $TrapUntilA$ from where component 2 can reach A *and* component 1 can keep the assembly inside $TrapUntilA$ until A is reached. We can then write the liveness specifications

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

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

$$C \triangleq Attractor_1(A \vee TrapUntilA)$$

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

$$L_1 \wedge L_2 \Rightarrow \Box\Diamond Goal$$

the assembly will operate as desired.

We could have simply found $Attractor_2(A)$ (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) [67]. The construction described

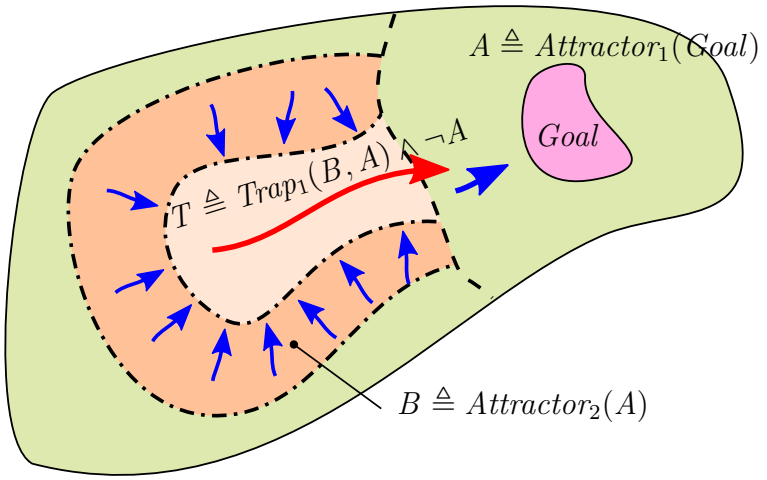


Figure 6.2: How traps are constructed (simple case).

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

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

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

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

The above is a shorthand for the trap operator defined in Section 3.4, 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

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

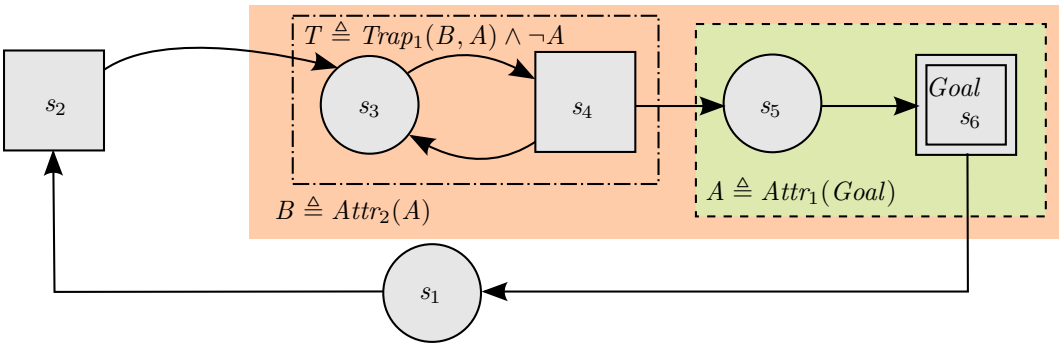


Figure 6.3: An example where a trap is found.

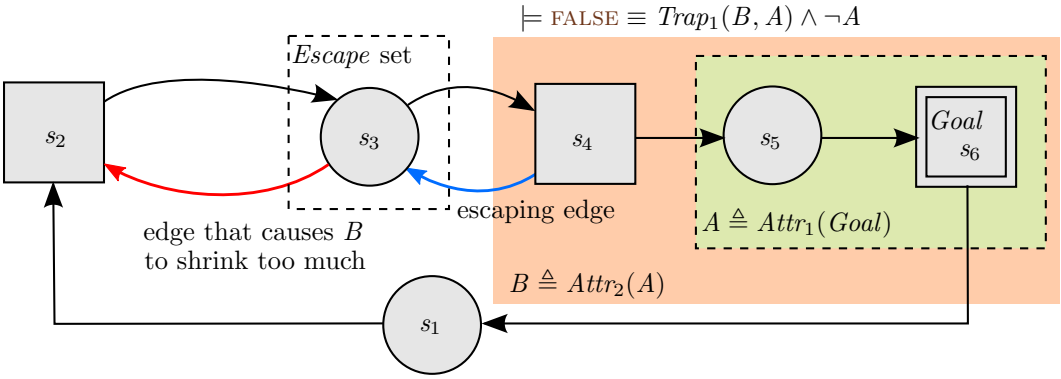


Figure 6.4: The simple approach cannot find a trap in this example. Compared to Fig. 6.3, the failure is due to the edge $\langle s_4, s_3 \rangle$.

These sets are illustrated in Fig. 6.2. Letting $TrapUntilA \triangleq T$, we obtain realizable properties L_1, L_2 (the full specifications include safety, initial conditions, and are defined using *Unzip*, but this section focuses on the liveness parts).

The algorithm we described derives from an earlier version for the case without hidden variables [67, 65]. Covering all initial conditions of the assembly is not possible in general [67, §III-B], unless either safety is restricted, or a syntactic fragment larger than GR(1) is used [67, §IV-A], which is equivalent to using auxiliary variables hidden by temporal quantification.

6.3 Finding assumptions in more cases

Forming a trap is the key step for constructing liveness assumptions. But the approach of Section 6.2 can fail to find a trap, even in cases when our intuition suggests otherwise. The reason is too small a set B, causing $Trap_1(B, A)$ to be empty. We use an example to explain why, and then a solution.

Example 5. Consider the graph shown in Fig. 6.3. Component 1 chooses the next node when at a disk, and component 2 when at a box. A trap is found

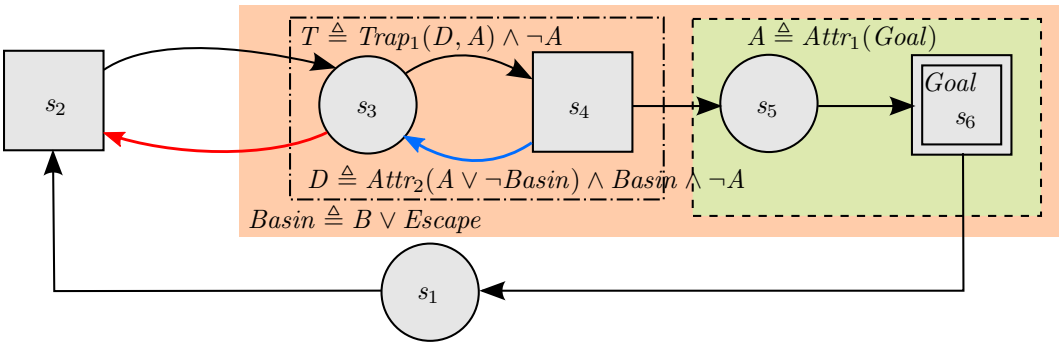


Figure 6.5: Including the states where component 2 can escape allows finding the trap suggested by the specifier's intuition.

for Fig. 6.3, because component 2 can reach A from both nodes s_3 and s_4. Fig. 6.4 shows a modification with the edge $\langle s_3, s_2 \rangle$ added. No trap is found in this case, because $B \wedge \neg A$ contains only node s_4, so component 2 can move "backwards" from s_4 to s_3. So a larger B is needed, but why did B shrink compared to Fig. 6.3?

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. 6.5. 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 implication

$$T \Rightarrow Basin$$

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

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

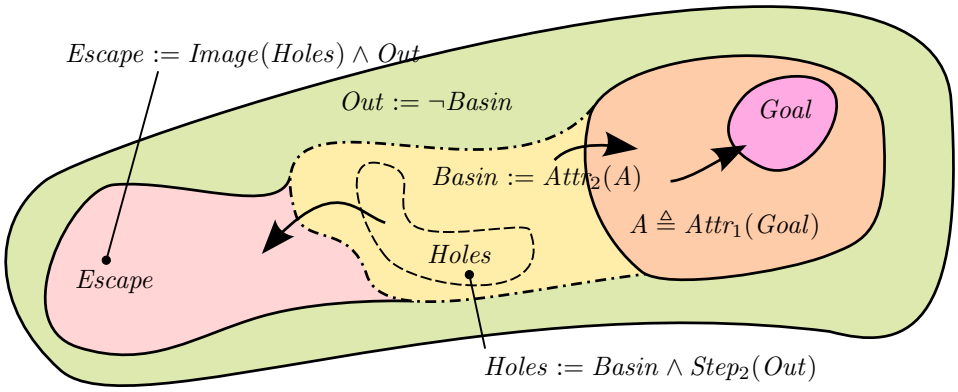


Figure 6.6: Collecting escapes that can cause a trap set to not form.

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

$$D \triangleq Attr_2(A \vee \neg Basin) \wedge \neg(A \vee \neg Basin)$$

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

$$T \triangleq Trap_1(D, A) \wedge \neg A$$

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

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

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 Chapter 8. □

The above discussion referred to individual states. A symbolic approach relies on manipulating collections of states. Fig. 6.6 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

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

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

$$Escape \ := Out \wedge Image(Holes),$$

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

$$Image(x,\ y,\ Source(\_,\ \_),\ Next(\_,\ \_,\ \_,\ \_)) \ \triangleq$$
$$\quad \exists\, p,\ q :\ Source(p,\ q) \wedge Next(p,\ q,\ x,\ y)$$

The resulting *Basin* is used for computing $D := Attr_2(A \vee \neg Basin) \wedge Basin \wedge \neg A$ and $Trap_1(D, A) \wedge \neg A$. If the latter is nonempty, then we have found a trap. Otherwise, the above computation is iterated using the larger *Basin* as described in the following sections.

6.4 Taking observability into account

So far we ignored that each component observes different information. What information is available depends on the parameter values (Chapter 5). 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:

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

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

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 (Section 5.3). Thus, we can apply the *Attractor* and *Trap* operators. The sets of states when observability is taken into account are shown in Fig. 6.7. The

indices correspond to components, with the mask parameters that correspond to each of them. The main difference with Section 6.3 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 1 (Soundness)**.** ASSUME *: The sets of states D and T are computed as in Fig. 6.7.* 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 1 implies that component 1 cannot prevent component 2 from reaching $\neg D$. So it ensures that component 1 cannot stay forever within T, and

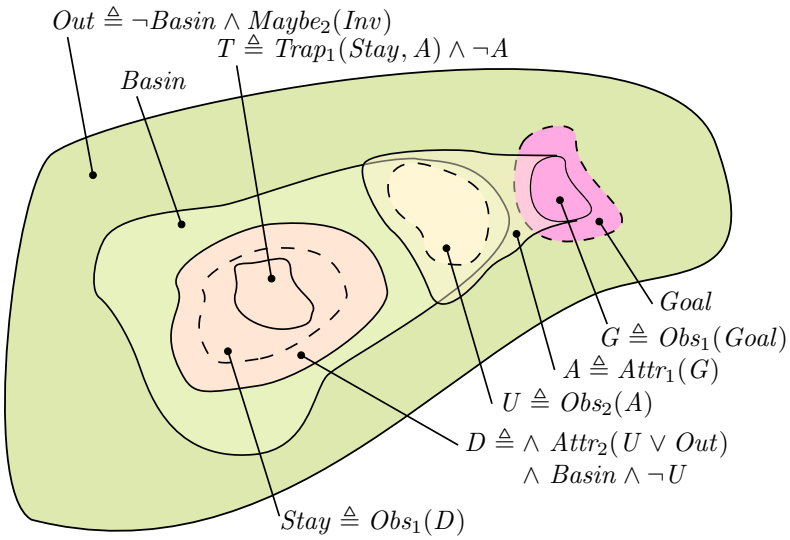


Figure 6.7: Accounting for observability when computing assumptions.

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 1 is the building block for computing more complex dependencies of objectives. For a single recurrence goal of component 1, multiple traps may be needed to cover the desired set of states (for which we use the global invariant Inv). If the procedure MakePinfoAssumption computes A, T, D, then by iterating this procedure until a least fixpoint is reached, we can find several traps, such that the corresponding persistence objectives suffice in order to eventually reach the $Goal$. This use is illustrated by the pseudocode

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

The procedure MakePinfoAssumption is defined in Section 6.6. This computation is in analogy to the least fixpoint computed for one goal in a GR(1) game [23].

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

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

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

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

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

$\vee\ (\mathit{free} = 0)$

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 $\mathit{spot}\_1, \mathit{spot}\_2, \mathit{spot}\_3$). The above expressions were computed from BDDs by using the approach described in Chapter 7, and the conjunct $\mathit{req} = 1$ was factored out of the disjuncts for brevity of the presentation. □

6.5 Multiple recurrence goals

The results of Section 6.4 are about one recurrence goal. By repeating the computation for different recurrence goals, for example $\Box\Diamond R_1$ and $\Box\Diamond R_2$ for component 1, suitable realizable properties can be found, for example $\Diamond\Box P_1 \vee \Box\Diamond R_1$ and $\Diamond\Box P_2 \vee \Box\Diamond R_2$. However, conjoining these two properties would not yield a GR(1) property. Instead, a GR(1) property can be formed by a suitable combination described below, provided that $\Box\Diamond\neg P_1 \wedge \Box\Diamond\neg P_2$ are implemented by components that can realize them irrespective of how component 1 behaves (unconditionally).

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

PROPOSITION Relaxation

ASSUME TEMPORAL P, TEMPORAL Q

PROVE $(\mathit{IsRealizable}(P) \wedge (P \Rightarrow Q)) \Rightarrow \mathit{IsRealizable}(Q)$

For what we are interested in, let

$$\begin{aligned} L &\triangleq \ \wedge\ \Diamond\Box P_1 \vee \Box\Diamond R_1 \\ &\qquad \wedge\ \Diamond\Box P_2 \vee \Box\Diamond R_2 \\ Q &\triangleq \ \vee\ \Diamond\Box P_1 \vee \Diamond\Box P_2 \\ &\qquad \vee\ \Box\Diamond R_1 \wedge \Box\Diamond R_2 \end{aligned}$$

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

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

and σ arises when using a component that implements Q, then it follows that $\sigma \models \Box\Diamond R_1 \wedge \Box\Diamond R_2$. 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.

$$\begin{array}{lll}
\textsc{theorem}\ \text{Let}\ Q & \triangleq & \lor\ \Diamond\Box\, T_1 \lor \Diamond\Box\, T_2 \\
 & & \lor\ \Box\Diamond R_1 \land \Box\Diamond R_2 \\
\quad \textsc{assume} & \mathit{IsRealizable}_1(Q) \land \land\ \neg D_1 \Rightarrow \neg T_1 & \\
 & \qquad\qquad\qquad\qquad \land\ \neg D_2 \Rightarrow \neg T_2 & \\
\quad \textsc{prove} & \forall f\ :\ \lor\ \neg\ \land\ \mathit{IsARealization}_1(f, Q) & \\
 & \qquad\qquad\qquad\qquad \land\ \mathit{Realization}_1(f, Q) & \\
 & \qquad\qquad\qquad\qquad \land\ \Box\Diamond\neg D_1 \land \Box\Diamond\neg D_2 & \\
 & \qquad\qquad\qquad \lor\ \Box\Diamond R_1 \land \Box\Diamond R_2 &
\end{array}$$

The operator *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 the e being used for component 1. The discussion above generalizes to more than two recurrence properties in an analogous way.

6.6 Detecting solutions in the presence of parametrization

The implementation of the computation described in Section 6.4 is shown in Algorithm 6.8. The controllable step operator, fixpoint and other computations are parametrized with respect to the communication between the components, as described in Chapter 5. 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 Section 6.7.2.

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 Section 6.2, which can fail as demonstrated by Fig. 6.4. 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. 6.9. 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. 6.10. 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 2 (Termination)**.** ASSUME *: A finite number of states satisfies the global invariant Inv.* PROVE *: Algorithm 6.8 terminates in a finite number of iterations.*

A structured proof style is used [120, 121].

$\langle 1 \rangle$ $k \triangleq$ CHOOSE $n \in Nat :$ TRUE

$\langle 1 \rangle$ SUFFICES $\vee\ Terminates(iter = k)$
$\vee\ EnlargesStrictly(Basin,\ iter = k)$

BY ASSUMPTION Finitely many relevant states satisfy Basin.

$\langle 1 \rangle 1$.CASE $At(L1,\ iter = k) :\ \models Escape \equiv$ FALSE

$\langle 2 \rangle 1.\ Terminates(iter = k)$

BY $\langle 1 \rangle 1,\ WhileGuard$

$\langle 2 \rangle$ QED

BY $\langle 2 \rangle 1$

$\langle 1 \rangle 2$.CASE $At(L1,\ iter = k + 1) :\ \neg \models Escape \equiv$ FALSE)

$\langle 2 \rangle 1.\ At(L2,\ iter = k) :\ \models Out' \Rightarrow \neg Basin$

$\langle 2 \rangle 2.\ At(L3,\ iter = k) :\ \models Escape' \Rightarrow Out$

$\langle 2 \rangle 3.\ At(L3,\ iter = k) :\ \models Escape' \Rightarrow \neg Basin$

BY $\langle 2 \rangle 1,\ \langle 2 \rangle 2$

$\langle 2 \rangle 4.\ At(L4,\ iter = k) :$
$\models (Escape \wedge Basin) \equiv$ FALSE

BY $\langle 2 \rangle 3$

$\langle 2 \rangle 5.\ At(L4,\ iter = k) :\ \models Escape \Rightarrow Basin'$

$\langle 2 \rangle 6.\ At(L4,\ iter = k) :$
$\neg \models (Basin' \wedge \neg Basin) \equiv$ FALSE

Algo. 6.8: Algorithm for constructing contracts of recurrence-persistence pairs. The presentation borrows elements from PlusCal [119].

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 $Escape := \wedge\ Maybe(Escape, Team)$
      $\wedge\ Out \wedge \neg Converged$
  L4 $Basin := Basin \vee Escape$
    $\eta_{player}, \eta_{team} :=$ MakePair(
      $A, Basin, Player, Team)$
    $Converged := \vee\ Converged$
      $\vee\ NonEmptySlices(\eta_{player})$
  return $A, \eta_{player}, \eta_{team}$

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

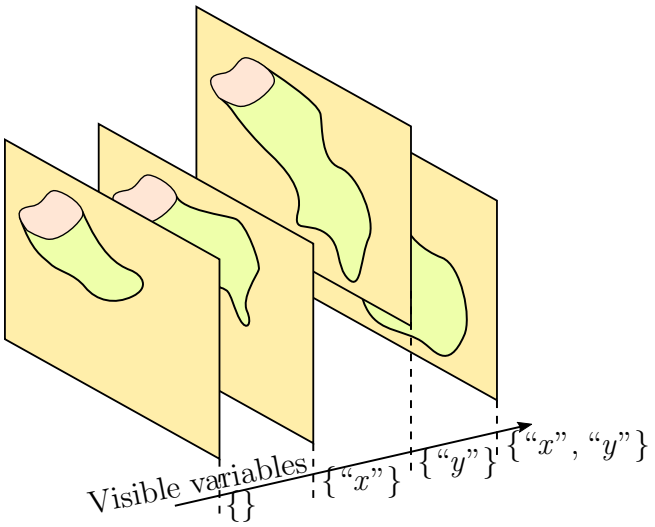


Figure 6.9: Slices of the state space that correspond to different assignments of values to the parameters.

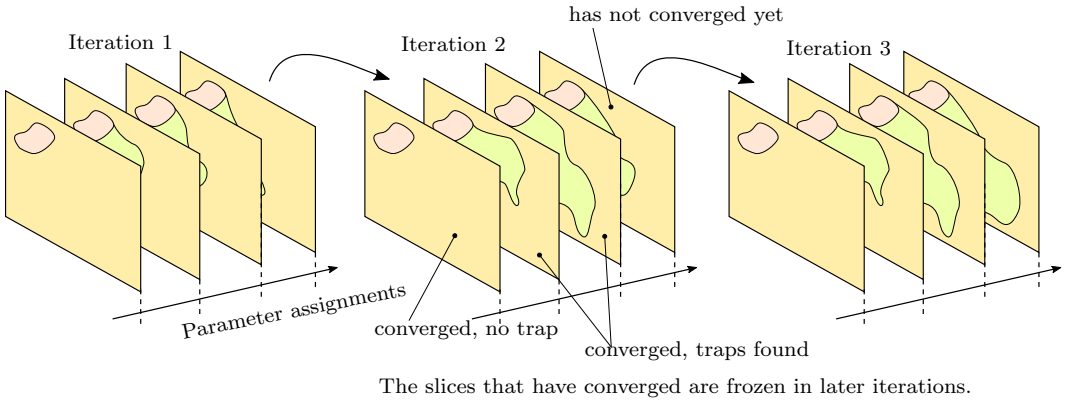


Figure 6.10: In iterations of non-monotonic operators that depend on parameters, when a solution is found for some parameter values (a slice), then no further iterations should occur for those values.

BY $\langle 1\rangle 2$, $\langle 2\rangle 4$, $\langle 2\rangle 5$

$\langle 2\rangle 7$. $At(L4,\ iter = k) :\ \models Basin \Rightarrow Basin'$

$\langle 2\rangle 8$. $EnlargesStrictly(Basin,\ iter = k)$

BY $\langle 2\rangle 7$

$\langle 2\rangle$ QED

BY $\langle 2\rangle 8$

$\langle 1\rangle$ QED

BY $\langle 1\rangle 1$, $\langle 1\rangle 2$

6.6.1 Characterizing the parametrization

Parameters are TLA$^+$ constants, also known as *rigid variables* [117]. 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 a 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 [142, 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 [114, 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).

6.7 Other considerations

6.7.1 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 Section 6.4 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.

6.7.2 Systems with more than two components

By applying Theorem 1 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 Chapter 8.

When the procedure MakePair of Algorithm 6.8 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.

6.7.3 Switching interconnection architecture

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

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

The information available to each component is a prerequisite for realizability of its objectives. In the case of more than one interconnections, the goals of components are conditioned on the corresponding interconnection mode. In other words, persistence goals are conjoined to the interconnection mode they correspond, i.e., $\Diamond\Box(T \wedge (cnct = k))$, and recurrence goals are required only provided the corresponding interconnection mode is active, i.e., $\Box\Diamond((cnct =$

$k) \Rightarrow \neg D)$. This can be regarded as a component assuming that if it provides enough information to its environment, then it can in return request reactions that become feasible for the environment when that information is available.

Chapter 7

GENERATING MINIMAL SPECIFICATIONS

7.1 Minimal disjunctive normal form

We use binary decision diagrams [30, 31] for the symbolic computations described in previous sections. BDDs are typically used in symbolic model checking for verifying that a system has certain properties [39, 92], in synthesis of controllers [84, 160] (e.g., as circuits), and in electronic design automation [188, 75]. 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. 7.1. 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

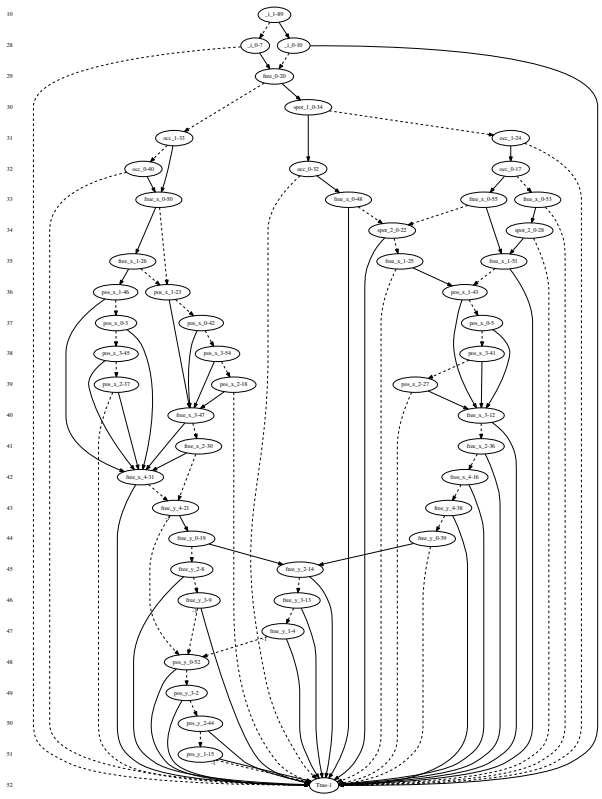


Figure 7.1: The binary decision diagram from which the formula of the global invariant was generated for the charging station example in Section 5.2. The variable names shown are the "bits" that are used to represent the integer-valued variables. A BDD isn't very suitable to help a human understand what Boolean expression it represents.

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 [132, 169, 27, 44]. Logic minimization is useful for reducing the number of physical elements used to implement a circuit, thus the circuit's physical area. The problem of finding a minimal DNF for a given Boolean function can be formulated as a minimal set covering problem, and is NP-hard. Algorithms for logic minimization are typically based on a branch-and-bound search.

We implemented an exact minimal covering algorithm [44] 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 [47, 46, 45, 43]. The original algorithm was formulated for the general case of a finite (complete) lattice,

and symbolically implemented for the case of the Boolean lattice [44, 140]. 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 (hyperrectangles aligned to axes). The implementation is available as part of the Python package `omega` [68]. Briefly, the problem of finding a minimal DNF formula of the form we described can be expressed as follows, where f is the Boolean function that is represented as a BDD and a formula is to be found. The *Domain* in our approach is a Cartesian product of integer variable ranges.

EXTENDS *FiniteSets*, *Integers*

CONSTANTS *Variables*, *Domain*, *CareSet*

$Assignments \triangleq [Variables \to Int]$

ASSUME

$\land (Domain \subseteq Assignments) \land (CareSet \subseteq Domain)$

$\land IsFiniteSet(Domain) \land IsFiniteSet(Variables)$

$\land (CareSet \neq \{\}) \land f \in [Domain \to \text{BOOLEAN}\ ]$

$EndPoint(k) \triangleq [1\ ..\ k \to Domain]$

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

$(a[var] \leq x[var]) \land (x[var] \leq b[var])$

$IsInRegion(x,\ p,\ q) \triangleq \exists\, i \in \text{DOMAIN}\ p :$

$IsInOrthotope(x,\ p[i],\ q[i])$

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

$f[x] \equiv IsInRegion(x,\ p,\ q)$

p, q define a cover that contains k orthotopes

$IsMinDNF(k,\ p,\ q,\ f) \triangleq$

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

$\land SameOver(f,\ p,\ q,\ CareSet)$

$\land \forall\, r \in Nat :\ \forall\, u,\ v \in EndPoint(r) :$

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

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

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

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

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

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

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

Besides reading the final result of a symbolic computation, we have found the method of decompiling BDDs as minimal DNF formulas over integer-valued variables an indispensable aid during the *development* of symbolic algorithms. Symbolic operations are implicit: the developer cannot inspect the values of variables as readily as for enumerative algorithms. It is highly unlikely that any symbolic program works on first writing. Bugs will usually be present, and some debugging needed. Being able to print small expressions for the BDD values of variables in symbolic code has helped us considerably during development efforts. Another area of using the algorithm is for inspecting controllers synthesized from temporal logic specifications.

Example 7. We show the usefulness of decompiling BDDs by revisiting the charging station example from Section 5.2. Fig. 7.1 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 [168], 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

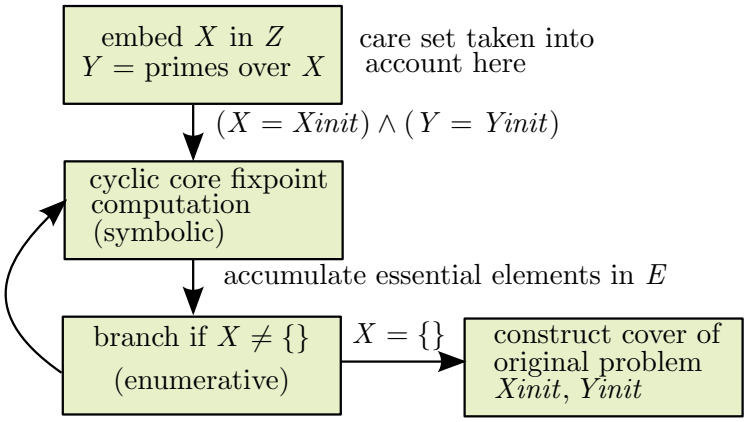


Figure 7.2: Overall minimal covering algorithm [44].

the minimal covering algorithm described above, we obtain the minimal DNF formula

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

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. □

7.2 Set covering over a lattice

We use an algorithm originally proposed for two-level logic minimization [44], which has three parts:

1. Computing essential and ambiguous elements (cyclic core);

2. branching by picking an ambiguous element as candidate; and

3. pruning subtrees of the search tree that branching creates.

Pruning is performed using lower and upper bounds on the minimal solution.

In Section 7.2.1 we describe how bounds on the minimal solution are computed for initialization and pruning of branches during the branch and bound search. In Section 7.2.2 we describe in more detail the computation of essential elements and elements in the *cyclic core* (a set of covering elements Y is a cyclic core when removing any single $y \in Y$ yields a feasible covering problem, and for no $y_1, y_2 \in Y$ does y_1 cover more than y_2). In Section 7.2.3 we revisit the computation of all minimal covers to the given minimal cover problem.

A TLA$^+$ specification for the cyclic core computation can be found in the appendices, together with proofs of its correctness. The main part of the proofs have been checked using the proof assistant TLAPS [35, 36, 48, 136] A note in the postamble of a module signifies whether TLAPS has checked it, and the version of the proof assistant used (namely, the modules *CyclicCore*, *StrongReduction*, *Lattices*, *MinCover*, *Optimization*, and *FiniteSetFacts*). The theorems and proofs are organized into modules of general interest, and modules specific to the cyclic core computation. This criterion for organization has been used also in the specification of Naiad [166, 9]. For basic facts about finite sets, functions, sequences, and well founded induction, we use the library of modules distributed with TLAPS.

The implementation of the algorithm is written in Python within the package `omega` [63] using the binary decision diagram package `dd` [62]. The specification of the cyclic core computation is shown below, together with theorems about its properties.

MODULE *CyclicCoreExcerpt*

EXTENDS

FiniteSetFacts, *Integers*, *Lattices*,
MinCover, *Optimization*, *TLAPS*

CONSTANTS

Leq, *Xinit*, *Yinit*

VARIABLES

X, Current set to be covered.

Y, Set of elements available for covering X.

E, Accumulates essential elements.

$Xold$, $Yold$, History variables used to detect fixpoint.

i Program counter.

$Z \triangleq Support(Leq)$

ASSUMPTION $CostIsCard \triangleq$

$Cost = [cover \in \text{SUBSET } Z \mapsto Cardinality(cover)]$

Definitions for convenience.

$RowRed(u, v) \triangleq MaxCeilings(u, v, Leq)$

$ColRed(u, v) \triangleq MaxFloors(v, u, Leq)$

$InitIsFeasible \triangleq \exists C : IsACoverFrom(C, Xinit, Yinit, Leq)$

ASSUMPTION $ProblemInput \triangleq$

$\land IsACompleteLattice(Leq)$

$\land IsFiniteSet(Z)$

$\land Xinit \subseteq Z$

$\land Yinit \subseteq Z$

$\land InitIsFeasible$

Specification of cyclic core computation.

$TypeInv \triangleq$

$\land X \in \text{SUBSET } Z$

$\land Y \in \text{SUBSET } Z$

$\land E \in \text{SUBSET } Z$

$\land Xold \in \text{SUBSET } Z$

$\land Yold \in \text{SUBSET } Z$

$\land i \in 1 \,..\, 3$

$Init \triangleq$

$\land X = Xinit$

$\land Y = Yinit$

$\land E = \{\}$

$\land Xold = \{\}$

$\land Yold = \{\}$

$\land\ i = 1$

$ReduceColumns \triangleq$
$\land\ (i = 1) \land (i' = 2)$
$\land\ Y' = ColRed(X,\ Y)$
$\land\ Xold' = X$
$\land\ Yold' = Y$
$\land$ UNCHANGED $\langle X,\ E\rangle$

$ReduceRows \triangleq$
$\land\ (i = 2) \land (i' = 3)$
$\land\ X' = RowRed(X,\ Y)$
$\land$ UNCHANGED $\langle Y,\ E,\ Xold,\ Yold\rangle$

$RemoveEssential \triangleq$
$\land\ (i = 3) \land (i' = 1)$
$\land$ LET
$Ess \triangleq X \cap Y$ Essential elements.
IN
$\land\ X' = X \setminus Ess$
$\land\ Y' = Y \setminus Ess$
$\land\ E' = E \cup Ess$
$\land$ UNCHANGED $\langle Xold,\ Yold\rangle$

$Next \triangleq$
$\lor\ ReduceColumns$
$\lor\ ReduceRows$
$\lor\ RemoveEssential$

$vars \triangleq \langle X,\ Y,\ E,\ Xold,\ Yold,\ i\rangle$
$Spec \triangleq Init \land \Box[Next]_{vars} \land \mathrm{WF}_{vars}(Next)$

Invariants

$IsFeasible \triangleq \exists\, C :\ IsAMinCover(C,\ X,\ Y,\ Leq)$
$HatIsMinCover \triangleq$
$\forall\, C,\ H :$
$\lor \neg \land\ IsAMinCover(C,\ X,\ Y,\ Leq)$
$\land\ IsAHat(H,\ C \cup E,\ Yinit,\ Leq)$

$$\lor \land \mathit{IsAMinCover}(H,\ \mathit{Xinit},\ \mathit{Yinit},\ \mathit{Leq})$$
$$\land\ \mathit{Cardinality}(H) = \mathit{Cardinality}(C) + \mathit{Cardinality}(E)$$

$\mathit{ReachesFixpoint} \triangleq \Diamond\Box[\text{FALSE}]_{\langle X,\ Y\rangle}$

THEOREM $\mathit{TypeOK} \triangleq \mathit{Spec} \Rightarrow \Box\mathit{TypeInv}$

Any minimal covers of X, Y yield (via *Hat*) a minimal cover of the initial problem *Xinit*, *Yinit*.

THEOREM $\mathit{RecoveringMinCover} \triangleq$
$\mathit{Spec} \Rightarrow \Box\mathit{HatIsMinCover}$

THEOREM $\mathit{RemainsFeasible} \triangleq$
$\mathit{Spec} \Rightarrow \Box\mathit{IsFeasible}$

THEOREM $\mathit{Termination} \triangleq$
$\mathit{Spec} \Rightarrow \mathit{ReachesFixpoint}$

The role of a care set For covering problems that arise in an application with existing structure, using a care set that represents this structure can simplify the cover. For example, in a specification where we already know the input formulas, the minimal DNF that covers a set of variable assignments can be smaller when the rest of the specification is taken into account as a care set.

The meaning of a care set is illustrated in Fig. 7.3. Within the lattice of rectangles, covering the set X requires 3 rectangles. If we take the set shown as a care set though, we can cover X using only 2 rectangles. Points outside $X \cap \mathit{CareSet}$ are ignored, even if covered. In other words, in order to describe X without any care set as "background" information, we need more linear inequalities than needed if we know that some points can be ignored, e.g., because they never occur in our application.

If a care set is defined, it is taken into account during initialization of the algorithm, for computing the set Y of prime implicants using X and the care set. The set to be covered should be contained in the care set, and the care set contained within the type hints that guide the selection of bitvector widths in the symbolic encoding, as illustrated by Fig. 7.4.

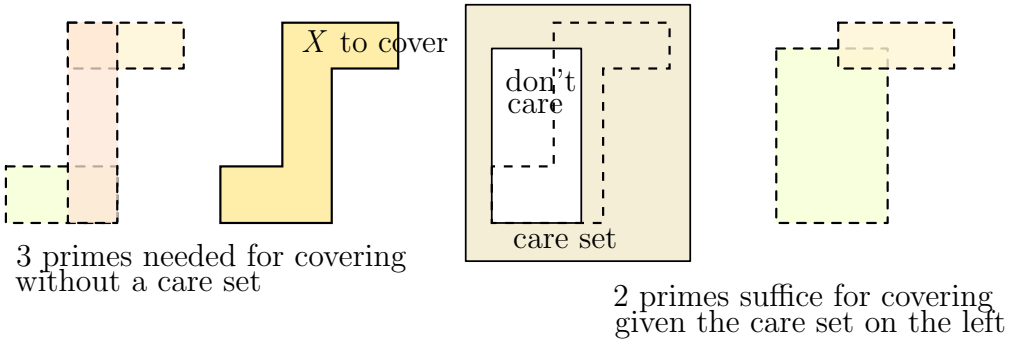


Figure 7.3: Using a care set that defines "don't care elements" allows finding simpler covers.

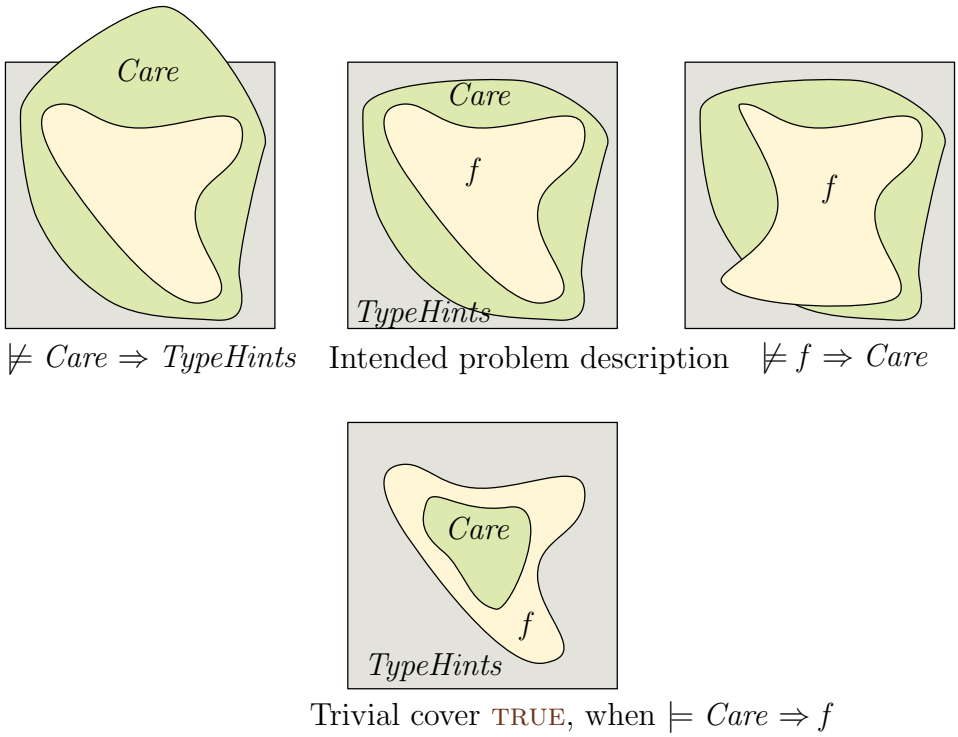


Figure 7.4: Different cases of problems. Except for the middle top case, the others raise warnings or errors in the implementation.

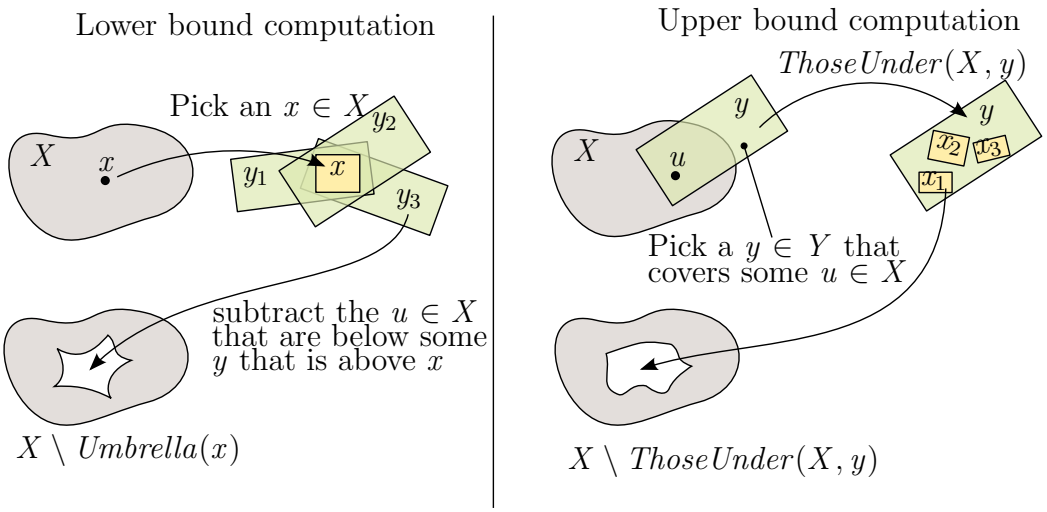


Figure 7.5: Computation of a lower bound iterates through the operations shown on the left. After all $u \in Umbrella(x)$ are removed from X, any v picked later will not be covered by any of the y that covered x. Computation of an upper bound iterates through the operations shown on the right. It is unnecessary to remove y from Y when computing an upper bound, because removing all the x below from X implies that y will not be picked again in any later iteration.

7.2.1 Upper and lower bounds for initialization and pruning

The branch and bound computation is well known, so we do not describe it. The computation of upper and lower bounds, though, is worth sketching. The bounds are needed for pruning branches during the branching search [44].

An upper bound is computed once upon initialization, by finding *some* cover, as illustrated in Fig. 7.5. This computation iteratively picks an element $y \in Y$ that cover at least one $u \in X$, adds it to the cover being constructed, and removes from X all those $v \in X$ covered by y. The iteration terminates when X becomes empty.

Lower bounds are computed throughout the branching search, by picking an $x \in X$, and removing from X the "umbrella" of x, i.e., all $u \in X$ that are covered by some $y \in Y$ that also covers this x. Again, the iteration terminates when X has been emptied. Fig. 7.6 shows what the operator *Umbrella* means. In both the upper and lower bound computations, the corresponding y elements are removed from Y in each iteration, but that operation is secondary to the algorithm's objective.

$$\begin{aligned} \mathit{Umbrella}(x_1, X, Y) &= \textsc{union}\ \{\mathit{ThoseUnder}(Xy) : y \in \mathit{ThoseOver}(Y, x_1)\} \\ &= \textsc{union}\ \{\mathit{ThoseUnder}(X, y_1), \mathit{ThoseUnder}(X, y_2)\} \\ &= \textsc{union}\ \{\{x_1, x_3\}, \{x_1, x_2\}\} \\ &= \{x_1, x_2, x_3\} \end{aligned}$$

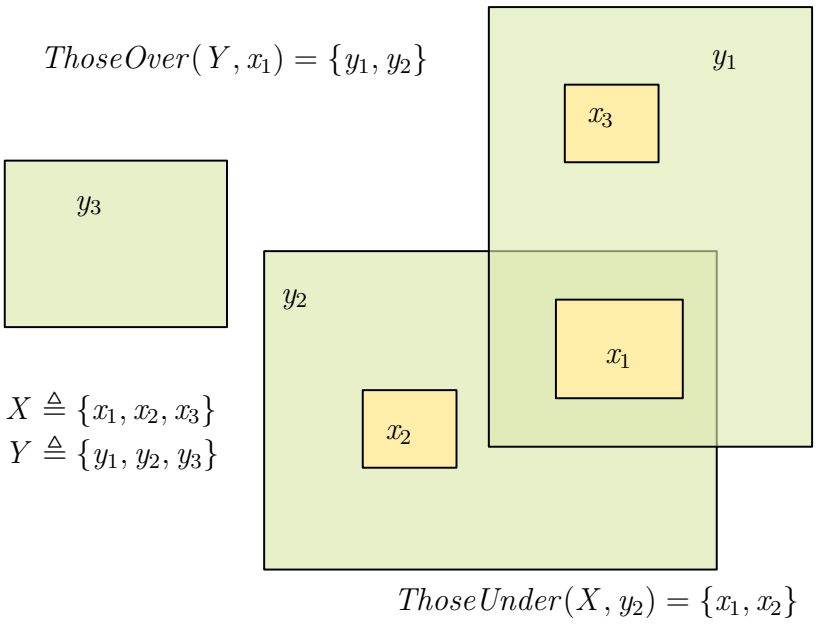


Figure 7.6: Illustration of *Umbrella*. The umbrella of x_1 contains those $x \in X$ that are covered by some $y \in Y$ that covers x_1.

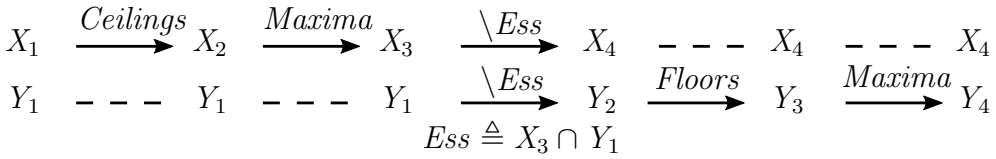


Figure 7.7: Steps within a single iteration while computing the cyclic core.

7.2.2 Cyclic core computation

The cyclic core computation iterates through three steps shown expanded in Fig. 7.7:

1. Reduction of the set of covering elements:

$$Y' = \mathit{Maxima}(\mathit{Floors}(Y, X, \mathit{Leq}), \mathit{Leq})$$

2. Reduction of the set to be covered:

$$X' = \mathit{Maxima}(\mathit{Ceilings}(X, Y, \mathit{Leq}), \mathit{Leq})$$

3. Removal and accumulation of essential elements

$$Ess = X \cap Y,\ X' = X \setminus Ess,\ Y' = Y \setminus Ess,\ E' = E \cup Ess.$$

Taking the floor of elements in Y maps (minimal) covers from Y to (minimal) covers from $Floors(Y)$, because by definition $Floor(y)$ covers exactly those elements covered by y (Fig. 7.10). Since y and $Floor(y)$ are both upper bounds of those x under y, but $Floor(y)$ is the supremum of that set, it follows that $Floor(y) \leq y$. Fig. 7.8 illustrates this relation.

$Maxima(Y)$ maps each (minimal) cover from Y to a (minimal) cover from $Maxima(Y) \subseteq Y$, which is also a (minimal) cover from Y (thus it is a mapping from SUBSET Y to SUBSET Y). This relation is illustrated in Fig. 7.8.

The operators *Ceilings* and *Maxima* on X leave the set of minimal covers unchanged. In the case of $Maxima(X)$, this is due to transitivity of the partial order (Fig. 7.8), and in the case of *Ceilings*, it is because by definition $Ceiling(x)$ is smaller than those y above x, so each $Ceiling(x)$ is covered by exactly the same elements that x was covered (enlargement of x and transitivity of $\leq$, Fig. 7.9 and Fig. 7.8). This iteration yields a fixpoint. If X is nonempty in this fixpoint, then the resulting covering problem defined by X, Y is called as the cyclic core, and requires branching in order to be solved. An example problem with a nonempty cyclic core is shown in Fig. 7.11.

7.2.3 Constructing all minimal covers

The original algorithm [44, Thm. 3] can generate some minimal covers of the given minimal covering problem, but not necessarily all. In particular, the step that constructs minimal covers of the input problem from minimal covers of the cyclic core (in each branching iteration) needs to be modified to account for the maximization step $Y = Maxima(Floors(Y, X, Leq), Leq)$.

Enumerating all minimal covers without enumerating covers other than minimal ones is known as *strong reduction* [44, § 2.4]. Even though the cyclic core computation and branch and bound in the original algorithm do yield the results necessary for constructing the set of all minimal covers to the input problem, the reconstruction step needs to be modified as described below.

Without this modification, some minimal covers can be lost, as demonstrated by the covering problem shown in Fig. 7.13. The steps of the cyclic core

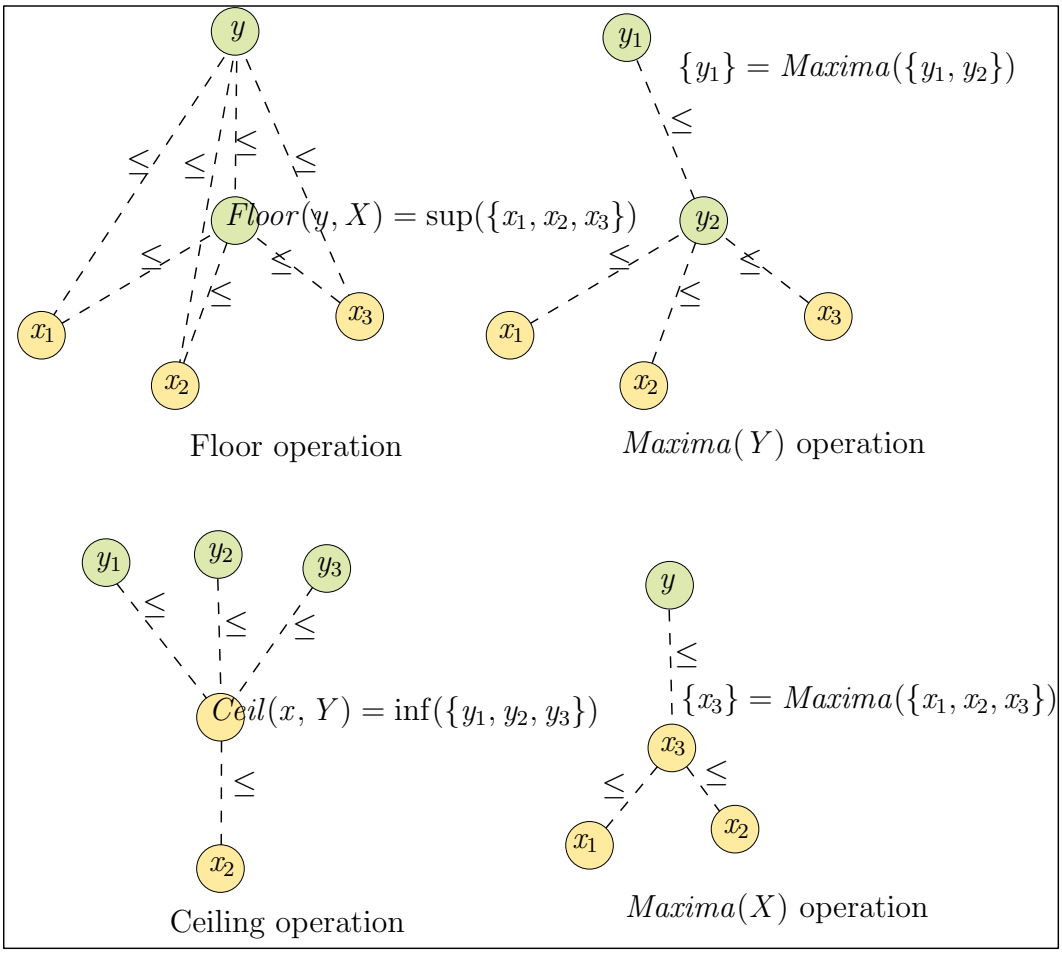


Figure 7.8: The four operations applied when computing the cyclic core: the floor of y is the supremum of those elements x that are below y, and the ceil of x is the infimum of those elements y that are above x. Maximization has the usual meaning. The algorithm's properties rely on the relations shown, which arise from the lattice.

iteration until reaching a fixpoint are shown in Fig. 7.14 (in this example, the cyclic core is empty, so no branching is needed). What happens in this example is that the maximization step $Ym = Maxima(Yf, Leq)$ (with $Yf = Floors(Y, X, Leq)$) yields a set Ym that does not have all of Y above it. In the end, this leads to the minimal cover $\{3, 4\}$. The only element from $Yinit$ above 3 is 3, and above 4 is 4. So the reconstruction of [44, Thm. 3] yields a single minimal cover, $\{3, 4\}$.

However, $\{2, 4\}$ and $\{3, 5\}$ are minimal covers too. The algorithm proposed below does recover also these two minimal covers. It does so by finding the elements below 3 that do cover all x that only 3 covers (i.e., 4 does not cover

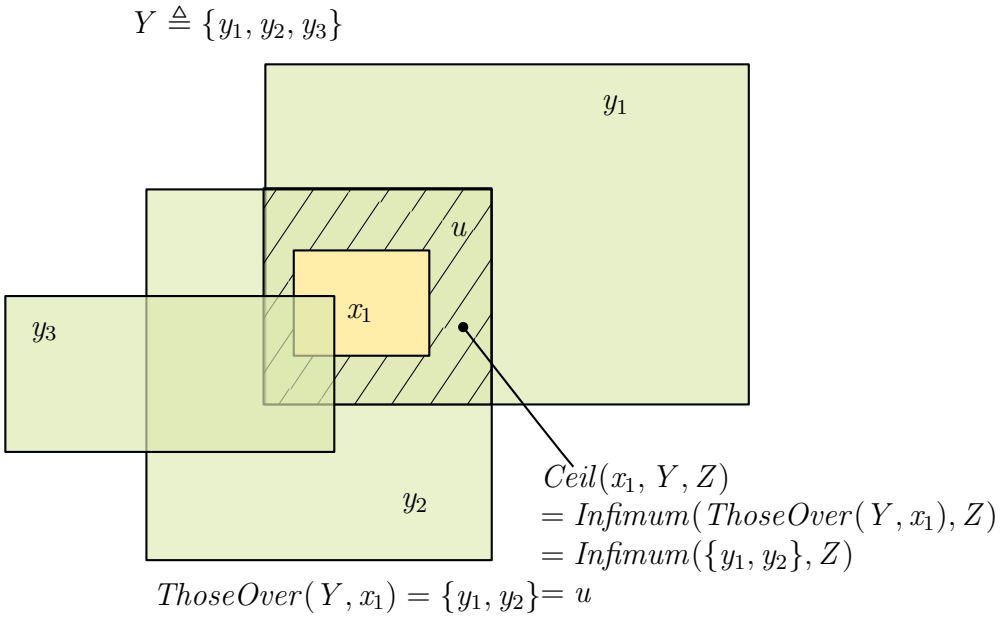


Figure 7.9: Illustration of *Ceil*. The lattice *Leq* is the subset relation between rectangles, with $(Z \times Z) =$ DOMAIN *Leq*.

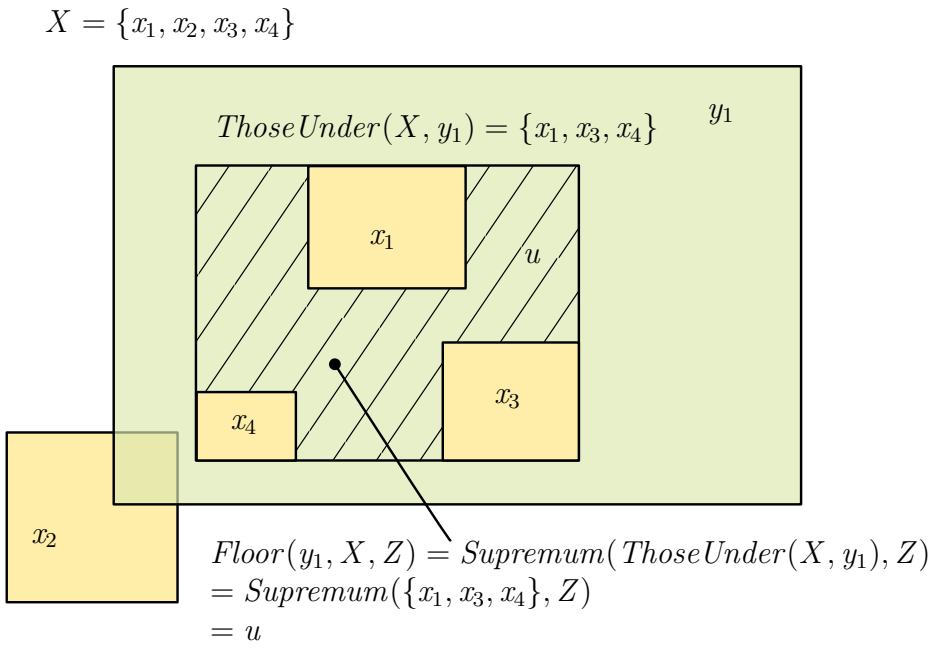


Figure 7.10: Illustration of *Floor*.

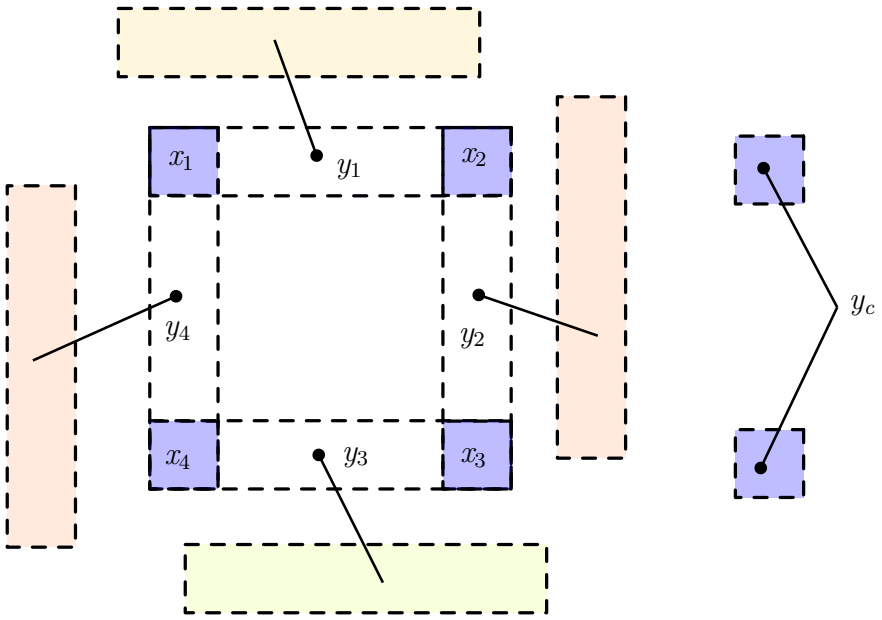


Figure 7.11: A covering problem that does yield a nonempty cyclic core, $Xinit = \{x_1, x_2, x_3, x_4\}$, $Yinit = \{y_1, y_2, y_3, y_4\}$. The cyclic core is $X = Xinit$ and $Y = \{y_{c1}, y_{c2}, y_{c3}, y_{c4}\}$. Each y_{ck} is the supremum of those x below y_k. For example, y_{c2} is shown on the right (translated from where it actually is, in order to avoid visual overlap).

those). This set is $\{6\}$ (7 is covered by 4, so not only by 3). This step inverts $Maxima(Yf, Leq)$. The next step is as in the original algorithm, inverting $Floors(Y, X, Leq)$. Namely, the elements above 6 are 2 and 3, which yield the original covers $\{2, 4\}$ and $\{3, 4\}$. Symmetrically, replacing 4 by 8 leads to the original covers $\{3, 4\}$ and $\{3, 5\}$.

The proposed algorithm for constructing the set of minimal covers of X, Y from the set of minimal covers of $X, Maxima(Y, Leq)$ is shown below, together with a specification, and the theorem *StrongReductionSafety* about its correctness.[1] Given a minimal cover $C_m \in$ SUBSET $Maxima(Y, Leq)$, the algorithm iterates though the elements $y_m \in C_m$, and "shrinks" each y_m by enumerating all those replacements $y \in Y$ that are below y_m, and cover all $x \in X$ that are covered *only* by y_m. For each y_m there are certainly some elements in X covered only by y_m, because otherwise $C_m \setminus \{y_m\}$ would be a cover cheaper than C_m, but C_m is minimal. The set of such sufficient y below a given y_m can be computed symbolically, before enumerating those elements. This avoids enumerating covers that are not minimal. All covers

[1] A proof can be found in the module *StrongReduction*.

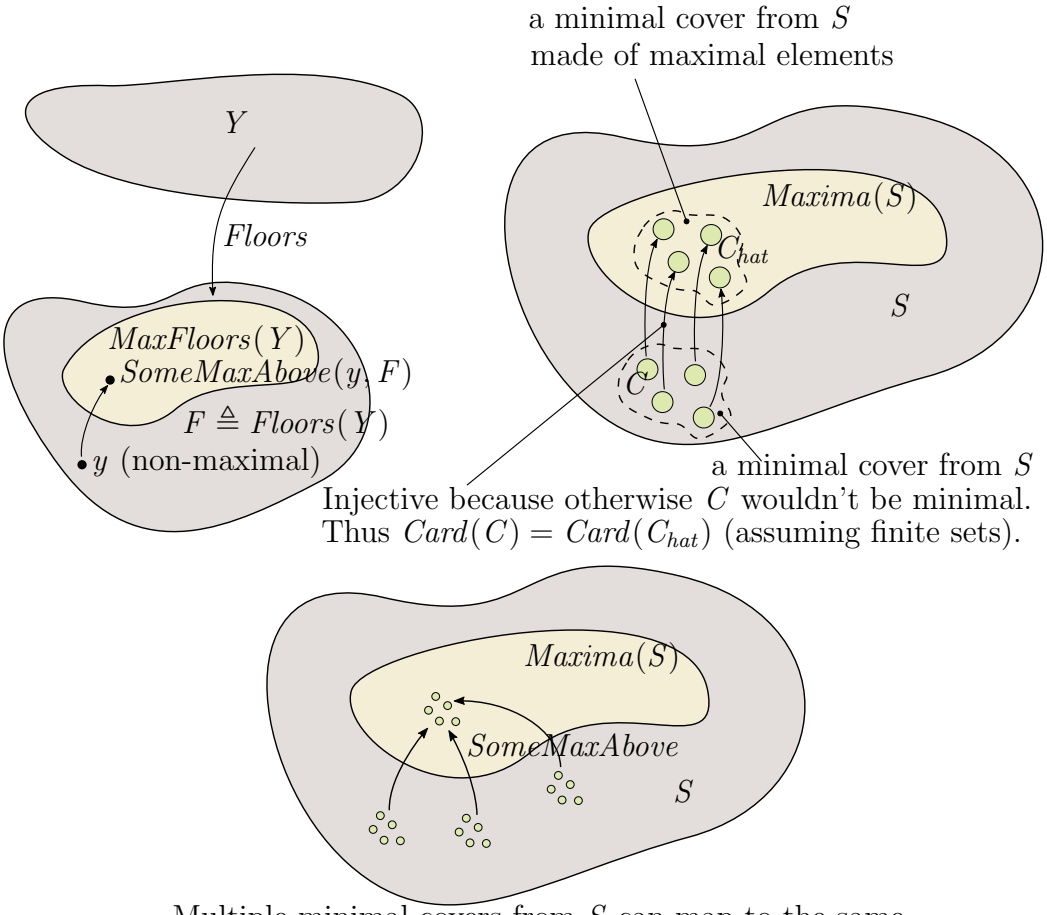


Figure 7.12: The maxima of any set S form a kernel that contains minimal covers, each of them representing an equivalence class of minimal covers from S. Thus, *SomeMaxAbove* induces a partition of minimal covers into equivalence classes, in effect "folding" the set of minimal covers into its intersection with $Maxima(S)$.

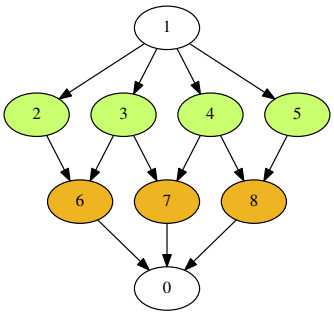


Figure 7.13: Example of a covering problem that demonstrates that the algorithm proposed in [44] does not ensure strong reduction (i.e., enumeration of all minimal covers, and only those). With the modification proposed in this document, strong reduction is ensured. The partial order is $2 \leq 1$ etc.

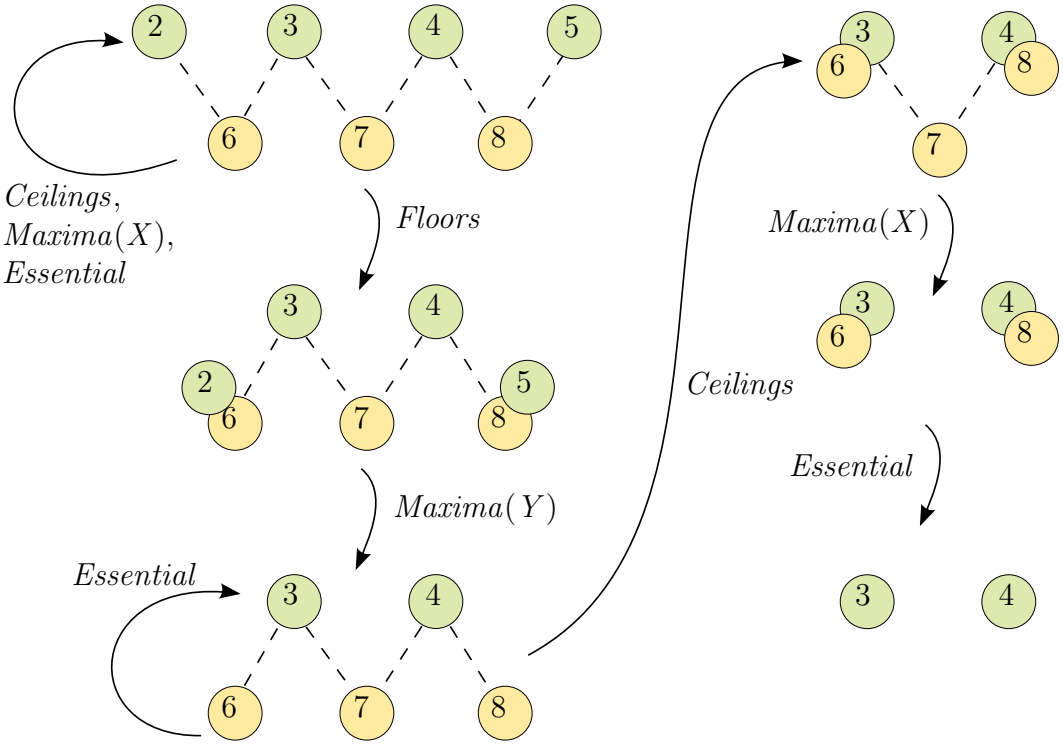


Figure 7.14: Steps of the cyclic core computation until reaching a fixpoint. This example shows that the minimal cover found does not yield all minimal covers to the input problem using the elements above elements from the cover [44]. Instead, first a "downward" step is needed.

that result by each such replacement are minimal covers from Y, so no redundant enumeration is performed. Moreover, if any cover $C \in \text{SUBSET } Y$ with $Cardinality(C) = Cardinality(C_m)$ is not enumerated, then some $y_m \in C_m$ cannot be replaced with a $y \in C$ that covers all the $x \in X$ covered by only y_m from among the $y \in PartialCover \cup Image(Lm, k..N)$, where $PartialCover$ is defined in the algorithm description.

Each minimal cover C from Y is elementwise below some minimal cover $C_m \in \text{SUBSET } Maxima(Y, Leq)$. Thus, applying the below algorithm to each minimal cover $C_m \in \text{SUBSET } Maxima(Y, Leq)$ yields in total the minimal covers of X consisting of elements from Y.

In contrast to the algorithm for enumerating some minimal covers of the input problem, which simply finds a "hat" of the E by elements from Y, enumerating all minimal covers requires backwards construction step-by-step, in reverse through each iteration of the cyclic core. In the presence of branching in the branch-and-bound part of the algorithm, all branches that *could* contain minimal covers need to be searched for finding all minimal covers to the original problem. An enumeration of this kind may not be efficient, nevertheless it is an algorithm for exhaustive reconstruction of minimal covers.

MODULE *StrongReductionExcerpt*

EXTENDS

FiniteSetFacts, *FunctionTheorems*, *Lattices*,
Sequences, *SequenceTheorems*, *TLAPS*

CONSTANTS Leq, X, Y

$Z \triangleq Support(Leq)$

ASSUMPTION $CostIsCard \triangleq$
$Cost = [cover \in \text{SUBSET } Z \mapsto Cardinality(cover)]$

ASSUMPTION $ProblemInput \triangleq$
$\land\ IsACompleteLattice(Leq)$
$\land\ IsFiniteSet(Z)$
$\land\ X \subseteq Z$
$\land\ Y \subseteq Z$

$Only(ymax,\ C) \triangleq \{u \in X :\ \forall\, yother \in C \setminus \{ymax\} :$
$\quad \neg Leq[u,\ yother]\}$

$BelowAndSuff(ymax,\ C,\ V) \triangleq$
$\quad \{y \in V :$
$\qquad \wedge\ Leq[y,\ ymax]$
$\qquad \wedge \forall\, q \in Only(ymax,\ C) :\ Leq[q,\ y]\}$

Cm is a cover of X from $Maxima(Y, Leq)$
$AllCandidatesBelow(Cm,\ V) \triangleq$
$\quad \{S \in \text{SUBSET}\ V :$
$\qquad \wedge\ Cardinality(S) = Cardinality(Cm)$
unnecessary to consider smaller subsets (they cannot be covers), or larger subsets (they cannot be minimal)
$\qquad \wedge\ Refines(S,\ Cm,\ Leq)\}$

$Enumerate(S) \triangleq$
$\quad \text{LET}\ Dom \triangleq 1\ ..\ Cardinality(S)$
$\quad \text{IN}\quad \text{CHOOSE}\ f :\ f \in Bijection(Dom,\ S)$

$Image(f,\ S) \triangleq \{f[x] :\ x \in S\}$

Specification of the reconstruction of minimal covers from Y that are below a given cover Cm from $Maxima(Y, Leq)$.

CONSTANTS Cm
VARIABLES $stack,\ MinCoversBelow$

$Max \triangleq Maxima(Y,\ Leq)$
$Lm \triangleq Enumerate(Cm)$
$N \triangleq Cardinality(Cm)$ $N = Len(Lm)$
$Patch(r) \triangleq Image(Lm,\ r\ ..\ N)$

$TypeInv \triangleq \wedge\ stack \in Seq(\text{SUBSET}\ Y)$
$\qquad\qquad \wedge\ MinCoversBelow \subseteq \text{SUBSET}\ Y$

$Init \triangleq \wedge\ stack = \langle\{\}\rangle$
$\qquad\quad \wedge\ MinCoversBelow = \{\}$

Terminal case that adds a minimal cover to the set $MinCoversBelow$.

$Collect \triangleq$
LET
$end \triangleq Len(stack)$
$Partial \triangleq stack[end]$
$i \triangleq Cardinality(Partial)$
$front \triangleq SubSeq(stack, 1, end - 1)$
IN
$\land\ i = N$
$\land\ stack' = front$
$\land\ MinCoversBelow' = MinCoversBelow \cup \{Partial\}$

Branching that generates all minimal covers induced by replacing the next maximal element *ymax* with all those below it that suffice (*succ*).

$Expand \triangleq$
LET
$end \triangleq Len(stack)$
$Partial \triangleq stack[end]$
$i \triangleq Cardinality(Partial)$
$k \triangleq i + 1$
$front \triangleq SubSeq(stack, 1, end - 1)$

$ymax \triangleq Lm[k]$ element to replace
$Q \triangleq Partial \cup Patch(k)$
$succ \triangleq BelowAndSuff(ymax, Q, Y)$
$enum \triangleq Enumerate(succ)$
$more \triangleq [r \in 1 \,..\, Len(enum) \mapsto Partial \cup \{enum[r]\}]$
IN
$\land\ i < N$
$\land\ stack' = front \circ more$
$\land$ UNCHANGED $MinCoversBelow$

$Next \triangleq$
$\land\ stack \neq \langle\rangle$
$\land\ \lor\ Collect$
$\lor\ Expand$

$vars \triangleq \langle stack, MinCoversBelow \rangle$
$Spec \triangleq Init \land \Box[Next]_{vars} \land \mathrm{WF}_{vars}(Next)$

Invariants.

$LeqToBij(C) \triangleq \text{CHOOSE } g \in Bijection(1 \,..\, N,\ C) :$
$\quad \forall\, q \in 1 \,..\, N :\ Leq[g[q],\ Lm[q]]$

$IsPrefixCov(PartialCover,\ g) \triangleq$
$\quad \text{LET } i \triangleq Cardinality(PartialCover)$
$\quad \text{IN} \quad PartialCover = \{g[q] :\ q \in 1 \,..\, i\}$

$InvCompl(C) \triangleq$
$\quad \text{LET} \quad g \triangleq LeqToBij(C)$
$\quad \text{IN}$
$\qquad \vee \exists\, n \in \text{DOMAIN } stack :\ IsPrefixCov(stack[n],\ g)$
$\qquad \vee \neg IsAMinCover(C,\ X,\ Y,\ Leq)$
$\qquad \vee\ C \in MinCoversBelow$

$InvSound(C) \triangleq\ (C \in MinCoversBelow) \Rightarrow IsAMinCover(C,\ X,\ Y,\ Leq)$

$\text{THEOREM } StrongReductionSafety \triangleq$
$\quad \text{ASSUME}$
$\qquad \text{NEW } C, \quad IsAMinCover(Cm,\ X,\ Max,\ Leq)$
$\quad \text{PROVE}$
$\qquad \wedge\ Spec \Rightarrow \Box InvSound(C)$
$\qquad \wedge\ (C \in AllCandidatesBelow(Cm,\ Y))$
$\qquad\quad \Rightarrow (Spec \Rightarrow \Box InvCompl(C))$

```
def EnumerateMincoversBelow(Cm) :
  stack := ⟨{}⟩
  Lm := Enumerate(Cm)
  N := Cardinality(Cm)
  MinCoversBelow := {}
  while stack ≠ ⟨⟩ :
    end := Len(stack)
    PartialCover := stack[end]
    i := Cardinality(PartialCover)
    stack := SubSeq(stack, 1, end − 1)
    if i = N :
      MinCoversBelow := MinCoversBelow ∪ {PartialCover}
      continue
    k := i + 1
    succ := BelowAndSuff(Lm[k], PartialCover ∪ Image(Lm, k..N), Y)
    for z ∈ succ :
      NewCover := PartialCover ∪ {z}
      stack := stack ∘ ⟨NewCover⟩
  return MinCoversBelow
```

7.2.4 Definitions from the modules *Lattices*, *Optimization*, and *MinCover*

MODULE *OptimizationExcerpt*

EXTENDS *FiniteSetFacts*, *Integers*, *WellFoundedInduction*

$$IsAFunction(f) \triangleq f = [x \in \text{DOMAIN } f \mapsto f[x]]$$

$$Support(R) \triangleq \{p[1] : p \in \text{DOMAIN } R\} \cup \{p[2] : p \in \text{DOMAIN } R\}$$

$$IsReflexive(R) \triangleq \text{LET } S \triangleq Support(R) \text{ IN } \forall x \in S : R[x, x]$$

$$IsIrreflexive(R) \triangleq \text{LET } S \triangleq Support(R) \text{ IN } \forall x \in S : \neg R[x, x]$$

$$IsTransitive(R) \triangleq \text{LET } S \triangleq Support(R) \text{ IN } \forall x, y, z \in S : (R[x, y] \land R[y, z]) \Rightarrow R[x, z]$$

$$IsSymmetric(R) \triangleq \text{LET } S \triangleq Support(R)$$

$$\text{IN} \quad \forall\, x,\, y \in S :\ R[x,\, y] \Rightarrow R[y,\, x]$$

$$IsAntiSymmetric(R) \triangleq \text{LET } S \triangleq Support(R)$$
$$\text{IN} \quad \forall\, x,\, y \in S :\ (R[x,\, y] \land (x \neq y)) \Rightarrow \neg R[y,\, x]$$

S is a set of pairwise comparable elements (totality).

$$IsChain(S,\, Leq) \triangleq \forall\, x,\, y \in S :\ Leq[x,\, y] \lor Leq[y,\, x]$$

S is a set of pairwise incomparable elements.

$$IsAntiChain(S,\, Leq) \triangleq \forall\, x,\, y \in S :$$
$$(x \neq y) \Rightarrow (\neg Leq[x,\, y] \ \land\ \neg Leq[y,\, x])$$

Optimization

When the minimum exists, it is unique, similarly for the maximum.

$$IsMinimum(r,\, S,\, Leq) \triangleq \land\, r \in S$$
$$\land\, \forall\, u \in S \setminus \{r\} :\ Leq[r,\, u]$$

$$IsMaximum(r,\, S,\, Leq) \triangleq \land\, r \in S$$
$$\land\, \forall\, u \in S \setminus \{r\} :\ Leq[u,\, r]$$

This definition requires that *Leq* be reflexive, so it applies to partial orders.

$$IsMinimalRefl(r,\, S,\, Leq) \triangleq \land\, r \in S$$
$$\land\, \forall\, u \in S \setminus \{r\} :\ \neg Leq[u,\, r]$$

$$IsMaximalRefl(r,\, S,\, Leq) \triangleq \land\, r \in S$$
$$\land\, \forall\, u \in S \setminus \{r\} :\ \neg Leq[r,\, u]$$

A general definition, which applies even if *Leq* is not anti-symmetric, and so also to preorders.

$$IsMinimal(r,\, S,\, Leq) \triangleq \land\, r \in S$$
$$\land\, \forall\, u \in S :\ Leq[u,\, r] \Rightarrow Leq[r,\, u]$$

$$IsMaximal(r,\, S,\, Leq) \triangleq \land\, r \in S$$
$$\land\, \forall\, u \in S :\ Leq[r,\, u] \Rightarrow Leq[u,\, r]$$

If a minimum does exist, then it is unique, so clearly "minima" refers to minimal elements. In presence of the minimum, *Minima* is a singleton.

$$Minima(S,\, Leq) \triangleq \{x \in S :\ IsMinimal(x,\, S,\, Leq)\}$$
$$Maxima(S,\, Leq) \triangleq \{x \in S :\ IsMaximal(x,\, S,\, Leq)\}$$

$$IndicatorFuncToRel(f) \triangleq \{x \in \text{DOMAIN } f :\ f[x] = \text{TRUE}\}$$

$IrreflexiveFrom(Leq) \triangleq$

LET

$S \triangleq Support(Leq)$

IN

$[t \in S \times S \mapsto \text{IF } t[1] = t[2] \text{ THEN FALSE ELSE } Leq[t]]$

MODULE *MinCoverExcerpt*

EXTENDS *Integers*, *Optimization*

CONSTANTS *Cost*

Minimal set covering

$CostLeq[t \in (\text{DOMAIN } Cost) \times (\text{DOMAIN } Cost)] \triangleq$

LET

$r \triangleq t[1]$

$u \triangleq t[2]$

IN $Cost[r] \leq Cost[u]$

$CardinalityAsCost(Z) \triangleq Cost = [cover \in \text{SUBSET } Z \mapsto Cardinality(cover)]$

C and X suffice to define a cover, because the notion of covering involves elements from a cover and a target set to cover. Y is irrelevant.

$IsACover(C, X, IsUnder) \triangleq \forall x \in X : \exists y \in C : IsUnder[x, y]$

$IsACoverFrom(C, X, Y, IsUnder) \triangleq$

$\land C \in \text{SUBSET } Y$

$\land IsACover(C, X, IsUnder)$

$CoversOf(X, Y, IsUnder) \triangleq \{C \in \text{SUBSET } Y : IsACover(C, X, IsUnder)\}$

The set Y is irrelevant to the notion of a cover, but is necessary to define a notion of minimal element.

$IsAMinCover(C, X, Y, IsUnder) \triangleq$

LET

$Covers \triangleq CoversOf(X, Y, IsUnder)$

IN

$IsMinimal(C, Covers, CostLeq)$

$MinCost(X, Y, IsUnder) \triangleq$

LET

$Cov \triangleq CoversOf(X, Y, IsUnder)$

```
        min ≜ CHOOSE u ∈ Minima(Cov, CostLeq) : TRUE
    IN
        Cost[min]
```

$IsACover(C, X, Leq) \equiv Refines(X, C, Leq)$

$Refines(A,\ B,\ Leq) \triangleq \forall\, u \in A :\ \exists\, v \in B :\ Leq[u,\ v]$

MODULE *LatticesExcerpt*

EXTENDS *FiniteSetFacts*, *MinCover*, *Optimization*

$ThoseUnder(X,\ y,\ Leq) \triangleq \{x \in X :\ Leq[x,\ y]\}$

$ThoseOver(Y,\ x,\ Leq) \triangleq \{y \in Y :\ Leq[x,\ y]\}$

$Umbrella(x,\ X,\ Y,\ Leq) \triangleq$ UNION $\{$
$\quad ThoseUnder(X,\ y,\ Leq) :\ y \in ThoseOver(Y,\ x,\ Leq)\}$

$IsBelow(r,\ S,\ Leq) \triangleq \forall\, u \in S :\ Leq[r,\ u]$

$IsAbove(r,\ S,\ Leq) \triangleq \forall\, u \in S :\ Leq[u,\ r]$

$IsTightBound(r,\ S,\ Leq) \triangleq$
LET
$\quad Z \triangleq Support(Leq)$
IN
$\quad \land\, r \in Z$
$\quad \land\, IsAbove(r,\ S,\ Leq)$
$\quad \land\, \forall\, q \in Z :\ IsAbove(q,\ S,\ Leq) \Rightarrow Leq[r,\ q]$

$HasTightBound(S,\ Leq) \triangleq$
LET $Z \triangleq Support(Leq)$
IN $\exists\, r \in Z :\ IsTightBound(r,\ S,\ Leq)$

$TightBound(S,\ Leq) \triangleq$
LET $Z \triangleq Support(Leq)$
IN CHOOSE $r \in Z :\ IsTightBound(r,\ S,\ Leq)$

$UpsideDown(Leq) \triangleq$
LET $Z \triangleq Support(Leq)$
IN $[t \in Z \times Z \mapsto Leq[t[2],\ t[1]]]$

$HasSup(S,\ Leq) \triangleq HasTightBound(S,\ Leq)$

$$HasInf(S,\ Leq) \triangleq \text{LET } Geq \triangleq UpsideDown(Leq) \text{ IN } HasTightBound(S,\ Geq)$$

$$Supremum(S,\ Leq) \triangleq TightBound(S,\ Leq)$$

$$Infimum(S,\ Leq) \triangleq \text{LET } Geq \triangleq UpsideDown(Leq) \text{ IN } TightBound(S,\ Geq)$$

$$Floor(y,\ X,\ Leq) \triangleq Supremum(ThoseUnder(X,\ y,\ Leq),\ Leq)$$

$$Ceil(x,\ Y,\ Leq) \triangleq Infimum(ThoseOver(Y,\ x,\ Leq),\ Leq)$$

$$Floors(S,\ X,\ Leq) \triangleq \{Floor(y,\ X,\ Leq) :\ y \in S\}$$

$$Ceilings(S,\ Y,\ Leq) \triangleq \{Ceil(x,\ Y,\ Leq) :\ x \in S\}$$

$\max \tau_X$ or "column reduction" in [44].

$$MaxFloors(S,\ X,\ Leq) \triangleq Maxima(Floors(S,\ X,\ Leq),\ Leq)$$

$\max \tau_Y$ or "row reduction" in [44].

$$MaxCeilings(S,\ Y,\ Leq) \triangleq Maxima(Ceilings(S,\ Y,\ Leq),\ Leq)$$

$$IsAQuasiOrder(R) \triangleq \land IsReflexive(R) \land IsTransitive(R) \land IsAFunction(R) \land \exists\, S :\ S \times S = \text{DOMAIN } R$$

$$IsAPartialOrder(R) \triangleq \land IsReflexive(R) \land IsTransitive(R) \land IsAntiSymmetric(R) \land IsAFunction(R) \land \exists\, S :\ S \times S = \text{DOMAIN } R$$

$$IsALattice(R) \triangleq \land IsAPartialOrder(R) \land \text{LET } Z \triangleq Support(R) \text{ IN } \forall\, S \in \text{SUBSET } Z :\ \lor Cardinality(S) \neq 2 \lor HasInf(S,\ R) \land HasSup(S,\ R)$$

$$IsACompleteLattice(R) \triangleq \land IsAPartialOrder(R) \land \text{LET } Z \triangleq Support(R) \text{ IN } \forall\, S \in \text{SUBSET } Z :\ HasInf(S,\ R) \land HasSup(S,\ R)$$

$$SomeAbove(u,\ Y,\ Leq) \triangleq \text{CHOOSE } r \in Y :\ Leq[u,\ r]$$

$$SomeMaxAbove(u,\ Y,\ Leq) \triangleq \text{CHOOSE } m \in Maxima(Y,\ Leq) :\ Leq[u,\ m]$$

$Hat(S,\ Y,\ Leq) \triangleq \{SomeAbove(y,\ Y,\ Leq) :\ y \in S\}$

$IsAHat(H,\ C,\ Y,\ Leq) \triangleq$

$\quad \land\ H \in \text{SUBSET}\ Y$

$\quad \land\ Refines(C,\ H,\ Leq)$

$\quad \land\ Cardinality(H) \leq Cardinality(C)$

$MaxHat(S,\ Y,\ Leq) \triangleq \{SomeMaxAbove(y,\ Y,\ Leq) :\ y \in S\}$

$SomeUnfloor(u,\ X,\ Y,\ Leq) \triangleq \text{CHOOSE}\ y \in Y :\ u = Floor(y,\ X,\ Leq)$

$Unfloors(S,\ X,\ Y,\ Leq) \triangleq \{SomeUnfloor(y,\ X,\ Y,\ Leq) :\ y \in S\}$

$IsUnfloor(C,\ F,\ X,\ Leq) \triangleq \land\ F = Floors(C,\ X,\ Leq)$

$\qquad\qquad\qquad\qquad\qquad\quad \land\ Cardinality(C) \leq Cardinality(F)$

Chapter 8

EXAMPLE

The example we consider concerns the subsystems involved in controlling the landing gear of an aircraft [67, 25]. Three modules are involved, as shown in Fig. 8.1. The autopilot controls the altitude, flight speed, and mode of the aircraft. The gear module positions the landing gear, which can be extended, retracted, or in some transitory configuration. The third module operates the doors that seal the gear storage area. The variables take integer values, with appropriate units that can be ignored for our purpose here. We specify the following main properties collectively for these modules:

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

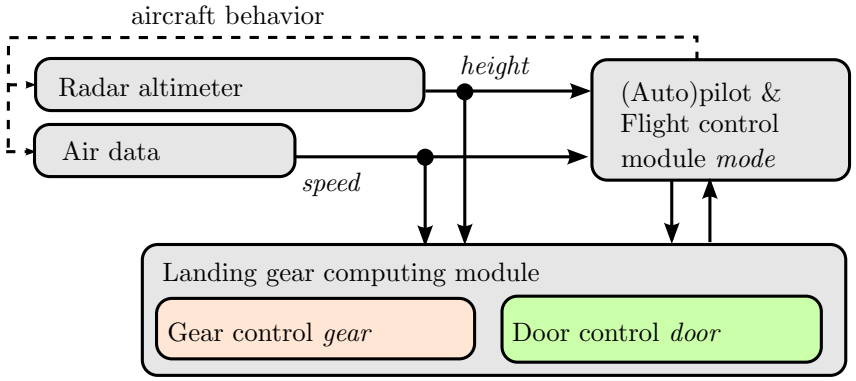


Figure 8.1: Landing gear avionics.

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

EXTENDS *Integers*

VARIABLES *mode*, *height*, *speed*, *door*, *gear*, *turn*

CONSTANTS *max_height*, *max_speed*, *door_down*,
gear_down, *threshold_height*, *threshold_speed*

mode 0, 2, 1 used below

$$Modes \triangleq \{\text{“landing”}, \text{“intermediate”}, \text{“cruise”}\}$$

$$Autopilot \triangleq \langle height, mode, speed \rangle$$

$$AutopilotTurn \triangleq turn = 1$$

$$DoorTurn \triangleq turn = 2$$

$$GearTurn \triangleq turn = 3$$

$$\begin{aligned} Init \triangleq\ & \wedge (mode = \text{“landing”}) \wedge (height = 0) \\ & \wedge (speed = 0) \wedge (door = door\_down) \\ & \wedge (gear = gear\_down) \wedge (turn = 1) \end{aligned}$$

$$\begin{aligned} AutopilotNext \triangleq\ & \wedge mode \in Modes \wedge height \in 0 \,..\, max\_height \\ & \wedge speed \in 0 \,..\, max\_speed \\ & \wedge (gear \neq gear\_down) \Rightarrow (height > threshold\_height) \\ & \wedge (mode = \text{“landing”}) \Rightarrow (gear = gear\_down) \\ & \wedge (mode = \text{“cruise”}) \Rightarrow ((gear = 0) \wedge (door = 0)) \\ & \wedge (height = 0) \Rightarrow (gear = gear\_down) \\ & \wedge AutopilotTurn \vee \text{UNCHANGED } \langle height, mode, speed \rangle \end{aligned}$$

$$\begin{aligned} DoorNext \triangleq\ & \wedge door \in 0 \,..\, door\_down \\ & \wedge ((speed > threshold\_speed) \Rightarrow (door = 0)) \\ & \wedge DoorTurn \vee \text{UNCHANGED } door \end{aligned}$$

$$\begin{aligned} GearNext \triangleq\ & \wedge gear \in 0 \,..\, gear\_down \\ & \wedge (gear \neq 0) \Rightarrow (door = door\_down) \\ & \wedge GearTurn \vee \text{UNCHANGED } gear \end{aligned}$$

$$\begin{aligned} SchedulerNext \triangleq\ & \wedge turn' = \text{IF } turn = 3 \text{ THEN } 1 \text{ ELSE } turn + 1 \\ & \wedge turn \in 1 \,..\, 3 \end{aligned}$$

$$Next \triangleq \wedge AutopilotNext \wedge GearNext$$

$$\land\ DoorNext \land SchedulerNext$$

$$vars \triangleq \langle mode,\ height,\ speed,\ door,\ gear,\ turn \rangle$$

$$Recurrence \triangleq \land \Box\Diamond(mode = \text{“landing”}) \land \Box\Diamond(mode = \text{“cruise”})$$

$$Spec \triangleq Init \land \Box[Next]_{vars} \land Recurrence$$

For brevity, we will let $mode \in 0..2$ in the discussion below. The components change in an interleaving way, based on the value of the variable *turn*. The scheduler changes its state in every step. So the scheduler changes in a noninterleaving way with respect to the other components. The specification has constant parameters $max\_height, \ldots$ 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

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

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 Chapter 7 is

$$
\begin{array}{l}
\mathit{AutopilotStep}(\mathit{door},\ \mathit{gear},\ \mathit{turn},\ \mathit{height},\ \mathit{mode},\ \mathit{speed},\\
\qquad\qquad \mathit{height}',\ \mathit{mode}',\ \mathit{speed}') \triangleq \\
\quad \land\ \mathit{turn} = 1 \land \mathit{door} \in 0\,..\,5 \land \mathit{gear} \in 0\,..\,5 \\
\quad \land\ \mathit{height} \in 0\,..\,100 \land \mathit{height}' \in 0\,..\,100 \\
\quad \land\ \mathit{mode} \in 0\,..\,2 \land \mathit{mode}' \in 0\,..\,2 \\
\quad \land\ \mathit{speed} \in 0\,..\,40 \quad \land \mathit{speed}' \in 0\,..\,40 \\
\quad \land\ \ \lor\ \land\ (\mathit{door} = 0) \land (\mathit{height}' \in 76\,..\,100) \\
\qquad\quad\ \land\ (\mathit{mode}' \in 1\,..\,2) \\
\qquad \lor\ (\mathit{gear} = 5) \land (\mathit{height}' \in 0\,..\,75) \\
\qquad \lor\ (\mathit{gear} = 5) \land (\mathit{mode}' = 0) \\
\qquad \lor\ \land\ (\mathit{height}' \in 76\,..\,100) \land (\mathit{mode}' = 2) \\
\qquad\quad\ \land\ (\mathit{speed}' \in 0\,..\,30) \\
\qquad \text{The two conjuncts below were used as care predicate.} \\
\quad \land\ \mathit{Inv}(\mathit{door},\ \mathit{gear},\ 1,\ \mathit{height},\ \mathit{mode},\ \mathit{speed}) \\
\quad \land\ (\exists\, \mathit{door},\ \mathit{gear} : \\
\qquad\quad \mathit{Inv}(\mathit{door},\ \mathit{gear},\ 2,\ \mathit{height},\ \mathit{mode},\ \mathit{speed}))'
\end{array}
$$

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

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

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

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

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

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

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

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

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

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

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

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

$$
\begin{array}{l}
SimplerAutopilotStep \triangleq \\
\quad \land gear \in 0 \,..\, 5 \land height \in 0 \,..\, 100 \land height' \in 0 \,..\, 100 \\
\quad \land mode \in 0 \,..\, 2 \land mode' \in 0 \,..\, 2 \\
\quad \land speed \in 0 \,..\, 40 \land speed' \in 0 \,..\, 40 \\
\quad \land \lor (gear = 5) \land (height' \in 0 \,..\, 75) \\
\qquad \lor (gear = 5) \land (mode' = 0) \\
\qquad \lor \land (gear \in 0 \,..\, 4) \land (height' \in 76 \,..\, 100)
\end{array}
$$

$$
\begin{array}{l}
\quad \land (mode \in 0 \,..\, 1) \land (mode' \in 1 \,..\, 2) \\
\lor \ \land (height' \in 76 \,..\, 100) \land (mode' = 2) \\
\quad \land (speed' \in 0 \,..\, 30) \\
\lor \ \land (height' \in 76 \,..\, 100) \land (mode' \in 1 \,..\, 2) \\
\quad \land (speed \in 31 \,..\, 40)
\end{array}
$$

$\land$ LET $turn = 1$ IN $InvH$

$\land (\exists\, turn,\, gear :\ InvH)'$

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

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

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

$$
\begin{array}{l}
SimplerGearModuleNext \ \triangleq \\
\quad \land\ gear \in 0 \,..\, 5 \land gear' \in 0 \,..\, 5 \\
\quad \land\ height \in 0 \,..\, 100 \land mode \in 0 \,..\, 2 \\
\quad \land\ speed \in 0 \,..\, 40 \\
\quad \land\ \ \lor (gear \in 0 \,..\, 4) \land (gear' = 0) \\
\qquad \lor (gear \in 1 \,..\, 5) \land (gear' = 5) \\
\qquad \lor\ \land (height \in 76 \,..\, 100) \land (mode = 2) \\
\qquad\quad \land (speed \in 0 \,..\, 30)
\end{array}
$$

$\land$ LET $turn \triangleq 3$ IN $InvH$

$\land$ (LET $turn \triangleq 1$

IN $\exists\, height,\, mode,\, speed :\ InvH)'$

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

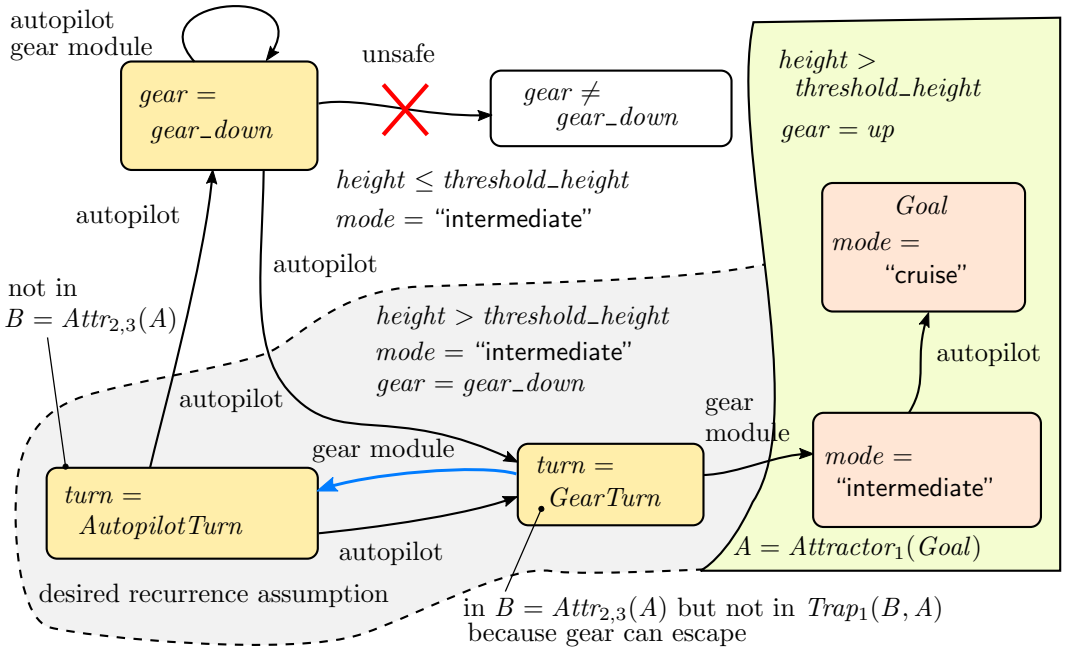


Figure 8.2: The reason why the algorithm of Section 6.3 is useful in the landing gear example.

define the care predicate (the last two conjuncts), which allowed for a simpler formula to be found.

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

For constructing liveness specifications, we start with the autopilot as the "root" module, and the door module and gear module lumped into a "team". The basic algorithm described in Section 6.2 cannot find a trap set, which demonstrates why the algorithm described in Section 6.3 is needed. The reason

is illustrated in Fig. 8.2, when the current goal of the autopilot is to enter cruise mode.[1] The autopilot can enter cruise mode from the intermediate mode when the gear is retracted (up). The gear can retract when extended, but it could also idle, leading to the state on the bottom left. In that state, it is the autopilot's turn, and the autopilot could idle, or change the height to less than or equal to *threshold_height*. This would prevent the gear from retracting. Therefore, the bottom left state is not in the team's attractor of A (the autopilot's attractor of cruise mode). This leads to an empty trap when the basic algorithm is used. By using the approach of Section 6.3, the *Basin* is enlarged to incorporate the bottom left state, and a weaker goal is generated for the gear. This goal takes into account that the gear should reach either A, or the top left state (which corresponds to several states). The autopilot has a choice to not go backwards, thus it can keep the behavior within the two bottom left states, until the gear does retract.

Algorithm 6.8 produces specifications for the autopilot and the lumped door and gear modules. Different mask parameters are used for each recurrence goal, and thus different interconnection architectures. The goal that is generated for the lumped modules is used as the goal in a recursive call to Algorithm 6.8. 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

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

and the resulting persistence objective $\Diamond\Box(AutopilotTrap \wedge (cnct = 0))$. As expected, the autopilot is allowed to keep waiting while the doors are still open ($door \in 1..5$ in second disjunct), and until the gear reacts, only *provided* the autopilot has reached and maintains the height above the threshold ($height \in 76..100$), and it keeps the mode to intermediate. The last two constraints are required because height below the threshold or mode equal to

[1] This figure corresponds to an earlier version of the specification.

landing would prevent the gear from being able to retract. Notice that the autopilot does not need to observe the gear state, only the door state, because when the doors close, the global invariant *Inv* implies that the gear has been retracted too. Therefore, the specification of the autopilot in this interconnection mode is expressed without occurrence of the variable *gear*.

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

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

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

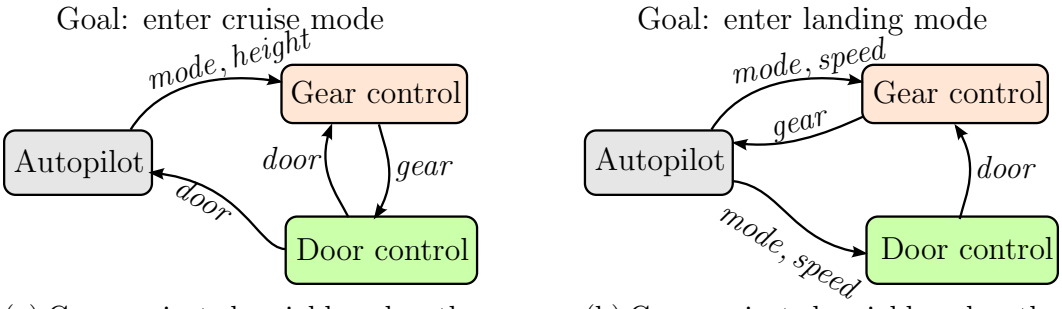
$$
\begin{array}{l}
Gear\_Trap \triangleq \\
\wedge\, turn \in 1 \,..\, 3 \wedge door \in 0 \,..\, 5 \\
\wedge\, gear \in 0 \,..\, 5 \wedge height \in 0 \,..\, 100 \wedge mode \in 0 \,..\, 2 \\
\wedge \quad \vee \wedge (turn = 2) \wedge (gear = 0) \\
\qquad\quad \wedge (height \in 76 \,..\, 100) \wedge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

and for the door module

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

Again, these goals are conditioned using the current interconnection *cnct*. The above specifications can be understood as follows. The gear assumes that the doors will close, provided the gear has retracted itself (conjunct $gear = 0$). The gear cannot assume that the doors will close while the gear is still extended, because that would be too strong an assumption. It would be realizable by the gear, but unrealizable by the doors. Similar to what we remarked about the autopilot earlier, this is an error that could arise if we specified the subsystem goals by hand, instead of generating them automatically. Provided the gear has retracted, it is allowed to wait until the door retracts, and also until it is the autopilot's turn. Note that the gear does not receive *speed* information in this interconnection, which is shown in Fig. 8.3a. For the doors, the requirement is that if the gear has retracted ($gear = 0$), then the doors should not be open ($door \in 1..5$).

The above discussion corresponds to the interconnection architecture while the autopilot has cruise mode as its current goal. A different interconnection architecture, shown in Fig. 8.3b 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



(a) Communicated variables when the goal is cruise mode.

(b) Communicated variables when the goal is landing mode.

Figure 8.3: Variables communicated between subsystems, depending on the current goal.

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, and only then can the autopilot enter landing mode). An interesting observation regarding the connectivity in Fig. 8.3b is that the gear needs to observe both *mode* and *speed*. This requirement results because the gear module needs to be able to observe globally that the doors are still within *D_Door_module*, which requires information about the *mode* and *speed*. However, we would expect this to be information necessary only to the door module. Indeed, by using the complement of the subsystem goal as *Within* to change $Stay := Observable(D, Inv, Inv, Player)$ in Algorithm 6.8 to $Stay := Observable(D, Within \wedge Inv, Inv, Player)$, as described in Section 6.7.2, 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 [61].

Chapter 9

OPEN SYSTEM SPECIFICATIONS

9.1 Assume-guarantee specifications

9.1.1 Forms of stepwise implication

Open-system specifications constrain the behavior of component variables in presence of appropriate environment behavior. This kind of relation resembles logical implication ($\Rightarrow$), suggesting defining a binary operator. In this section, we compare two operators of this kind:

- The TLA$^+$ operator $\stackrel{+}{\triangleright}$ (named *WhilePlus* below)
- A quinary operator based on work by Klein and Pnueli [96, Eq. 3 on p. 163].

Klein and Pnueli defined the (PastRTLA$^+$) formula

$$
\begin{array}{lll}
KP \triangleq & \lor \neg EnvInit & \\
& \lor \land SysInit & \\
& \quad \land \Box(UpToNow(EnvNext) \Rightarrow SysNext) & \text{Stepwise implication} \\
& \quad \land \lor \Diamond\neg EnvNext & \text{Environment error (“liveness gap”)} \\
& \qquad \lor \exists\, i \in 0 \mathinner{..} n : \Diamond\Box P(i) & \text{generalized Streett pair} \\
& \qquad \lor \forall\, j \in 0 \mathinner{..} m : \Box\Diamond R(j) &
\end{array}
$$

The formula $UpToNow(EnvNext)$ is true when the action *EnvNext* has been true in all previous steps and the current step. In LTL, *UpToNow* is called “historically” [128].

The “assume-guarantee” character of *RawWhile* is manifest in the stepwise implication and the environment error. The “GR(1)” character is manifest in the Streett pair that constrains the component. Thus, the assume-guarantee form we choose is orthogonal to whether the liveness part is a GR(1) or GR(k) property, so we can lump all liveness in order to obtain the general form

$$
\begin{array}{l}
RawWhile(EnvInit,\, SysInit,\, EnvNext,\, SysNext,\, Liveness) \triangleq \\
\quad \lor \neg EnvInit \\
\quad \lor \land SysInit \\
\qquad \land \Box(UpToNow(EnvNext) \Rightarrow SysNext)
\end{array}
$$

$\land\ \lor \Diamond\neg \mathit{EnvNext}$
$\quad \lor \mathit{Liveness}$

For reasons discussed in Section 9.6 we study the operator *RawWhilePlus*, instead of *RawWhile*, defined as follows:

$\mathit{RawWhilePlus}(\mathit{IeP}(\_, \_), \mathit{EnvInit}, \mathit{SysInit}, \mathit{EnvNext}, \mathit{SysNext}, \mathit{Le}, \mathit{Ls}) \triangleq$
$\quad \lor \neg\exists\, p, q : \mathit{IeP}(p, q)$ satisfiability of *EnvInit*
$\quad \lor \land \mathit{SysInit}$
$\qquad \land \mathit{EnvInit} \Rightarrow \land \Box(\mathit{Earlier}(\mathit{EnvNext}) \Rightarrow \mathit{SysNext})$
$\qquad\qquad\quad \land (\Box \mathit{EnvNext} \land \mathit{Le}) \Rightarrow \mathit{Ls}$

The main difference between *RawWhile* and *RawWhilePlus* is in using the operator *Earlier* instead of *UpToNow* [67, 65, 88]. The expression $\mathit{Earlier}(\mathit{EnvNext})$ is true when the action *EnvNext* has been true in all previous steps (omitting the current step). We use the operator *RawWhilePlus* with the argument *EnvInit* defined as $\mathit{IeP}(x, y)$, where x, y are variables declared in the relevant context. So the argument *EnvInit* could be omitted, but that would require adding x and y as arguments, and thus cluttering this definition.

It can be shown that *RawWhilePlus* is expressible with the quinary operator

$\mathit{RawWhilePlusDisj}(\mathit{InitC}, \mathit{InitD}, \mathit{EnvNext}, \mathit{SysNext}, \mathit{Liveness}) \triangleq$
$\quad \mathit{InitC} \Rightarrow \land \mathit{InitD}$
$\qquad\qquad\quad \land \Box(\mathit{Earlier}(\mathit{EnvNext}) \Rightarrow \mathit{SysNext})$
$\qquad\qquad\quad \land \mathit{Liveness} \lor \Diamond\neg \mathit{EnvNext}$

It does not matter whether the main operator is disjunction. Any property expressed with the operator *RawWhilePlusDisj* is expressible using the following operator:

$\mathit{RawWhilePlusConj}(\mathit{InitA}, \mathit{InitB}, \mathit{EnvNext}, \mathit{SysNext}, \mathit{Liveness}) \triangleq$
$\quad \land \mathit{InitB}$
$\quad \land \mathit{InitA} \Rightarrow \land \Box(\mathit{Earlier}(\mathit{EnvNext}) \Rightarrow \mathit{SysNext})$
$\qquad\qquad\quad \land \mathit{Liveness} \lor \Diamond\neg \mathit{EnvNext}$

The previous operators are in raw TLA$^+$ with past operators (PastRTLA$^+$) (see Section 3.2). The operator *RawWhilePlus* is suitable for algorithmic implementation [153]. We next examine the relation between *WhilePlus* and *RawWhilePlus*, which has also been examined under more restrictive assumptions in [88].

9.1.2 Stepwise form of $\overset{+}{\triangleright}$

Arguments represented as machine-closed pairs Machine-closure of the pair

$$EnvInit \wedge \Box[EnvNext]_v, \quad Le$$

and of the pair

$$InitSys \wedge \Box[SysNext]_v, \quad Ls$$

imply equivalence of the properties defined using the operators *RawWhilePlus* and $\overset{+}{\triangleright}$, as implied by the below theorem. In absence of machine-closure, these two operators are not equivalent, as discussed in more detail in Section 9.2.[1]

A syntactic definition of closure [113, 11]

$$\begin{aligned}
MustUnstep(b) \triangleq\ & \wedge\ b = \text{TRUE} \\
& \wedge\ \Box[b' = \text{FALSE}]_b \\
& \wedge\ \Diamond(b = \text{FALSE})
\end{aligned}$$

$$SamePrefix(b,\ u,\ x) \triangleq \Box(b \Rightarrow (u = x))$$

$$Front(P(\_,\ \_),\ x,\ b) \triangleq \exists\, u :\ P(u) \wedge SamePrefix(b,\ u,\ x)$$

$$Cl(P(\_),\ x) \triangleq \forall\, b :\ MustUnstep(b) \Rightarrow Front(P,\ x,\ b)$$

Each property is decomposable into safety and liveness [12].

$$SafetyPart(P(\_),\ x) \triangleq Cl(P,\ x)$$

$$LivenessPart(P(\_),\ x) \triangleq SafetyPart(P,\ x) \Rightarrow P(x)$$ [88, Sec. 2.3 on p. 54]

Below we use this operator with adapted arity.

$$\begin{aligned}
&IsMachineClosed(S(_),\ L(_)) \triangleq \\
&\quad \text{LET}\ SL(u) \triangleq S(u) \wedge L(u) \\
&\quad \text{IN}\ \ \forall\, x :\ S(x) \equiv Cl(SL,\ x)
\end{aligned}$$

$$\begin{aligned}
&\text{THEOREM}\ \ PhiEquivRawPhi \triangleq \\
&\quad \text{ASSUME} \\
&\qquad \text{VARIABLE}\ x,\ \text{VARIABLE}\ y, \\
&\qquad \text{NEW}\ sigma, \\
&\qquad IsABehavior(sigma), \\
&\qquad \text{CONSTANT}\ IeP(_,\ _), \\
&\qquad \text{CONSTANT}\ IsP(_,\ _), \\
&\qquad \text{CONSTANT}\ NeP(_,\ _,\ _,\ _),
\end{aligned}$$

[1] A proof of the theorem *PhiEquivRawPhi* is in the module *WhilePlusTheorems*.

CONSTANT $NsP(\_, \_, \_, \_)$,

TEMPORAL Le, TEMPORAL Ls, thus TLA+ formulas

$\land \forall\, u,\, v :\ IeP(u,\, v) \in \text{BOOLEAN}$

$\land \forall\, u,\, v :\ IsP(u,\, v) \in \text{BOOLEAN}$

$\land \forall\, a,\, b,\, c,\, d :\ NeP(a,\, b,\, c,\, d) \in \text{BOOLEAN}$

$\land \forall\, a,\, b,\, c,\, d :\ NsP(a,\, b,\, c,\, d) \in \text{BOOLEAN}$,

LET

$v \triangleq \langle x,\, y \rangle$

$Is \triangleq IsP(x,\, y)$

$Ie \triangleq IeP(x,\, y)$

$Ne \triangleq NeP(x,\, y,\, x',\, y')$

$Ns \triangleq NsP(x,\, y,\, x',\, y')$

IN

$\land\, IsMachineClosed(Ie \land \Box[Ne]_v,\, Le)$

$\land\, IsMachineClosed(Is \land \Box[Ns]_v,\, Ls)$

PROVE

LET

$v \triangleq \langle x,\, y \rangle$

$Is \triangleq IsP(x,\, y)$

$Ie \triangleq IeP(x,\, y)$

$Ne \triangleq NeP(x,\, y,\, x',\, y')$

$Ns \triangleq NsP(x,\, y,\, x',\, y')$

$A \triangleq Ie \land \Box[Ne]_v \land Le$

$G \triangleq Is \land \Box[Ns]_v \land Ls$

$Phi \triangleq A \stackrel{+}{\triangleright} G$

$EnvNext \triangleq [Ne]_v$

$SysNext \triangleq [Ns]_v$

$RawPhi \triangleq RawWhilePlus($

$IeP,\, Ie,\, Is,$

$EnvNext,\, SysNext,\, Le,\, Ls)$

IN

$(sigma,\, 0 \models RawPhi) \quad \equiv \quad (sigma \models Phi)$

PROOF

BY $RawPhiImpliesPhi,\ PhiImpliesRawPhi$

9.1.3 Shifting liveness around

It is of practical interest to know when we can shift liveness conjuncts from the environment to the component property, or vice versa. Shifts of this kind negate the liveness, turning recurrence into persistence, and vice versa.

The specifier usually imagines an "environment"

$$Env \triangleq EnvInit \wedge \Box[EnvNext]_v$$

and a "component"

$$Sys \triangleq SysInit \wedge \Box[SysNext]_v \wedge Liveness$$

However, *RawWhilePlusDisj* depends on five arguments ($EnvInit, \ldots, Liveness$); not two ($Env, Sys$). This difference is essential, in that there are properties expressible with *RawWhilePlusDisj* but inexpressible with any binary operator that takes *Env* and *Sys* as arguments (we show this inexpressibility in Section 9.2).

One pitfall is to change any of the formulas $EnvInit, \ldots, Liveness$, while keeping *Env* and *Sys* the same. If such a change leads from a machine-closed to machine-unclosed representation of either *Env* or *Sys*, then in general the property *RawWhilePlusDisj* changes.

Assume that both A and G are written using machine-closed representations. If we shift the property Le to G, then the closure of $A \xrightarrow{+} G$ remains unchanged. More precisely

THEOREM
  ASSUME
    $\wedge\ A \equiv (Ie \wedge \Box[Ne]_v \wedge Le)$
    $\wedge\ Cl(A) \equiv (Ie \wedge \Box[Ne]_v)$ The pair $Ie \wedge \Box[Ne]_v, Le$ is machine-closed.
    $\wedge\ G \equiv (Is \wedge \Box[Ns]_v \wedge Ls)$
The pair $Is \wedge \Box[Ns]_v, Ls$ is machine-closed. This implies that the pair $Is \wedge \Box[Ns]_v, Ls \vee \neg Le$ is machine-closed (see Q below).
    $\wedge\ Cl(G) \equiv (Is \wedge \Box[Ns]_v)$
  PROVE
    LET
      $P \triangleq Ie \wedge \Box[Ne]_v$
      $Q \triangleq Is \wedge \Box[Ns]_v \wedge (Le \Rightarrow Ls)$
    IN

$$A \stackrel{+}{\triangleright} G \;\equiv\; P \stackrel{+}{\triangleright} Q$$

This theorem can be used in the reverse direction too, i.e., to shift Le to the first argument (the "environment" property). In that case, persistence disjuncts from the formula $(Le \Rightarrow Ls)$ within Q are shifted to become recurrence conjuncts in the formula Le within the formula that represents A.

In absence of machine-closure, the above rewriting can change the open-system property. There are two cases in GR(1):

1. Shifting recurrence from A to persistence in Q. If the pair $Ie \wedge \Box[Ne]_v, Le$ is machine-unclosed, then $Cl(P)$ is weaker than $Cl(A)$ (because Le imposes a safety constraint on $Ie \wedge \Box[Ne]_v$), so this transformation relaxes the assumption's closure.

   If the pair $Is \wedge \Box[Ns]_v, Ls$ is machine-unclosed, then relaxing the liveness to $(Ls \vee \neg Le)$ can weaken the closure (not necessarily). Thus, $Cl(Q)$ may be weaker than $Cl(G)$.

   The combined effect of relaxing A to P and changing G to Q in this way is nontrivial.

2. Shifting persistence from Q to recurrence in A. This transformation strengthens the guarantee from Q to G. If the pair $Is \wedge \Box[Ns]_v, Ls$ is not machine-closed, then the closure $Cl(G)$ may be stronger than $Cl(Q)$.

   This shift conjoins Le to $Ie \wedge \Box[Ne]_v$ inside A, so $Cl(A)$ is stronger than $Cl(P)$ if the pair $Ie \wedge \Box[Ne]_v, Le$ is machine-unclosed.

To summarize, in absence of machine-closure, shifting liveness from assumption to guarantee relaxes the closure of the assumption and possibly of the guarantee. Shifting liveness from the guarantee to the assumption strengthens the closure of the assumption, and possibly of the guarantee too.

In both cases, the combined effect from changing both arguments appears to have a nontrivial effect on the open-system property that the two properties define when fed to the operator $\stackrel{+}{\triangleright}$.

Remark 10. Lamport defined the operator "as long as" directly in semantics [110, $\trianglelefteq$ on p. 220]. This operator is similar to *RawWhile*. The operator "one step longer" [110, $\triangleleft$ on p. 220] is equivalent to *StepwiseImpl*.[2] □

[2] The operator *StepwiseImpl* is defined in the module *WhilePlusTheorems*.

Remark 11. A reusable definition of $\stackrel{+}{\triangleright}$ within TLA$^+$ has to include at least one argument for variables, because there is no way to refer to all declared variables from within TLA$^+$. □

Remark 12. The operator $\mathit{WhilePlus}(A, G, x, y)$ does differ from $\stackrel{+}{\triangleright}$ if A or G depend on variables other than x and y. This is a potential cause of errors by the specifier when using *WhilePlus*. □

9.2 Degrees of freedom needed for representation

9.2.1 Overview

Klein and Pnueli defined a temporal formula that describes those open-system properties that are implemented by a known fixpoint algorithm [96, 153]. The quinary operator *RawWhilePlus* is the strictly causal version of the Klein-Pnueli formula (Section 9.1). In this section we define the operator

$$\begin{array}{l}\mathit{RawWhilePlus}(\mathit{EnvInit},\ \mathit{SysInit},\ \mathit{EnvNext},\ \mathit{SysNext},\ \mathit{Liveness}) \triangleq \\ \quad \wedge\ \mathit{SysInit} \\ \quad \wedge\ \mathit{EnvInit} \Rightarrow\ \wedge\ \Box(\mathit{Earlier}(\mathit{EnvNext}) \Rightarrow \mathit{SysNext}) \\ \qquad\qquad\qquad\ \wedge\ \mathit{Liveness} \vee \Diamond\neg\mathit{EnvNext}\end{array}$$

This operator is called *RawWhilePlusConj* in Section 9.1 and for satisfiable *IeP* the *RawWhilePlus* from Section 9.1 is equivalent to this *RawWhilePlus*. The above suffices in this section.

When writing specifications, we tend to think in terms of two properties: an "environment" and a "component". In accord with this thinking, the operator $\stackrel{+}{\triangleright}$ takes two arguments. Can we define *RawWhilePlus* too as a binary operator?

> Is any property describable by *RawWhilePlus* also describable by some binary operator over *Env* and *Sys*? (The answer is negative.)

Earlier we showed that *RawWhilePlus* is the stepwise form of $\stackrel{+}{\triangleright}$ (i.e., going from two to five arguments). So any property expressed using $\mathit{Env} \stackrel{+}{\triangleright} \mathit{Sys}$ can be expressed using *RawWhilePlus*, by feeding to *RawWhilePlus* the machine-closed representation of *Env* and *Sys*. What we investigate now is the opposite direction.

The answer depends on whether any property describable by applying the operator *RawWhilePlus* to the five arguments

$$EnvInit, SysInit, EnvNext, SysNext, Liveness$$

can also be described by the application of *some* operator to the two arguments

$Env \triangleq EnvInit \wedge \Box[EnvNext]_v$

$Sys \triangleq SysInit \wedge \Box[SysNext]_v \wedge Liveness$

A property defined by *RawWhilePlus* changes when we change any of the five arguments (except for corner cases). So we expect the two arguments *Env* and *Sys* to depend on all five arguments $EnvInit, \ldots, Liveness$. As expected, we show that the two properties *Env* and *Sys* alone cannot capture all the information that the *RawWhilePlus* operator uses from the five arguments.

9.2.2 Example

In other words, if we are given *Env* and *Sys* in a canonical representation with state machine and liveness different than those used in *RawWhilePlus*, then we cannot find out what those properties were. If we try to make a property from this *Env* and *Sys*, it will be the same for all representations of *Env* and *Sys*, unlike the *RawWhilePlus* properties that result for those representations, which are different.

Fig. 9.1 illustrates the concept of multiplicity of representations. Any property P can be described by several different pairs of safety and liveness, for example $P \equiv P_1$ and $P \equiv P_2$, where $P_1 \triangleq I_1 \wedge \Box[Next_1]_v \wedge L_1$ and $P_2 \triangleq I_2 \wedge \Box[Next_2]_v \wedge L_2$. If these representations are machine-closed

$$\wedge\ Cl(I_1 \wedge \Box[Next_1]_v \wedge L_1) \equiv I_1 \wedge \Box[Next_1]_v$$
$$\wedge\ Cl(I_2 \wedge \Box[Next_2]_v \wedge L_2) \equiv I_2 \wedge \Box[Next_2]_v$$

then it follows that using $I_1, Next_1, L_1$ in *RawWhilePlus* yields the same property as that obtained by using P in $\stackrel{+}{\rightarrow}$. This holds if we replace $I_1, Next_1, L_1$ by $I_2, Next_2, L_2$. But for a machine-unclosed representation

$$P_3 \triangleq I_3 \wedge \Box[Next_3]_v \wedge L_3$$

where $P \equiv P_3$, using $I_3, Next_3, L_3$ in *RawWhilePlus* can yield a *different* property than what we get by using P in $\stackrel{+}{\rightarrow}$. So given *Env* and *Sys* in some representation, it is impossible to ensure that feeding that representation to

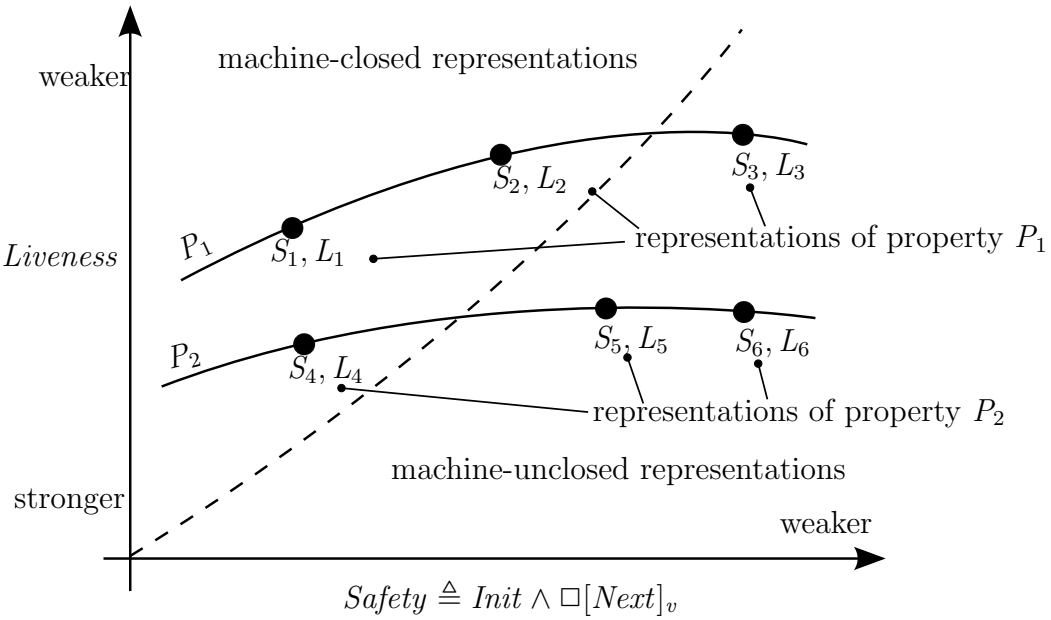


Figure 9.1: Each temporal property P can be represented in multiple ways as a conjunction $S \wedge L$ of a safety formula S and a liveness formula L (except for corner cases). For some pairs S, L the liveness property does not impose a safety constraint on S, but for others it does. The first case is called machine-closed, the second machine-unclosed. The ordering of properties along axes in this figure is a simplification (two properties can be incomparable).

RawWhilePlus yields the same property as feeding *Env* and *Sys* to $\overset{+}{\rightarrow}$. If we convert the representation to a machine-closed one, then we get the same property with $\overset{+}{\rightarrow}$.

9.2.3 Useful notions

Three notions are relevant to the discussion below (listed from stronger to weaker):

1. Equivalence of two properties $\models P \equiv Q$ (the collections of behaviors that P and Q describe coincide)

2. Equisynthesizability $\models \forall f : \mathit{IsARealization}(f, P) \equiv \mathit{IsARealization}(f, Q)$ (the collections of implementations of P and Q coincide)

3. Equirealizability of two properties $\models \mathit{IsRealizable}(P) \equiv \mathit{IsRealizable}(Q)$ (P and Q are both realizable or both unrealizable).

For example, if the variable y represents the component, then the properties

$P \triangleq$ TRUE and $Q \triangleq \Box(y = 0)$ are equirealizable. They are not equisynthesizable.

9.2.4 Multiple representations for a closed system

There would not be any problem if any property *Sys* (*Env*) had a unique representation using an initial condition *Init*, and action *Next*, and a liveness formula. There would be only a single way of expressing *Sys* with these kinds of subformulas, and a single choice of arguments for *RawWhilePlus*. This is not the case, as shown below.

For any state predicates $I1, I2, v$ and actions $A1, A2$, if $I1 \equiv I2$ and $A1 \equiv A2$, then the RTLA$^+$ formula $(I1 \land \Box A1) \equiv (I2 \land \Box A2)$ and the TLA$^+$ formula $(I1 \land \Box[A1]_v) \equiv (I2 \land \Box[A2]_v)$ are both valid. The converse does *not* hold, because an action may lead to deadends. In other words, a property of the form $\Box Next$ is describable by multiple inequivalent actions.[3]

PROPOSITION *RawTails* $\triangleq$ in RTLA+
PROVE
  LET
    $A1 \triangleq$ FALSE
    $A2 \triangleq (x = 1) \land (x' = 2)$
  IN
    $\land (\Box A1) \equiv (\Box A2)$
    $\land \exists\, x : \neg(A1 \equiv A2)$

The corresponding phenomenon in TLA$^+$ is observed in the presence of liveness. Safety alone cannot give rise to deadends in TLA$^+$, due to stutter-extensibility (an infinite stuttering tail). More precisely, the following proposition is valid.[4]

PROPOSITION *InvertingTails* $\triangleq$
  ASSUME
    STATE v,
    ACTION $A1$, ACTION $A2$,
    $(\Box[A1]_v) \equiv (\Box[A2]_v)$
  PROVE
    $\langle A1\rangle_v \equiv \langle A2\rangle_v$

Conjoining an initial condition preserves the above inversion result over reachable states, but not at unreachable states.[5]

[3] A proof of *RawTails* is in the module *Representation*.
[4] A proof of *InvertingTails* is in the module *Representation*.
[5] A proof of *InvertingStateMachines* is in the module *Representation*.

PROPOSITION $InvertingStateMachines \triangleq$
  ASSUME
    STATE $I1$, STATE $I2$, STATE v,
    ACTION $A1$, ACTION $A2$,
    LET
      $SM1 \triangleq I1 \land \Box[A1]_v$
      $SM2 \triangleq I2 \land \Box[A2]_v$
    IN
      $SM1 \equiv SM2$
  PROVE
    LET
      $SM1 \triangleq I1 \land \Box[A1]_v$
    IN
      $\land\, I1 \equiv I2$
      $\land\, SM1 \Rightarrow \Box[A1 \land A2]_v$

Unfortunately, in TLA$^+$ uniqueness disappears when liveness is included, because deadends can again form, as in *RawTails* above (in RTLA$^+$).

9.2.5 Mismatch between RawWhilePlus and WhilePlus in absence of machine closure

With liveness present, we prove that no binary operator of the properties

$Env \triangleq EnvInit \land \Box EnvNext$
$Sys \triangleq SysInit \land \Box SysInit \land Liveness$

can express the property

$$RawWhilePlus(EnvInit, SysInit, EnvNext, SysNext, Liveness),$$

simply because we can change the latter while keeping *Env* and *Sys* the same. In other words, there are two quintuples

$$EnvInit_1, \ldots, Liveness_1 \qquad \text{and} \qquad EnvInit_2, \ldots, Liveness_2$$

that yield the same *Env* and *Sys*, but different *RawWhilePlus* properties. So any binary operator that depends on *Env* and *Sys* cannot describe both properties.

We give examples of both stutter-sensitive and stutter-invariant properties. That the *RawWhilePlus* properties are "different" means they are inequivalent. In these examples, the properties are not equirealizable.

9.2.5.1 Stutter-sensitive example (RTLA$^+$)

Inequivalence The first quintuple is

$$\begin{aligned}
&\mathit{EnvInit}1 \triangleq \text{TRUE}\\
&\mathit{EnvNext}1 \triangleq y = 0\\
&\mathit{SysInit}1 \triangleq y = 0\\
&\mathit{SysNext}1 \triangleq (y \in 0\,..\,1) \land (y' = y + 1)\\
&\mathit{Liveness}1 \triangleq \text{TRUE}
\end{aligned}$$

The second quintuple differs by only

$$\mathit{SysNext}2 \triangleq \text{FALSE}$$

Conjunction for the first quintuple yields

$$\begin{aligned}
\mathit{Env}1 &\triangleq (\mathit{EnvInit}1 \land \Box\mathit{EnvNext})\\
&\equiv \Box(y = 0)\\
\mathit{Sys}1 &\triangleq (\mathit{SysInit}1 \land \Box\mathit{SysNext}1 \land \mathit{Liveness}1)\\
&\equiv \text{FALSE}
\end{aligned}$$

and the same pair results for the second quintuple too

$$\begin{aligned}
&\mathit{Env}2 \equiv \Box(y = 0)\\
&\mathit{Sys}2 \equiv \text{FALSE}
\end{aligned}$$

The *RawWhilePlus* properties are

$$\begin{aligned}
Q1 &\triangleq \mathit{RawWhilePlus}(\mathit{EnvInit}1,\ \mathit{SysInit}1,\ \mathit{EnvNext}1,\ \mathit{SysNext}1,\ \mathit{Liveness}1)\\
&\equiv \land\ y = 0\\
&\quad \land \Box \lor \neg\mathit{Earlier}(y = 0)\\
&\qquad \lor \land\ y \in 0\,..\,1\\
&\qquad\quad \land\ y' = y + 1\\
Q2 &\triangleq \mathit{RawWhilePlus}(\mathit{EnvInit}2,\ \mathit{SysInit}2,\ \mathit{EnvNext}2,\ \mathit{SysNext}2,\ \mathit{Liveness}2)\\
&\equiv \text{FALSE}
\end{aligned}$$

Clearly, the properties $Q1$ and $Q2$ are inequivalent ($Q1$ is satisfied by a behavior where y takes the values $0, 1, 2, 2, \ldots$, as illustrated in Fig. 9.2, whereas $Q2$ is unsatisfiable). Any binary operator will yield the same property when applied to the pair $\mathit{Env}1, \mathit{Sys}1$ and to the pair $\mathit{Env}2, \mathit{Sys}2$ (because $\mathit{Env}1 \equiv \mathit{Env}2$ and $\mathit{Sys}1 \equiv \mathit{Sys}2$). This property cannot be equivalent to both $Q1$ and $Q2$. So no binary operator can express *RawWhilePlus* via *Env* and *Sys*.

Inequirealizability Assume that the system controls the variable y. Then property $Q1$ is realized by an implementation that initializes $y = 0$ and increments variable y by 1 in each step. Property $Q2$ is unrealizable. So $Q1$ and $Q2$ are not equirealizable (thus neither equisynthesizable).

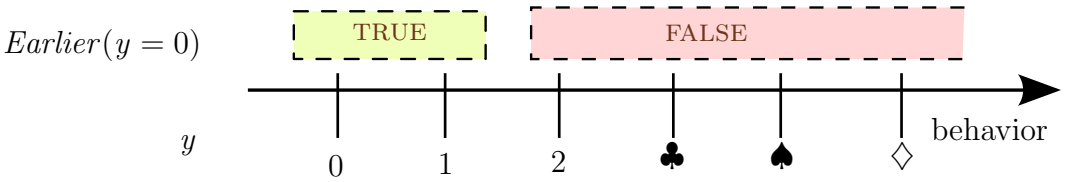


Figure 9.2: *Into the wild*: after a few steps the component violates the environment action. From there on it can behave arbitrarily.

We started by presenting an example in RTLA$^+$ to emphasize that the above observation is unrelated to stutter-invariance. The stutter-invariant example given below obtains the same effect via interaction between safety and liveness.

9.2.5.2 Stutter-invariant example

We now want to convert the RTLA$^+$ example of the previous section to a TLA$^+$ example (so from stutter-sensitive to stutter invariant). For this objective it is useful to consider the stutter-closure of a property defined by an RTLA$^+$ formula, which is a property expressible in TLA$^+$. In particular, the stutter closure of the RTLA$^+$ property $\Box \mathit{Next}(x, x')$ is

$$
\begin{array}{l}
\mathit{StutterClosure}(\mathit{Next}(\_, \_),\ x) \triangleq \\
\quad \land \Box[\mathit{Next}(x,\ x')]_x \\
\quad \land \ \lor \Box\Diamond\langle \text{TRUE} \rangle_x \\
\qquad \lor \Box\Diamond \text{ENABLED}\ \land \mathit{Next}(x,\ x') \\
\qquad\qquad\qquad\quad \land \text{UNCHANGED}\ x
\end{array}
$$

where *Next* is a constant operator, and x a variable (alternatively, we need to know all the declared variables). This property is satisfied by a behavior that takes only *Next*-steps, and stops changing x only at states where *Next* allows so.

We apply *StuttterClosure* to create an example that involves stutter-invariant properties that are the stuttering-closures of the properties from the example in Section 9.2.5.1.

$$
\begin{array}{ll}
\mathit{EnvInit}1 & \triangleq \ \text{TRUE} \\
\mathit{EnvNext}1 & \triangleq \ y = 0 \\
\mathit{SysInit}1 & \triangleq \ y = 0 \\
\mathit{SysNext}1 & \triangleq \ [(y \in 0\,..\,1) \land (y' = y + 1)]_y \\
\mathit{Liveness}1 & \triangleq \ \Box\Diamond\langle \text{TRUE} \rangle_y
\end{array}
$$

The second quintuple differs by only

$SysNext2 \triangleq$ FALSE

Notice that $Liveness1$ results from $StutterClosure$, because the disjunct

$$\begin{aligned}&\Box\Diamond\text{ENABLED } \langle\rangle \wedge (y \in 0\,..\,1) \wedge (y' = y + 1)\\ &\qquad\qquad \wedge \text{UNCHANGED } y\\ &\equiv \text{FALSE}\end{aligned}$$

Similarly to the stutter-sensitive example, both cases yield the same component property

$$\begin{aligned}Env1 &\equiv \Box(y = 0)\\ Sys1 &\equiv \wedge y = 0\\ &\quad \wedge \Box[(y \in 0\,..\,1) \wedge (y' = y + 1)]_y\\ &\quad \wedge \Box\Diamond\langle\text{TRUE}\rangle_y\\ &\equiv \text{FALSE}\end{aligned}$$

and

$$\begin{aligned}Env2 &\equiv \Box(y = 0)\\ Sys2 &\equiv \text{FALSE}\end{aligned}$$

The $RawWhilePlus$ properties are

$$\begin{aligned}Q1 &\triangleq RawWhilePlus(EnvInit1, SysInit1, EnvNext1, SysNext1, Liveness1)\\ &\equiv \wedge y = 0\\ &\quad \wedge \Box \vee \neg Earlier(y = 0)\\ &\qquad\quad \vee y' = y\\ &\qquad\quad \vee \wedge y \in 0\,..\,1\\ &\qquad\qquad \wedge y' = y + 1\\ &\quad \wedge \Box(y = 0) \Rightarrow \Box\Diamond\langle\text{TRUE}\rangle_y\\ Q2 &\triangleq RawWhilePlus(EnvInit2, SysInit2, EnvNext2, SysNext2, Liveness2)\\ &\equiv \text{FALSE}\end{aligned}$$

Again, the properties $Q1$ and $Q2$ are inequivalent. The corresponding pairs $Env1, Sys1$ and $Env2, Sys2$ though are identical. Again, no binary operator that depends on Env and Sys can equal two different properties ($Q1$ and $Q2$). To emphasize this, the value of such an operator would be

$$MagicOperator(\Box(y = 0), \text{FALSE})$$

and equal to both $MagicOperator(Env1, Sys1)$ and $MagicOperator(Env2, Sys2)$, but unequal to $Q1$ or unequal to $Q2$.

In addition, property $Q2$ is unrealizable, whereas $Q1$ is realizable (since there exists some behavior that satisfies it, and $Q1$ depends on only variable y, which the component controls).

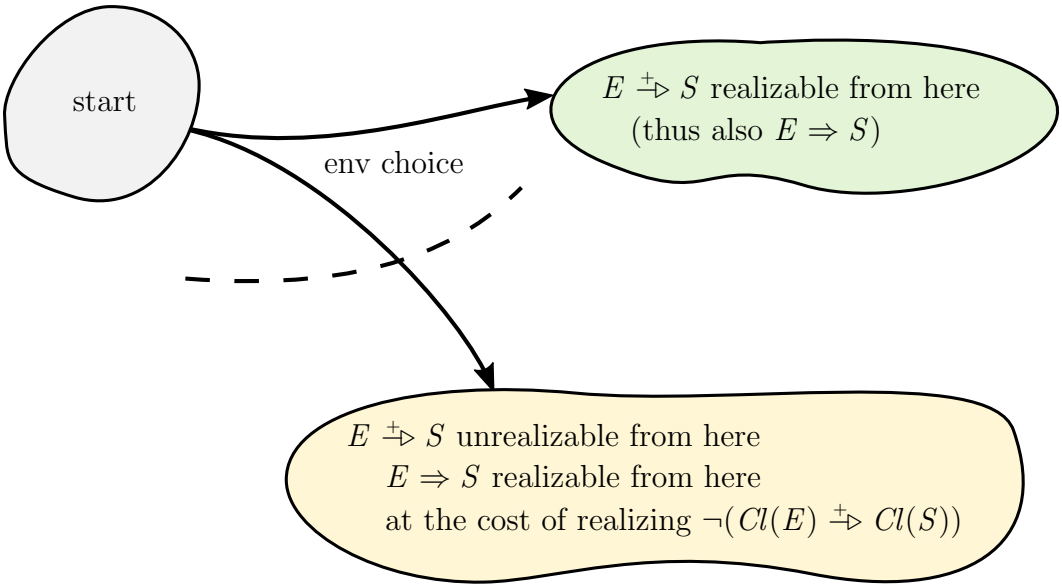


Figure 9.3: The property $E \Rightarrow S$ can be realizable at the cost of violating $E \stackrel{+}{\triangleright} S$. The environment can have a choice between these two outcomes, even when $S \stackrel{+}{\triangleright} E$ is realizable by it. Relative well-separation [131] implies that an environment that implements $Cl(E)$ won't accidentally let the system realize $(E \Rightarrow S) \wedge \neg(E \stackrel{+}{\triangleright} S)$.

9.2.6 Notions of well-separation

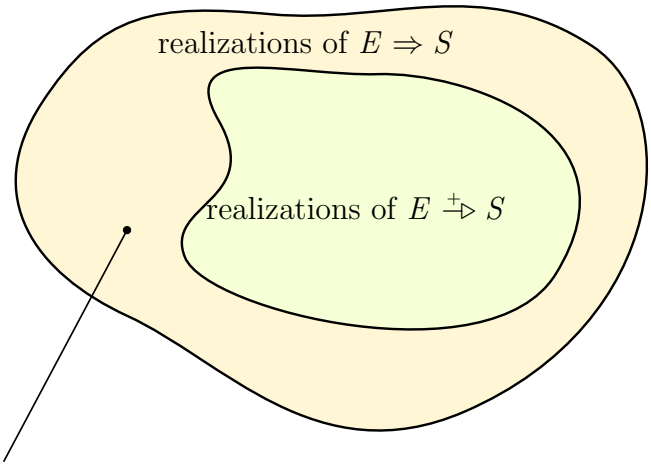
Klein and Pnueli study the equisynthesizability of *RawWhile* and $E \Rightarrow S$ [96].[6] We are interested in comparing $\stackrel{+}{\triangleright}$ and $E \Rightarrow S$.[7] The property $\stackrel{+}{\triangleright}$ is not equivalent to $E \Rightarrow S$, nor equirealizable [131].[8] Assuming safety as E, these properties are equisynthesizable if both the environment specification $Cl(S) \stackrel{+}{\triangleright} E$ is realizable (by the environment) *and* the assembled system cannot reach states from where $\neg(Cl(E) \stackrel{+}{\triangleright} Cl(S))$ is realizable by the system. This condition corresponds to well-separation with respect to a guarantee [131]. We discuss why this condition is necessary for equisynthesizability.

The specification $E \Rightarrow S$ can be realizable when $E \stackrel{+}{\triangleright} S$ is unrealizable, despite $Cl(S) \stackrel{+}{\triangleright} E$ being realizable by the environment (an environment that does not realize E). Such an environment can give the component an opportunity

[6] "Equirealizable" in [96] is called equisynthesizable here, and equirealizable here is a weaker notion than there.

[7] We work with $\stackrel{+}{\triangleright}$ but the same principles apply to *WhilePlusHalf*, due to similar syntactic properties with respect to closure.

[8] Klein and Pnueli define well-separation as a property of the environment E [96]. Relative well-separation is a property of the pair of properties E, S, not of the environment alone.



These realizations make a difference only when we assemble
the component with an environment that does not realize $Cl(S) \overset{+}{\triangleright} E$.

Figure 9.4: Comparing the collections of realizations of $E \Rightarrow S$ and $E \overset{+}{\triangleright} S$.

to realize[9]

$$(E \Rightarrow S) \wedge \neg(E \overset{+}{\triangleright} S).$$

The environment could prevent the component from realizing $E \overset{+}{\triangleright} S$, even though $E \Rightarrow S$ remains realizable.

Remark 13. If the closures prevent the assembly from reaching a state from where the component can predict that it can force a future violation of E, then the properties $E \Rightarrow M$ and $E \overset{+}{\triangleright} M$ become equisynthesizable. This is relevant to an observation about implementations [7, Sec. 5.1 on p. 528]. □

The puzzling point then is how $E \Rightarrow S$ can remain realizable when $E \overset{+}{\triangleright} S$ is not, and although $Cl(S) \overset{+}{\triangleright} E$ is realizable by the environment. The answer is that an environment that realizes $Cl(S) \overset{+}{\triangleright} E$ does not ensure E (a stronger property). Such an environment *could* let the component violate *SysNext* before *EnvNext* in its path to implementing $E \Rightarrow S$ (and this outcome *does* satisfy $Cl(S) \overset{+}{\triangleright} E$).

This is why *relative* well-separation requires more than environment realizability [131]. In particular, well-separation requires that the assembly cannot safely reach states from where the component can realize $E \Rightarrow S$ by violating $E \overset{+}{\triangleright} S$. When this is the case, composing a component that implements $E \Rightarrow S$ with an environment that implements $Cl(S) \overset{+}{\triangleright} E$ *does* keep the component away from states where it can benefit from violating the action

[9] The property $(E \Rightarrow S) \wedge \neg(E \overset{+}{\triangleright} S)$ is equivalent to $(E \Rightarrow S) \wedge \neg(Cl(E) \overset{+}{\triangleright} Cl(S))$.

SysNext before the action *EnvNext* is violated. An even stronger requirement is well-separation, which requires realizability of E instead of $Cl(S) \stackrel{+}{\triangleright} E$, i.e., irrespective of the component's specification.

Klein and Pnueli observe that well-separation is stronger than requiring a realizable environment, and that realizability is too weak to ensure equivalence of $\Rightarrow$ and *RawWhile* [96, footnote on p. 171] Indeed, realizability of $Cl(S) \stackrel{+}{\triangleright} E$ by the environment does not suffice to ensure equirealizability, and well-separation requires realizability of E (unconditional).

9.3 *WhilePlusHalf*

The property $Env \stackrel{+}{\triangleright} Sys$ is unrealizable if the action *SysNext* constrains primed environment variables (x') [66], which makes writing realizable specifications less straightforward.[10] Relaxing *Sys* to leave environment variables unconstrained can avoid this situation. This relaxation though raises issues of underspecification with respect to the axioms of TLA$^+$. We want to define open-system properties that are realizable when *Env* and *Sys* suggest so, and also expressible with some binary operator that resembles implication.

Motivated by these observations, we define the operator *WhilePlusHalf* below, which is realizable when expected from thinking in terms of *Env* and *Sys*. *WhilePlusHalf* is obtained as a modification of *WhilePlus* (the syntactic definition of $\stackrel{+}{\triangleright}$). We also prove that *WhilePlusHalf* is equivalent to the operator *RawWhilePlusHalf* in raw TLA$^+$, which does not involve temporal quantification and is suitable for fixpoint computations [153].

The value of machine-unclosed representations We may want to write a specification that lacks machine closure for a layer higher than the implementation, in order to write a more readable specification [1, p. 142], [117, p. 203], [53]. The operator *WhilePlusHalf* allows writing a machine-unclosed representation of *Sys* that constrains x'. Otherwise, we would have to find a machine-closed representation that leaves x' unconstrained, which involves reasoning similar to synthesis.

Bounded temporal formulas Boundedness is useful in avoiding dependence on underspecified parts of TLA$^+$ (and several operators are specified

[10] Besides *SysNext*, a liveness subformula too can impose constraints on x'.

only within a set of values, not for arbitrary values). Deducing properties of unbounded formulas can be harder than for bounded formulas.[11] For example, whether an unbounded specification allows the component to eschew the intended liveness formula (in *Sys*) by letting x' take arbitrary values along the tail of some behavior that is a witness for temporal quantification ($\exists$) within *WhilePlus*. If so, then the system could "pretend" unrealistic environment behavior in the witness behavior's tail in order to satisfy $\Box[SysNext]_v \wedge SysLiveness$, while forcing $\Diamond\langle \neg EnvNext \rangle_v$) in the actual behavior.

9.3.1 Modifying the *WhilePlus* operator

The operator *WhilePlus*, defined below within TLA$^+$, is equivalent to the operator $\overset{+}{\triangleright}$ [117, p. 337] (which is defined semantically [117, p. 316]).

$$
\begin{aligned}
&MayUnstep(b) \triangleq \ \wedge\, b \in \text{BOOLEAN} \\
&\qquad\qquad\qquad\ \wedge\, \Box[b' = \text{FALSE}]_b \\
&Unstep(b) \triangleq \ \wedge\, MayUnstep(b) \\
&\qquad\qquad\ \wedge\, \Diamond(b = \text{FALSE}) \\
&SamePrefix(b,\, u,\, v,\, x,\, y) \triangleq \ \Box(b \Rightarrow (\langle u,\, v\rangle = \langle x,\, y\rangle)) \\
&PlusHalf(b,\, v,\, y) \triangleq \ \wedge\, v = y \\
&\qquad\qquad\qquad\ \wedge\, \Box[b \Rightarrow (v' = y')]_{\langle b,\, v,\, y\rangle} \\
&Front(P(\_,\, \_),\, x,\, y,\, b) \triangleq \\
&\quad \exists\, u,\, v : \\
&\qquad \wedge\, P(u,\, v) \\
&\qquad \wedge\, SamePrefix(b,\, u,\, v,\, x,\, y) \\
&FrontPlusHalf(P(\_,\, \_),\, x,\, y,\, b) \triangleq \\
&\quad \exists\, u,\, v : \\
&\qquad \wedge\, P(u,\, v) \\
&\qquad \wedge\, SamePrefix(b,\, u,\, v,\, x,\, y) \\
&\qquad \wedge\, PlusHalf(b,\, v,\, y) \\
&FrontPlus(P(\_,\, \_),\, x,\, y,\, b) \triangleq \exists\, u,\, v : \\
&\quad \text{LET} \\
&\qquad vars \triangleq \langle b,\, x,\, y,\, u,\, v\rangle
\end{aligned}
$$

[11] Boundedness relates to whether a formula's meaning depends on what model of a theory we use [100, pp. 161–163], and to decidability [100, p. 168 and §IV. 3].

$Init \triangleq \langle u, v\rangle = \langle x, y\rangle$

$Next \triangleq b \Rightarrow (\langle u', v'\rangle = \langle x', y'\rangle)$

$Plus \triangleq \Box[Next]_{vars}$

IN

$\land P(u, v)$

$\land Init \land Plus$

$WhilePlus(A(\_, \_), G(\_, \_), x, y) \triangleq$

$\forall b : (MayUnstep(b) \land Front(A, x, y, b)) \Rightarrow FrontPlus(G, x, y, b)$

We propose the following operator, which differs from *WhilePlus* in that within the consequent primed environment variables are not constrained (and initial component variable values are unconditionally constrained).

$WhilePlusHalf(A(\_, \_), G(\_, \_), x, y) \triangleq$

$\forall b : (MayUnstep(b) \land Front(A, x, y, b)) \Rightarrow FrontPlusHalf(G, x, y, b)$

$WPH(A, G, x, y) \triangleq WhilePlusHalf(A, G, x, y)$ a shorthand

9.3.2 Stepwise form of *WhilePlusHalf*

Properties written using *WhilePlusHalf* are expressible as the conjunction of a "stepwise" safety property and an implication (theorem *WhilePlusHalfAsConj* below), similarly to the tautology that holds for $\stackrel{+}{\rightarrow}$. This form is useful for expressing *WhilePlusHalf* in raw TLA$^+$ with past operators (theorem *WhilePlusHalfStepwiseForm* below), similarly to the connection between $\stackrel{+}{\rightarrow}$ and *RawWhilePlus* (Section 9.1).[12]

THEOREM $WhilePlusHalfAsConj \triangleq$

ASSUME

VARIABLE x, VARIABLE y,

TEMPORAL $A(\_, \_)$, TEMPORAL $G(\_, \_)$

PROVE

LET

$ClA(u, v) \triangleq Cl(A, u, v)$

$ClG(u, v) \triangleq Cl(G, u, v)$

IN

$WPH(A, G, x, y) \equiv \land WPH(ClA, ClG, x, y)$

$\land A(x, y) \Rightarrow G(x, y)$

[12] The theorems *WhilePlusHalfAsConj* and *WhilePlusHalfStepwiseForm* are proved in the module *WhilePlusHalfTheorems*.

$$
\begin{array}{l}
RawWhilePlusHalfFull(\\
\quad IeP(\_,\_),\ JeP(\_,\_),\ IsP(\_,\_),\\
\quad EnvNext,\ Next,\ SysNext,\ Le,\ Ls) \triangleq\\
\quad \lor \neg\exists\, p,\ q :\ IeP(p,\ q) \Rightarrow JeP(p,\ q)\\
\quad \lor\ \land \exists\, p :\ IsP(p,\ y)\\
\qquad \land\ \lor\ \neg\ \lor\ \neg IeP(x,\ y)\\
\qquad\qquad\ \lor JeP(x,\ y)\\
\qquad\quad \lor\ \land IsP(x,\ y)\\
\qquad\qquad \land IeP(x,\ y) \lor \Box(Next \land SysNext)\\
\qquad\qquad \land\ \lor \neg IeP(x,\ y)\\
\qquad\qquad\quad \lor \Box(Earlier(EnvNext) \Rightarrow\ \land Earlier(Next)\\
\qquad\qquad\qquad\qquad\qquad\qquad \land SysNext)\\
\qquad\qquad \land\ \lor\ \neg\ \lor\ \neg IeP(x,\ y)\\
\qquad\qquad\qquad\ \lor JeP(x,\ y) \land Le \land \Box EnvNext\\
\qquad\qquad\quad \lor Ls
\end{array}
$$

$$
\begin{array}{l}
\textsc{theorem}\ \ WhilePlusHalfStepwiseForm \triangleq\\
\quad \textsc{assume}\\
\qquad \textsc{variable}\ x,\ \textsc{variable}\ y,\\
\qquad \textsc{new}\ sigma,\ IsABehavior(sigma),\\
\qquad \textsc{constant}\ IeP(\_,\_),\\
\qquad \textsc{constant}\ JeP(\_,\_),\\
\qquad \textsc{constant}\ IsP(\_,\_),\\
\qquad \textsc{constant}\ NeP(\_,\_,\_,\_),\\
\qquad \textsc{constant}\ NsP(\_,\_,\_,\_),\\
\qquad \textsc{temporal}\ LeP,\ \textsc{temporal}\ LsP,\\
\qquad \land \forall\, u,\ v :\ IeP(u,\ v) \in \textsc{boolean}\\
\qquad \land \forall\, u,\ v :\ JeP(u,\ v) \in \textsc{boolean}\\
\qquad \land \forall\, u,\ v :\ IsP(u,\ v) \in \textsc{boolean}\\
\qquad \land \forall\, a,\ b,\ c,\ d :\ NeP(a,\ b,\ c,\ d) \in \textsc{boolean}\\
\qquad \land \forall\, a,\ b,\ c,\ d :\ NsP(a,\ b,\ c,\ d) \in \textsc{boolean}\ ,\\
\qquad \textsc{let}\\
\qquad\quad xy \triangleq \langle x,\ y\rangle\\
\qquad\quad Is \triangleq IsP(x,\ y)
\end{array}
$$

$Ie \triangleq IeP(x,\ y)$

$Je \triangleq JeP(x,\ y)$

$Ne \triangleq NeP(x,\ y,\ x',\ y')$

$Ns \triangleq NsP(x,\ y,\ x',\ y')$

$Le \triangleq LeP(x,\ y)$

$Ls \triangleq LsP(x,\ y)$

$A(u,\ v) \triangleq$

LET

$I \triangleq IeP(u,\ v)$

$J \triangleq JeP(u,\ v)$

$N \triangleq NeP(u,\ v,\ u',\ v')$

$vrs = \langle u,\ v \rangle$

$L \triangleq LeP(u,\ v)$

IN

$I \Rightarrow (J \land \Box[N]_{vrs} \land L)$

$Q(u,\ v) \triangleq$

$\lor \neg IeP(u,\ v)$

$\lor \land JeP(u,\ v)$

$\land \Box[NeP(u,\ v,\ u',\ v')]_{\langle u,\ v \rangle}$

$R(u,\ v) \triangleq \land IsP(u,\ v)$

$\land \Box[NsP(u,\ v,\ u',\ v')]_{\langle u,\ v \rangle}$

IN

$\land \forall\, u,\ v :\ Cl(A,\ u,\ v) \equiv Q(u,\ v)$

IsMachineClosed adapted by replacing x with arguments.

$\land \forall\, u,\ v :\ IsMachineClosed(R,\ LsP,\ u,\ v)$

PROVE

LET

$A(u,\ v) \triangleq$

LET

$I \triangleq IeP(u,\ v)$

$J \triangleq JeP(u,\ v)$

$N \triangleq NeP(u,\ v,\ u',\ v')$

$vrs = \langle u,\ v \rangle$

$$
\begin{array}{l}
\quad\quad L \triangleq LeP(u,\ v) \\
\quad \text{IN} \\
\quad\quad I \Rightarrow (J \land \Box[N]_{vrs} \land L) \\
G(u,\ v) \triangleq \\
\quad \text{LET} \\
\quad\quad I \triangleq IsP(u,\ v) \\
\quad\quad N \triangleq NsP(u,\ v,\ u',\ v') \\
\quad\quad vrs \triangleq \langle u,\ v \rangle \\
\quad\quad L \triangleq LsP(u,\ v) \\
\quad \text{IN} \\
\quad\quad I \land \Box[N]_{vrs} \land L \\
Phi \triangleq WhilePlusHalf(A,\ G,\ x,\ y) \\
\\
xy \triangleq \langle x,\ y \rangle \\
Ie \triangleq IeP(x,\ y) \\
Is \triangleq IsP(x,\ y) \\
Ne \triangleq NeP(x,\ y,\ x',\ y') \\
Ns \triangleq NsP(x,\ y,\ x',\ y') \\
Le \triangleq LeP(x,\ y) \\
Ls \triangleq LsP(x,\ y) \\
\\
EnvNext \triangleq [Ne]_{\langle x,\ y \rangle} \\
Next \triangleq [Ns]_{\langle x,\ y \rangle} \\
SysNext \triangleq [\exists\, r :\ NsP(x,\ y,\ r,\ y')]_y \\
\\
RawPhi \triangleq RawWhilePlusHalfFull(\\
\quad\quad IeP,\ JeP,\ IsP,\ EnvNext,\ SysNext,\ Le,\ Ls) \\
\text{IN} \\
\quad (sigma,\ 0 \models RawPhi) \equiv (sigma \models Phi)
\end{array}
$$

9.3.3 Closing an assembly: Should we constrain the component's initial state?

We compare two definitions of realizability: with and without choice of initial values of component variables.

$$
\begin{array}{l}
Realization(x,\ y,\ mem,\ f,\ g,\ y0,\ mem0,\ e) \triangleq \\
\quad \text{LET}
\end{array}
$$

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

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

More properties are realizable with *Realization* than with *RealizationOld*, because *Realization* lets the component choose the initial value of variable y. Properties formed using *WhilePlusHalf* can be realizable with *Realization* but unrealizable with *RealizationOld*. The reason is that *RealizationOld* leaves the initial value of y unconstrained, so the specification property should not constrain the initial state more than the assumption does.

If we specify two components A_1 and A_2 that leave the initial state unconstrained, then the assembly $A_1 \land A_2$ cannot imply any nontrivial initial condition. So we cannot assemble a closed-system from open-systems, unless each component constrains some part of the initial state.

This may sound reasonable, because any real system interacts with the rest of the world. However, after assembling all components relevant to the variables declared in a given specification, we expect to obtain a closed-system that models reality.

We can obtain a closed-system assembly by specifying each component with the operator *WhilePlusHalf* and an initial condition in the second argument (guarantee). By using *Realization*, instead of *RealizationOld*, we avoid unrealizability of such properties.

9.3.4 Machine-closed representation of open-systems

Abadi and Lamport note the tautology[13]

$$A \xrightarrow{+} G \;\equiv\; \wedge\, Cl(A) \xrightarrow{+} Cl(G)$$
$$\wedge\, A \Rightarrow G$$

The same tautology holds for the operator *WhilePlusHalf*.[14]

Interestingly, this particular conjunction is a machine-closed representation of the property $A \xrightarrow{+} G$. In other words the following theorem is provable.

THEOREM *WhilePlusSafetyLivenessDecomp* $\triangleq$
ASSUME
TEMPORAL A, TEMPORAL G
PROVE
$$\wedge\, SafetyPart(A \xrightarrow{+} G) \;\equiv\; Cl(A) \xrightarrow{+} Cl(G)$$
$$\wedge\, LivenessPart(A \xrightarrow{+} G)$$
$$\equiv (Cl(A) \xrightarrow{+} Cl(G)) \;\Rightarrow\; (A \xrightarrow{+} G)$$

The reason for this decomposition is that the stepwise conjunct holds if either eventually $Cl(A)$ is violated, thus also A (so $A \Rightarrow G$), or both $Cl(A)$ and $Cl(G)$ are satisfied, which implies that the liveness goals of both A and G remain reachable throughout the behavior, thus $A \Rightarrow G$ does not place any safety constrain on the stepwise part. A similar result can be shown about the operator *WhilePlusHalf*.

9.4 Forming open from closed systems

9.4.1 Trapped between over and under specification

9.4.1.1 Overspecification

Consider the specification

VARIABLES a, b the component controls b

$$A \triangleq \wedge\, a = 1$$
$$\wedge\, \Box(a \in 1\,..\,2)$$
$$\wedge\, \Box\Diamond(a = 2)$$

$$B \triangleq \wedge\, \Box(b \in 1\,..\,2)$$
$$\wedge\, \Box[b' = a]_{\langle a,\, b\rangle}$$

[13] The module *TemporalLogic* contains a proof of this tautology, and of the theorem *WhilePlusSafetyLivenessDecomp*.
[14] As proved in the module *WhilePlusHalfTheorems*).

$$\wedge\, \Box\Diamond(b = 2)$$

A first inspection suggests that the assumption A should suffice to ensure realizability of the guarantee B.[15] To put it differently, we want an operator that forms a realizable open-system from the pair A, B. The resulting open-system should mean what we expect. What do we expect?

Conjoining the open-system with the intended environment should imply the guarantee.[16]

$$(\mathit{OpenSystem} \wedge \mathit{Env}) \Rightarrow B$$

Let $\mathit{Env} \triangleq A$. The above implication is valid with the definition $\mathit{OpenSystem} \triangleq A \stackrel{+}{\triangleright} B$ because[17] $(A \wedge (A \stackrel{+}{\triangleright} B)) \Rightarrow B$ so the implication $(\mathit{OpenSystem} \wedge A) \Rightarrow B$ is valid.

Is the property *OpenSystem* realizable?[18] No, due to a counter-example where an arbitrary value of a' blocks the component from satisfying $Cl(B)$ in the safety part $Cl(A) \stackrel{+}{\triangleright} Cl(B)$ of *OpenSystem* [66, Prop. 7].

Briefly, $A \stackrel{+}{\triangleright} B$ is unrealizable because the finite behavior

$$\langle \underbrace{\begin{bmatrix} a \mapsto 1 \\ b \mapsto 2 \\ \vdots \end{bmatrix}}_{s_1} \quad \underbrace{\begin{bmatrix} a \mapsto 20 \\ b \mapsto * \\ \vdots \end{bmatrix}}_{s_2} \rangle$$

satisfies A up to s_1, so in any implementation the variable b should take a value $s_2.b$ such that $\langle s_1, s_2 \rangle$ be extendable to a behavior that satisfies B. This extension is impossible.[19]

THEOREM *NotExtensible* $\triangleq$

ASSUME

$$\begin{aligned} \exists\, \mathit{tau} : \quad & \wedge\, \mathit{IsABehavior}(\mathit{tau}) \\ & \wedge\, \mathit{tau} \models B \\ & \wedge\, \mathit{tau}[0].a = 1 \\ & \wedge\, \mathit{tau}[1].a = 20 \\ & \wedge\, \mathit{tau}[0].b = 2 \end{aligned}$$

[15] With realizability defined so that the component pick the initial value of variable b.
[16] This is a reasonable sanity check for any operator for defining open-systems.
[17] By the proposition *StepwiseAntecedent* in the module *TemporalLogic*.
[18] In the context of a reasonable definition of realizability.
[19] A proof of the theorem *NotExtensible* is in the module *UnzipTheorems*.

PROVE FALSE

Unrealizability above arises due to a' taking an arbitrary value that blocks the component, despite A prohibiting that value.[20] Unless the value a' is constrained within *Realization*, unrealizability is remains for the property $A \stackrel{+}{\triangleright} B$. Incorporating such an assumption within *Realization* splits the assumption into two parts (the operator A and a subformula within *Realization*). Using the operator *WhilePlusHalf* instead of $\stackrel{+}{\triangleright}$ can avoid this situation.

What we observed above is an instance of overspecification. We overspecified the component by constraining in B the next environment state (a'), which is unconstrained by $A \stackrel{+}{\triangleright} B$, regardless of what the assumption A is. Removing the constraint on a' from B avoids this situation, but raises another issue as discussed below.

9.4.1.2 Underspecification

Suppose that we modify the guarantee to

$$\begin{aligned} Bnew \triangleq\ & \wedge\, b = 1 \\ & \wedge\, \Box(b \in 1\,..\,2) \\ & \wedge\, \Box[(a \ \in 1\,..\,2) \Rightarrow (b' = a)]_{\langle a,\, b\rangle} \\ & \wedge\, \Box\Diamond(b = 2) \end{aligned}$$

That a part of the assumption now occurs within the guarantee indicates that separation into an assumption and guarantee passes through reasoning about the assembled system. In Section 9.4.2 we show how to specify the assembled system (closed) and transform that into an open-system specification.

Reconsidering the behavior that we discussed above, which starts with (was $b = 2$ at the initial state, but that difference is irrelevant)

$$\begin{bmatrix} a \mapsto 1 \\ b \mapsto 1 \end{bmatrix} \begin{bmatrix} a \mapsto 20 \\ b \mapsto * \end{bmatrix}$$

we can now extend it to satisfy *Bnew*, as follows

$$\tau \triangleq \begin{bmatrix} a \mapsto 1 \\ b \mapsto 1 \end{bmatrix} \begin{bmatrix} a \mapsto 20 \\ b \mapsto 1 \end{bmatrix} \begin{bmatrix} a \mapsto 1 \\ b \mapsto 2 \end{bmatrix} \cdots \text{stuttering tail}$$

[20] The same problem with $\stackrel{+}{\triangleright}$ is present in typed formalisms too; our observation is not an artifact of untypeness.

So the property $A \stackrel{+}{\triangleright} Bnew$ is realizable, in constrast to $A \stackrel{+}{\triangleright} B$. The modified property $Bnew$ has some unexpected properties. For instance, another witness behavior is

$$tau \triangleq \begin{bmatrix} a \mapsto 1 \\ b \mapsto 1 \end{bmatrix} \begin{bmatrix} a \mapsto 20 \\ b \mapsto 1 \end{bmatrix} \underbrace{\begin{bmatrix} a \mapsto \{1,3\} \\ b \mapsto 2 \end{bmatrix} \begin{bmatrix} a \mapsto Nat \\ b \mapsto 2 \end{bmatrix} \begin{bmatrix} a \mapsto \sqrt{2} \\ b \mapsto 2 \end{bmatrix} \ldots}_{\text{arbitrary values of } a}$$

The suffix of τ can assign arbitrary values to variable a, even if a satisfies A up to the second state, i.e., $PrefixSat(sigma, 2, A)$. A behavior that demonstrates this is the following

$$tau \triangleq \begin{bmatrix} a \mapsto 1 \\ b \mapsto 1 \end{bmatrix} \begin{bmatrix} a \mapsto 1 \\ b \mapsto 1 \end{bmatrix} \begin{bmatrix} a \mapsto \{1,3\} \\ b \mapsto 1 \end{bmatrix} \begin{bmatrix} a \mapsto Nat \\ b \mapsto 2 \end{bmatrix} \begin{bmatrix} a \mapsto \sqrt{2}^{Nat} \\ b \mapsto 2 \end{bmatrix} \ldots$$

(Variable b cannot take arbitrary values, because the definition of $Bnew$ includes the conjunct $\Box(b \in 1..2)$).

This behavior is a witness even for B, given this particular (2-state) prefix that satisfies the assumption. The arbitrariness of values assigned to variable a would better be avoided, as discussed below.

9.4.1.3 Practical considerations

One can argue that, in general, we can formulate a specification so as to allow arbitrariness of values within the suffix of the witness behavior, without compromises to what we can specify.

However, there are other factors to consider too. In order to mechanize the computation of a closure, we have to eliminate a temporal existential quantifier ($\exists\!\!\!\exists$). This elimination typically involves some reachability analysis (unless we are given a safety property, or a property as the conjunction of a machine-closed pair of a safety and a liveness property). Computing the closure is necessary for mechanized reasoning about the operator $\stackrel{+}{\triangleright}$, whenever any of its first two arguments is given as a conjunction of two properties that do not form a machine-closed pair (Section 9.1).

Semantic methods (BDDs, enumeration, etc.) are more automated than syntactic methods (theorem proving) so we assume that a semantic method is used to compute the closure of G when given as a machine-unclosed pair of formulas. In order to compute the closure $Cl(G)$, a semantic method needs

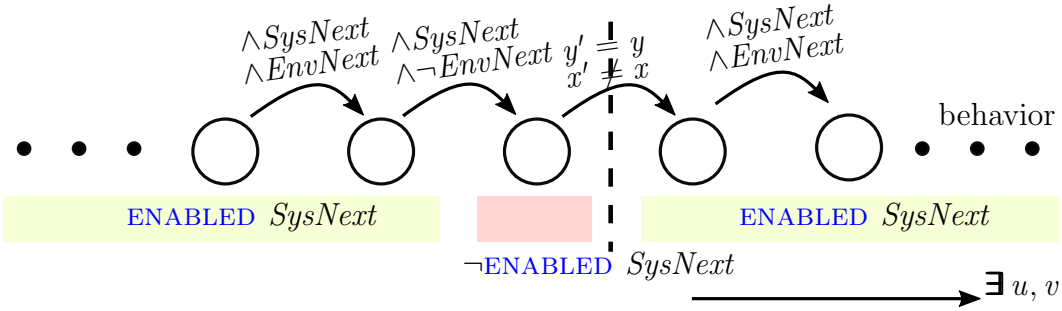


Figure 9.5: How $\Box[SysNext]_y$ within $\stackrel{+}{\triangleright}$ allows the component to restore enabledness within the tail of the behavior that witnesses temporal quantification $\exists$, even after environment glitches, but at the cost of underspecification. (For example, by allowing arbitrary environment glitches to occur in the tail of the witness behavior. The component can exploit this to exhibit undesired behavior in the unhidden part of the witness behavior.)

to interpret all the variables that occur in G (with the exception of G written in suitable syntactic form [7], and using a syntactic method). So G should be a bounded formula (Δ_0 [100]), i.e., a closed-system (with respect to relevant variables). Thus, a property G that leaves the environment variable a unconstrained is inconvenient for computing the closure mechanically.

One way to relax a property of the form $A \stackrel{+}{\triangleright} G$ in order to make it realizable is to by allowing the component to stutter y

$$G \triangleq Init \wedge \Box[N]_y \wedge L$$

This stuttering subscript allows A-steps, with

$$A \triangleq (x' \neq x) \wedge (y' = y),$$

from states where $\neg$ENABLED N back to states where ENABLED N. This specification style alows the component to restore the environment to a helpful state after glitches, as shown in Fig. 9.5. However, it also allows "wild" behavior of x within $\exists$. The possibility of wild behavior is relevant because the system could exploit it to prevent the environment from satisfying its liveness specification, or its safety specification.

9.4.2 The *Unzip* operator

Inventing an assumption and a guarantee that fit together as the specifier intends is an error-prone and time consuming process. Even worse, many times the resulting specification means something different than the specifier wants, in subtle ways. Recognizing mistakes in open-system specifications

is more difficult than reasoning about closed-system specifications. For this reason, we propose an approach for transforming a closed-system specification into an open-system specification.

The *Unzip* operator (defined in Section 4.2.2) can be expressed in raw TLA$^+$ with past operators as follows[21]

THEOREM
  ASSUME
    VARIABLE x, VARIABLE y,
    CONSTANT $I(\_, \_)$,
    CONSTANT $N(\_, \_)$,
    TEMPORAL $L(\_, \_)$
  PROVE
    LET
      $P(u, v) \triangleq \land I(u, v) \land L(u, v)$
                  $\land \Box[N(u, v, u', v')]_{\langle u, v\rangle}$
      $EnvNext \triangleq [\exists\, r :\ N(x, y, x', r)]_x$
      $SysNext \triangleq [\exists\, r :\ N(x, y, r, y')]_y$
      $Next \triangleq [N(x, y, x', y')]_{\langle x, y\rangle}$
      $Raw \triangleq$
          $\land \exists\, p :\ I(p, y)$
          $\land\ \lor \neg\exists\, q :\ I(x, q)$
             $\lor\ \land I(x, y)$
                $\land \Box \lor \neg Earlier(EnvNext)$
                     $\lor SysNext \land Earlier(Next)$
                $\land (\Box EnvNext) \Rightarrow L(x, y)$
    IN
      $Unzip(P, x, y) \equiv Raw$

The proof applies twice the transformation of *WhilePlusHalf* to stepwise form within PastRTLA$^+$.

Practical benefits TLA aids the specifier via syntactic constructs that ensure intended properties (stutter-invariance via $[A]_v$, machine-closure via WF and SF). The *Unzip* operator aligns with this approach (e.g., it prevents the component from leading the environment to a deadend). For example,

[21] A proof is in the module *WhilePlusHalfTheorems*.

$\mathit{Unzip}(\text{FALSE}, \ldots) \equiv \text{FALSE}$ (so unrealizable), which prevents accidental "vacuity".

TLA helps the specifier also by reducing reasoning about temporal formulas to reasoning about actions, when possible [112, author's abstract, Sec. 6 on p. 20], [114, Sec. 6.1.3 on p. 891], [11, Sec. 4.1 on p. 253]. In this direction of simplifying reasoning, *Unzip* reduces writing the specification of an open-system to writing the specification of a closed-system. For example, we need not decide how to split the specification into two arguments (of course, we still need to include sufficient constraints).

Remark 14. $\mathit{Unzip}(P)$ avoids trivial realizability, because there is no way to block the environment. That $\mathit{Cl}(P)$ implies the assumption, together with the fact that we are weakening $\exists y' : \mathit{Next}$, ensure that the component cannot block the environment. If P is satisfiable, then within *Unzip*, the argument $A(u, v)$ to *WhilePlusHalf* (Section 4.2.2) is realizable. (This also addresses the question of whether satisfiability or realizability should be used for defining vacuity.)

$$
\begin{array}{l}
(\exists\, u :\ \mathit{Next}(x,\ y,\ u,\ y')) \Rightarrow \\
(\exists\, v :\ \exists\, u \quad\ :\ \mathit{Next}(x,\ y,\ u,\ v)) \Rightarrow \\
(\exists\, u :\ \exists\, v \quad\ :\ \mathit{Next}(x,\ y,\ u,\ v)) \Rightarrow \\
\text{ENABLED}\ \exists\, v :\ \mathit{Next}(x,\ y,\ x',\ v)\ \Rightarrow \\
\text{ENABLED}\ \mathit{EnvNext}(x,\ y,\ x')
\end{array}
$$

So the component cannot force a behavior where $\Diamond\langle \neg \mathit{EnvNext}(x, y, x')\rangle_{\langle x,y\rangle}$, without first disabling *SysNext*. In other words, the system cannot block the assembly before the environment does. □

Regarding closure of the assumption The operator *Unzip* is defined using as assumption the property $A \triangleq \mathit{WPH}(P, P, y, x)$, which is a safety property, because, with $C(u, v) \triangleq \mathit{Cl}(P, u, v)$

$$A \equiv \mathit{WPH}(C, C, y, x) \wedge (P \Rightarrow P) \equiv \mathit{WPH}(C, C, y, x) \equiv \mathit{Cl}(A).$$

Thus, we need not compute a closure for the assumption A.

9.5 System factorization

9.5.1 Inessential noninterleaving

In developing the algorithm for hiding variables in Section 5.3, we used the operator *Step* (defined in Section 3.4). In Section 5.2 we applied existential

quantification to define the component actions *SysNext* and *EnvNext* from the action *WNext* of the closure W of the property *Assembly* that describes the assembled system. For simplicity, in this section we let *Next* stand for *WNext*, and *Assembly* for W (equivalently, we assume that *Assembly* is expressed with a machine-closed representation).

The operator *Step* corresponds to realizability of properties defined using *WhilePlus*, and the operator *GeneralStep* (defined in Section 3.4) to realizability of properties defined using *WhilePlusHalf*. This correspondence becomes evident by the stepwise forms of these operators, derived in Sections 9.3.2 and 9.4.2.

As we remarked in Section 5.2, the *Step* operator with the component actions *SysNext* and *EnvNext* corresponds to applying *Unzip* to the property *Assembly*. We justified this observation in Section 5.2 by assuming that only limited noninterleaving is allowed for the environment. In particular, we assumed that the environment can change in only a unique way in steps that the component changes, with a deterministic scheduler as an example. We now prove this claim, as follows. Let

$$\begin{aligned} ExistsUnique(P(\_)) \triangleq\ & \land \exists\, u :\ P(u) \\ & \land \forall\, u,\, v :\ (P(u) \land P(v)) \ \Rightarrow\ (u = v) \end{aligned}$$

The following theorem expresses that, uder the assumption that the environment changes in a unique way in system turns, then if both of the actions *SysNext* and *EnvNext* hold, so does the action *Next*.[22]

THEOREM *InessentialNoninterleaving* $\triangleq$

ASSUME

VARIABLE x, VARIABLE y, CONSTANT $Inv(\_, \_)$,

CONSTANT $SysTurn(\_, \_)$, CONSTANT $Next(\_, \_, \_, \_)$,

LET $SysNext(p,\, q,\, v) \triangleq \exists\, u :\ Next(p,\, q,\, u,\, v)$

$EnvNext(p,\, q,\, u) \triangleq \exists\, v :\ Next(p,\, q,\, u,\, v)$

IN

$\land\ \lor \neg(SysTurn(x,\, y) \land Inv(x,\, y))$

$\lor\ ExistsUnique(\text{LAMBDA}\ r :\ EnvNext(x,\, y,\, r))$

$\land\ SysNext(x,\, y,\, y') \land EnvNext(x,\, y,\, x')$

PROVE

[22] The module *Unzip* contains a proof of the theorem *InessentialNoninterleaving* and of other theorems in this section.

$$\lor \neg \land SysTurn(x,\ y)$$
$$\quad \land Inv(x,\ y)$$
$$\lor Next(x,\ y,\ x',\ y')$$

9.5.2 General case of noninterleaving specifications

We now address the case when the property *Assembly* is an arbitrary noninterleaving specification. Specifically, we show that a component defined as $Unzip(Assembly, x, y)$, which would require using the *GeneralStep* operator, has the same implementations as a component specified by $Unzip(New)$, where the operator *Step* can be used for the property *New*, instead of the operator *GeneralStep*. This result allows applying the decomposition results of Chapter 6 to noninterleaving assembly specifications, provided that the assembly specification is first transformed from *Assembly* to *New*.

This transformation removes what we call the "non-Cartesian" part from the action *Next* of *Assembly*, because this part is unrealizable by a Moore component (due to constraints between x' and y').

The starting point for noninterleaving specifications is

THEOREM *CPreSimplerByConjunctivity* $\triangleq$
ASSUME
NEW *Next*, NEW *SysNext*, NEW *EnvNext*, NEW *Target*,
$Next \equiv (SysNext \land EnvNext)$ Conjunctivity
PROVE
$(\land SysNext$
$\land EnvNext \Rightarrow Target)$
$\equiv$
$(\land SysNext$
$\land EnvNext \Rightarrow \land Next$
$\land Target)$

Therefore, whenever *Next* is the conjunction of *SysNext* and *EnvNext*, we can replace the *GeneralStep* operator with the *Step* operator. When do the actions *SysNext*, *EnvNext* satisfy the above assumption?

For arbitrary *SysNext*, *EnvNext* there is not much we can say, unless

$$Next(x, y, x', y') \triangleq SysNext(x, y, y') \land EnvNext(x, y, x')$$

To see why, consider the case that we are interested in

$$SysNext(p, q, v) \triangleq \exists u : SomeNext(p, q, u, v).$$

By the definition of *Next*,

$$(\exists u : Next(x, y, u, v)) \equiv (SysNext(x, y, v) \wedge \exists u : EnvNext(x, y, u)).$$

So it is not necessarily the case that $SysNext(x, y, v) \equiv \exists u : Next(x, y, u, v)$. For this reason, we focus on the case that we are interested in:

$SysNext(p,\ q,\ u,\ v) \triangleq \exists\, u :\ Next(p,\ q,\ u,\ v)$

$EnvNext(p,\ q,\ u,\ v) \triangleq \exists\, v :\ Next(p,\ q,\ u,\ v)$

This case corresponds to actions that occur within *Unzip* (whereas the pair *Next*, *SysNext* occurs in any instance of *WhilePlusHalf*).

We call *Cartesian* an action *Next* that satisfies the formula

$$Next \equiv ((\exists x' : Next) \wedge (\exists y' : Next)).$$

For example, this formula is false with $Next \triangleq x' = y'$.

We show below that we can always *transform* a property $Unzip(P)$ to a property $Unzip(NewP)$, so that both properties have the same implementations. The resulting actions *EnvNext* and *SysNext* are Cartesian, and constrain x' and y', respectively.

Conjunctivity ($(EnvNext \wedge SysNext) \equiv Next$) is insufficient to ensure that *Next* is Cartesian. In addition, the actions *EnvNext* and *SysNext* need to be enabled at the same states, and depend on only x' and y', respectively.

Assume that $SysNext(x, y, y')$ and $EnvNext(x, y, x')$ (dependence only on y' and x', respectively). Then, by letting

$NewNext \triangleq SysNext \wedge EnvNext$

$$\begin{aligned}\exists\, x' :\ NewNext &\equiv \exists\, x' :\ (SysNext \wedge EnvNext)\\ &\equiv SysNext \wedge \exists\, x' :\ EnvNext\\ &\equiv SysNext \wedge \text{ENABLED}\ EnvNext\end{aligned}$$

$$\begin{aligned}\exists\, y' :\ NewNext &\equiv \exists\, y' :\ (SysNext \wedge EnvNext)\\ &\equiv EnvNext \wedge \exists\, y' :\ SysNext\\ &\equiv EnvNext \wedge \text{ENABLED}\ SysNext\end{aligned}$$

So whether $\models SysNext \equiv \exists x' : NewNext$ depends on whether $\models (\exists y' : SysNext) \Rightarrow (\exists x' : EnvNext)$. Similarly, whether $\models EnvNext \equiv \exists y' : NewNext$ depends on whether $\models (\exists x' : EnvNext) \Rightarrow (\exists y' : SysNext)$. Overall, the condition is same enabledness:

$$(\exists x' : EnvNext(x, y, x')) \equiv (\exists y' : SysNext(x, y, y'))$$

which can be expressed as

$$(\text{ENABLED } EnvNext(x, y, x')) \equiv (\text{ENABLED } SysNext(x, y, y')).$$

We arrive at the following theorem

```
THEOREM EquienablednessImpliesCartesianity ≜
  ASSUME
    VARIABLE x, VARIABLE y,
    CONSTANT EnvNext(_, _, _),
    CONSTANT SysNext(_, _, _),
    (∃ u : EnvNext(x, y, u)) ≡ ∃ v : SysNext(x, y, v)
  PROVE
    The proof goal says that NewNext is Cartesian.
    LET
      NewNext(p, q, u, v) ≜ ∧ EnvNext(x, y, u)
                            ∧ SysNext(x, y, v)
    IN
      ∧ SysNext(x, y, y') ≡ ∃ u : NewNext(x, y, u, y')
      ∧ EnvNext(x, y, x') ≡ ∃ v : NewNext(x, y, x', v)
```

Actions *EnvNext*, *SysNext* that result from *Unzip* are enabled at the same states.

```
PROPOSITION EquiEnablednessFromUnzip ≜
  ASSUME
    VARIABLE x, VARIABLE y,
    CONSTANT Next(_, _, _, _),
    CONSTANT SysNext(_, _, _),
    CONSTANT EnvNext(_, _, _),
    ∧ ∀ v : SysNext(x, y, v) ≡ ∃ u : Next(x, y, u, v)
    ∧ ∀ u : EnvNext(x, y, u) ≡ ∃ v : Next(x, y, u, v)
  PROVE
    (∃ u : EnvNext(x, y, u)) ≡ ∃ v : SysNext(x, y, v)
```

COROLLARY
ASSUME
VARIABLE x, VARIABLE y,
CONSTANT $Next(\_, \_, \_, \_)$,
CONSTANT $SysNext(\_, \_, \_)$,
CONSTANT $EnvNext(\_, \_, \_)$,
$\land \forall\, v :\ SysNext(x,\ y,\ v) \equiv \exists\, u :\ Next(x,\ y,\ u,\ v)$
$\land \forall\, u :\ EnvNext(x,\ y,\ u) \equiv \exists\, v :\ Next(x,\ y,\ u,\ v)$
PROVE
LET
$NewNext(p,\ q,\ u,\ v) \triangleq \land EnvNext(x,\ y,\ u)$
$\land SysNext(x,\ y,\ v)$
IN
$\land SysNext(x,\ y,\ y') \equiv \exists\, u :\ NewNext(x,\ y,\ u,\ y')$
$\land EnvNext(x,\ y,\ x') \equiv \exists\, v :\ NewNext(x,\ y,\ x',\ v)$

By this theorem, given two actions that constrain x', y', we need to "balance" their enabledness. Using the above results, we arrive at the following "subtraction" of unrealizable steps from the action *Next*.

COROLLARY
ASSUME
VARIABLE x, VARIABLE y,
CONSTANT $Next(\_, \_, \_, \_)$
PROVE
LET

The operators *SysNext* and *EnvNext* are already "balanced", but may not imply *Next* when conjoined. This is why we have to do the factorization as the next theorem below.

$SysNext(p,\ q,\ v) \triangleq \exists\, u :\ Next(p,\ q,\ u,\ v)$
$EnvNext(p,\ q,\ u) \triangleq \exists\, v :\ Next(p,\ q,\ u,\ v)$
$NewNext(p,\ q,\ u,\ v) \triangleq$
$\land\ SysNext(x,\ y,\ v)$
$\land\ EnvNext(x,\ y,\ u)$

NewNext is conjunctive and Cartesian, so the controllable step operator is simpler when we apply *Unzip* to a property defined using *NewNext*.

IN
$\land SysNext(x,\ y,\ y') = \exists\, u :\ NewNext(x,\ y,\ u,\ y')$
$\land EnvNext(x,\ y,\ x') = \exists\, v :\ NewNext(x,\ y,\ x',\ v)$
THEOREM $SeparatingTheRealizablePart \triangleq$

ASSUME
VARIABLE x, VARIABLE y,
CONSTANT $Next(\_, \_, \_, \_)$,
CONSTANT $Target(\_, \_)$,
CONSTANT $EnvNext(\_, \_, \_)$,
CONSTANT $SysNext(\_, \_, \_)$,
$(\text{ENABLED } SysNext(x, y, y')) \Rightarrow \text{ENABLED } EnvNext(x, y, x')$
PROVE
LET

$$NewNext(u, v) \triangleq$$
$$\land \quad SysNext(x, y, v) \land EnvNext(x, y, u)$$
$$\land \quad \forall\, w : \; EnvNext(x, y, w) \Rightarrow Next(x, y, w, v)$$

The second conjunct shrinks the first in order to ensure receptivity at those states.

$$NewSysNext(v) \triangleq \exists\, u : \; NewNext(u, v)$$
$$NewEnvNext(u) \triangleq \exists\, v : \; NewNext(u, v)$$
$$A \triangleq \exists\, v :$$
$$\land\, SysNext(x, y, v)$$
$$\land\, \forall\, u : \; EnvNext(x, y, u) \Rightarrow \; \land\, Next(x, y, u, v)$$
$$\land\, Target(u, v)$$
$$B \triangleq \exists\, v :$$
$$\land\, NewSysNext(v)$$
$$\land\, \forall\, u : \; NewEnvNext(u) \Rightarrow \; \land\, NewNext(u, v)$$
$$\land\, Target(x', v)$$
$$C \triangleq \exists\, v :$$
$$\land\, NewSysNext(v)$$
$$\land\, \forall\, u : \; NewEnvNext(u) \Rightarrow Target(u, v)$$

IN

$$\land\, NewNext(x', y') \Rightarrow Next(x, y, x', y')$$
$$\land\, A \equiv B$$
$$\land\, A \equiv C$$
$$\land\, NewNext(x', y') \equiv (NewSysNext(y') \land NewEnvNext(x'))$$

COROLLARY
ASSUME
VARIABLE p, VARIABLE q,
CONSTANT $Next(\_, \_, \_, \_)$,

$$
\begin{array}{l}
\quad \text{CONSTANT } \mathit{Target}(\_, \_) \\
\text{PROVE} \\
\quad \text{LET} \\
\qquad \mathit{SysNext}(x,\, y,\, v) \triangleq \exists\, u :\ \mathit{Next}(x,\, y,\, u,\, v) \\
\qquad \mathit{EnvNext}(x,\, y,\, u) \triangleq \exists\, v :\ \mathit{Next}(x,\, y,\, u,\, v) \\
\qquad \mathit{NewNext}(x,\, y,\, u,\, v) \triangleq \\
\qquad\quad \land \quad \mathit{SysNext}(x,\, y,\, v) \land \mathit{EnvNext}(x,\, y,\, u) \\
\qquad\quad \land \quad \forall\, w :\ \mathit{EnvNext}(x,\, y,\, w) \Rightarrow \mathit{Next}(x,\, y,\, w,\, v) \\
\qquad \mathit{NewSysNext}(x,\, y,\, v) \triangleq \exists\, u :\ \mathit{NewNext}(x,\, y,\, u,\, v) \\
\qquad \mathit{NewEnvNext}(x,\, y,\, u) \triangleq \exists\, v :\ \mathit{NewNext}(x,\, y,\, u,\, v) \\
\qquad A(x,\, y) \triangleq \exists\, v :\ \forall\, u : \\
\qquad\quad \land\, \mathit{SysNext}(x,\, y,\, v) \\
\qquad\quad \land\, \mathit{EnvNext}(x,\, y,\, u) \Rightarrow \land\, \mathit{Next}(x,\, y,\, u,\, v) \\
\qquad\qquad\qquad\qquad\qquad\quad \land\, \mathit{Target}(u,\, v) \\
\quad \text{IN} \\
\qquad \land\, \mathit{NewNext}(p,\, q,\, p',\, q') \Rightarrow \mathit{Next}(p,\, q,\, p',\, q') \\
\qquad\quad \text{Conjunctivity and Cartesianity} \\
\qquad \land\, \mathit{NewNext}(p,\, q,\, p',\, q') \\
\qquad\quad \equiv \land\, \mathit{NewSysNext}(p,\, q,\, q') \\
\qquad\qquad \land\, \mathit{NewEnvNext}(p,\, q,\, p') \\
\qquad \land\, A(p,\, q) \equiv \exists\, v :\ \forall\, u : \\
\qquad\quad \land\, \mathit{NewSysNext}(p,\, q,\, v) \\
\qquad\quad \land\, \mathit{NewEnvNext}(p,\, q,\, u) \Rightarrow \land\, \mathit{NewNext}(p,\, q,\, u,\, v) \\
\qquad\qquad\qquad\qquad\qquad\quad \land\, \mathit{Target}(u,\, v) \\
\qquad \land\, A(p,\, q) \equiv \exists\, v :\ \forall\, u : \\
\qquad\quad \land\, \mathit{NewSysNext}(p,\, q,\, v) \\
\qquad\quad \land\, \mathit{NewEnvNext}(p,\, q,\, u) \Rightarrow \mathit{Target}(u,\, v)
\end{array}
$$

Thus, we can always transform an *Unzip* property so that the generator closed-system property is Cartesian, without changing the realizable part. In other words, if $P \triangleq I \land \Box[\mathit{Next}]_{\mathit{vars}}$ is used to define an open-system using *Unzip*, then even if the action *Next* includes noninterleaving changes where variables of more than two components change in a coupled way, then the coupling is unrealizable unless Cartesian.

Table 9.1: Controllable step operators.

	Strategy	
Implication	Moore	Mealy
UpToNow	$\exists\, y' :\ \forall\, x' :$ $\quad EnvNext \Rightarrow \wedge\, SysNext$ $\qquad\qquad \wedge\, Target$	$\forall\, x' :\ \exists\, y' :$ $\quad EnvNext \Rightarrow \wedge\, SysNext$ $\qquad\qquad \wedge\, Target$
Earlier	$\exists\, y' :\ \forall\, x' :$ $\quad \wedge\, SysNext$ $\quad \wedge\, EnvNext \Rightarrow Target$	$\forall\, x' :\ \exists\, y' :$ $\quad \wedge\, SysNext$ $\quad \wedge\, EnvNext \Rightarrow Target$

By showing that the realizable part remains unchanged when we remove the non-Cartesian part from a system, we have shown that there is no loss of generality in assuming that we are given a system with Cartesian action.

In summary, conjunctivity ($Next \equiv (SysNext \wedge EnvNext)$) is useful for using the *Step* operator in place of *GeneralStep* (see the theorem *CPreSimplerBy-Conjunctivity*. A Cartesian action ($Next \equiv ((\exists x' : Next) \wedge (\exists y' : Next))$) is useful for ensuring conjunctivity in the case of open-system properties that are defined using the operator *Unzip*.

9.6 Composition without circularity

9.6.1 Stepping

Four ways of specifying how an open-system interacts with the world arise when we try to avoid circularity, shown in Table 9.1. They result as a combination of two choices:

1. Whether the component chooses y' knowing the value x' ("Mealy"), or not ("Moore").
2. How soon after the environment fails is the component allowed to fail:
   a) Can fail as soon as the environment fails
   $$\Box(UpToNow(EnvNext) \Rightarrow SysNext)$$
   b) May fail only after the environment fails
   $$\Box(Earlier(EnvNext) \Rightarrow SysNext)$$

The component controls variable y and the environment variable x. In this section we discuss using RTLA$^+$. Below we show that each combination means something different. The temporal formulas relate to each other by

$$\begin{array}{ll} \text{LET} & P \triangleq \mathit{UpToNow}(\mathit{EnvNext}) \Rightarrow \mathit{SysNext} \\ & Q \triangleq (\mathit{EnvNext} \wedge \mathit{Earlier}(\mathit{EnvNext})) \Rightarrow \mathit{SysNext} \\ \text{IN} & P \equiv Q \end{array}$$

Tsay [183] calls *UpToNow* "weak" and *Earlier* "strong" assume-guarantee specifications. We can also observe that these correpond to causal and strictly causal systems. Another comparison worth observing is

$$\begin{array}{l} \wedge\ \mathit{SysNext} \\ \wedge\ \mathit{EnvNext} \Rightarrow \mathit{Target} \end{array}$$

versus

$$(\mathit{EnvNext} \Rightarrow (\mathit{SysNext} \wedge \mathit{Target})) \equiv \begin{array}{l} \wedge\ \mathit{EnvNext} \Rightarrow \mathit{SysNext} \\ \wedge\ \mathit{EnvNext} \Rightarrow \mathit{Target} \end{array}$$

The order of quantifiers in Table 9.1 corresponds to whether a Moore strategy ($\exists y' : \forall x'$) or a Mealy strategy ($\forall x' : \exists y'$) is used.

9.6.2 Comparing steps

Let

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

$$\mathit{StepU}(\mathit{EnvNext},\ \mathit{SysNext}) \triangleq \quad \mathit{Moore}, \mathit{UpToNow}$$
$$\exists\, y' :\ \forall\, x' :\ \mathit{EnvNext} \Rightarrow \begin{array}{l} \wedge\ \mathit{SysNext} \\ \wedge\ \mathit{Target} \end{array}$$

$$\mathit{StepAE}(\mathit{EnvNext},\ \mathit{SysNext}) \triangleq \quad \mathit{Mealy}, \mathit{Earlier}$$
$$\forall\, x' :\ \exists\, y' :\ \begin{array}{l} \wedge\ \mathit{SysNext} \\ \wedge\ \mathit{EnvNext} \Rightarrow \mathit{Target} \end{array}$$

$$\mathit{StepAEU}(\mathit{EnvNext},\ \mathit{SysNext}) \triangleq \quad \mathit{Mealy}, \mathit{UpToNow}$$
$$\forall\, x' :\ \exists\, y' :\ \mathit{EnvNext} \Rightarrow \begin{array}{l} \wedge\ \mathit{SysNext} \\ \wedge\ \mathit{Target} \end{array}$$

We show with a few examples why *Step* is the only operator that does not lead to circular dependency of components. This conclusion follows from requiring symmetry, i.e., that all components are Moore and specified in the same style.

Otherwise, heterogeneous collections of specifications can be used to avoid circularity. However, heterogeneity means additional complexity, translations of formulas (repriming) when comparing assumptions of one component to guarantees of another component, and so likely more errors.

There are two levels to consider:

1. What we can deduce from the specifications only. This is useful because we can work without involving lower-level details, in particular we don't need to reason about alternating quantifiers in order to tease more conclusions out of the problem description. This level means considering *Earlier* or *UpToNow*. It ignores whether the implementation is *Mealy* or *Moore*, unless quantifiers appear in the temporal specification (as is the case with *Unzip*).

2. Both of specification and realizability definition are taken into account (so *Earlier* or *UpToNow* together with Mealy or Moore). In this way we can deduce more at the expense of reasoning about quantifiers.

For each case we first consider the specification level (which does not distinguish between Mealy or Moore, so at that level we can prove the same things about *SpecU* and *SpecAEU*). Then we consider the implementation level too.

9.6.2.1 Up to now

Let

$$NextA \triangleq x' = 1$$
$$NextB \triangleq y' = 1$$

$$NA \triangleq UpToNow(NextB) \Rightarrow NextA$$
$$NB \triangleq UpToNow(NextA) \Rightarrow NextB$$

$$A \triangleq \Box NA$$
$$B \triangleq \Box NB$$

Even if both components implement their specifications, we cannot deduce that the assembly implements $\Box(NextA \wedge NextB)$

$$\Box(\neg NextA \wedge \neg NextB) \;\Rightarrow\; (A \wedge B)$$

(For example, $\Box((x' = -5) \wedge (y' = -5)) \Rightarrow \Box(\neg NextA \wedge \neg NextB)$. Clearly $\not\models (A \wedge B) \Rightarrow \Box(NextA \wedge NextB)$.) This failure is because *UpToNow* allows a

step that satisfies $NA \wedge NB$ to violate both actions *NextA* and *NextB*. This possibility becomes evident by rewriting as

$$\begin{aligned}(NA \wedge NB) \;\equiv\;& \wedge (NextA \wedge Earlier(NextB)) \Rightarrow NextB \\ & \wedge (NextB \wedge Earlier(NextA)) \Rightarrow NextA\end{aligned}$$

So we cannot deduce anything useful from specifications of the *UpToNow* form.

The reader will object that *StepU* is stronger than *StepAEU*. This is true, so by considering the implementation level we can deduce more about Moore components that implement the same *UpToNow* property. The circularity above arises because component A can pick an x' that violates action *NextA* in the same step that component B picks a y' that violates action *NextB*. This is possible for Mealy components because A's choice of x' depends on y' and B's choice of y' depends on x'. This dependence is formalized by including these variables as arguments of the next-step functions that implement A and B. What it means is that we assume components A and B will be able to communicate *within* the same step (intrastep communication).

9.6.2.2 Moore and up to now

But a Moore component A cannot depend on y', nor can a Moore component B depend on x'. Does *StepU* avoid the unistep failure that *UpToNow* does? Only in part. Moore components cannot pick x' and y' simultaneously in a way that violates both *NextA* and *NextB*, but only if not both components are unblocked.

What *StepU* still allows is two Moore components being blocked in the same behavior state ("simultaneously"–remember this is not physical time). So two components becoming blocked in the same step. The last example does not demonstrate this situation, but the next one does.

$$\begin{aligned}NextA &\triangleq (x = 1) \wedge (x' = 1) \\ NextB &\triangleq (y = 1) \wedge (y' = 1)\end{aligned}$$

$$\begin{aligned}NA &\triangleq UpToNow(NextB) \Rightarrow NextA \\ NB &\triangleq UpToNow(NextA) \Rightarrow NextB\end{aligned}$$

$$\begin{aligned}A &\triangleq \Box NA \\ B &\triangleq \Box NB\end{aligned}$$

Two Moore components that implement A and B can *start* from any state that satisfies $(x \neq 1) \wedge (y \neq 1)$, and behave arbitrarily. Though from a

state that satisfies $(x = 1) \wedge (y = 1)$ the assembly does imply the invariant $(x = 1) \wedge (y = 1)$. In more detail

$$\begin{aligned} \mathit{StepA} \triangleq\ & \exists\, x' :\ \forall\, y' :\ \ \vee \neg((y = 1) \wedge (y' = 1)) \\ & \qquad\qquad\qquad\quad \vee (x = 1) \wedge (x' = 1) \\ \equiv\ & \exists\, x' :\ (y = 1) \Rightarrow ((x\ \ = 1) \wedge (x' = 1)) \end{aligned}$$

$$\begin{aligned} \mathit{StepB} \triangleq\ & \exists\, y' :\ \forall\, x' :\ \ \vee \neg((x = 1) \wedge (x' = 1)) \\ & \qquad\qquad\qquad\quad \vee (y = 1) \wedge (x' = 1) \\ \equiv\ & \exists\, y' :\ (x = 1) \Rightarrow ((y\ \ = 1) \wedge (y' = 1)) \end{aligned}$$

A remedy is to conjoin initial conditions

$$\begin{aligned} \mathit{InitA} &\triangleq x = 1 \\ \mathit{InitB} &\triangleq y = 1 \end{aligned}$$

$$\begin{aligned} \mathit{NextA} &\triangleq (x = 1) \wedge (x' = 1) \\ \mathit{NextB} &\triangleq (y = 1) \wedge (y' = 1) \end{aligned}$$

$$\begin{aligned} \mathit{NA} &\triangleq \mathit{UpToNow}(\mathit{NextB}) \Rightarrow \mathit{NextA} \\ \mathit{NB} &\triangleq \mathit{UpToNow}(\mathit{NextA}) \Rightarrow \mathit{NextB} \end{aligned}$$

$$\begin{aligned} \mathit{NewA} &\triangleq \mathit{InitA} \wedge \Box \mathit{NA} \\ \mathit{NewB} &\triangleq \mathit{InitB} \wedge \Box \mathit{NB} \end{aligned}$$

The assembly of Moore components that implement *NewA* and *NewB* does imply the invariant $(x = 1) \wedge (y = 1)$. So the result we obtain in this way is as good as we can obtain with specifications that use *Earlier* instead of *UpToNow*. What advantage does *Step* have over *StepU*?

The operator *Step* corresponds to a specification that uses *Earlier*. So we can prove properties of the assembly by working only with the specification. In contrast, using *UpToNow* (so *StepU*), we must take into account that the components are Moore (via the definition of realizability, thus at the cost of reasoning about alternating quantifiers). Later we will compare *Step* and *StepU* in more detail.

Surprises by Moore and up to now Even with initial conditions, *UpToNow* can unexpectedly allow circularity. Consider the specification

$$\begin{aligned} \mathit{InitA} &\triangleq x = 1 \\ \mathit{InitB} &\triangleq y = 1 \end{aligned}$$

$$\mathit{NextA} \triangleq (x = 1) \wedge (x' \in 1 \,..\, 2)$$

$NextB \triangleq (y = 1) \wedge (y' \in 1 \,..\, 2)$

$NA \triangleq UpToNow(NextB) \Rightarrow NextA$
$NB \triangleq UpToNow(NextA) \Rightarrow NextB$

$A \triangleq InitA \wedge \Box NA$
$B \triangleq InitB \wedge \Box NB$

Thinking in terms of the closed-systems

$ClosedA \triangleq InitA \wedge \Box NextA$
$ClosedB \triangleq InitB \wedge \Box NextB$

one would expect that $(A \wedge B) \Rightarrow \Box(NextA \wedge NextB)$. This is not the case. Starting from a state that satisfies $InitA \wedge InitB$, a Moore component A can pick $x' = 2$ and a Moore component B can pick $y' = 2$. Both satisfy $NextA$ and $NextB$. The second state satisfies $(x = 2) \wedge (y = 2)$. From the second state onwards $\neg UpToNow(NextA) \wedge \neg UpToNow(NextB)$ (because the conjuncts $(x = 1)$ of $NextA$ and $(y = 1)$ of $NextB$ are unsatisfiable in the second state). So the assembly can behave arbitrarily after the second state. In contrast, the specifications

$NAPlus \triangleq Earlier(NextB) \Rightarrow NextA$
$NBPlus \triangleq Earlier(NextA) \Rightarrow NextB$

$APlus \triangleq InitA \wedge \Box NAPlus$
$BPlus \triangleq InitB \wedge \Box NBPlus$

do not allow any circularity to arise, and are realizable by two Moore components. The assembly of Moore components that realize $APlus$ and $BPlus$ does imply the invariant $(x = 1) \wedge (y = 1)$.

9.6.2.3 What about primed environment variables in the system action

The presence of $\forall x'$ within $Step$ means that if $SysNext$ depends on x', then that dependence is erased. It must be the case that $SysNext$ is satisfiable by some choice of y' *for any arbitrary* x'. This limitation may appear severe. Two observations show it is not.

A specification of the form

$StepwiseImpl(EnvNext, SysNext) \triangleq \Box(Earlier(EnvNext) \Rightarrow SysNext)$

is implemented as a Moore component using $Step$. If x' occurs in $SysNext$, then we can show that this presence of x' is *inessential*, in the sense that we can

write the realizable behaviors allowed by this property using the same operator (*StepwiseImpl*) together with different arguments (*EnvNext* and *SysNext*).

$$
\begin{array}{l}
\exists\, v :\ \forall\, u :\ \ \wedge\, \mathit{SysNext}(x,\ y,\ u,\ v) \\
\qquad\qquad\qquad\ \ \wedge\, \mathit{EnvNext}(x,\ y,\ u,\ v) \Rightarrow\ \mathit{Target}(u,\ v) \\
\equiv \\
\exists\, v :\ \ \wedge \forall\, u :\ \mathit{SysNext}(x,\ y,\ u,\ v) \\
\qquad\qquad\ \ \wedge \forall\, u :\ \mathit{EnvNext}(x,\ y,\ u,\ v) \Rightarrow\ \mathit{Target}(u,\ v) \\
\equiv \\
\exists\, v :\ \ \wedge \forall\, R :\ \mathit{SysNext}(x,\ y,\ R,\ v) \quad \text{define this as } \mathit{NewSysNext} \\
\qquad\qquad\ \ \wedge \forall\, u\ :\ \mathit{EnvNext}(x,\ y,\ u,\ v) \Rightarrow\ \mathit{Target}(u,\ v) \\
\equiv \\
\exists\, v :\ \ \wedge\, \mathit{NewSysNext}(x,\ y,\ v) \\
\qquad\qquad\ \ \wedge \forall\, u :\ \mathit{EnvNext}(x,\ y,\ u,\ v) \Rightarrow\ \mathit{Target}(u,\ v) \\
\equiv \\
\exists\, v :\ \ \wedge \forall\, u :\ \mathit{NewSysNext}(x,\ y,\ v) \\
\qquad\qquad\ \ \wedge \forall\, u :\ \mathit{EnvNext}(x,\ y,\ u,\ v) \Rightarrow\ \mathit{Target}(u,\ v) \\
\equiv \\
\exists\, v :\ \forall\, u :\ \ \wedge\, \mathit{NewSysNext}(x,\ y,\ v) \\
\qquad\qquad\qquad\ \ \wedge\, \mathit{EnvNext}(x,\ y,\ u,\ v) \Rightarrow\ \mathit{Target}(u,\ v)
\end{array}
$$

Thus we lose no expresiveness by restricting to specifications with action *SysNext* that has no occurrence of x'. This observation applies when we consider only specifications that can be written using *Earlier* and a Moore component. What about the more general case?

In general, a Moore component cannot react to x', so dependence of the choice y' on what x' *will be* leads to unrealizability [66]. This issue arises due to unbounded quantification. A quantifier bound is necessary, and this bound takes the form of *EnvNext*. However, as we saw earlier, *EnvNext* as antecedent leads to circularity at the specification level (i.e., it is too lax). For this reason we want to have a conjunct that is unconditional (as we do with *SysNext* within *Step*). But what possibilities exist in between these two extremes?

Combining these two alternatives, we arrive at the step operator

$$
\begin{array}{l}
\mathit{StepHalf}(\mathit{EnvNext},\ \mathit{SysNextImmediate},\ \mathit{SysNextLater}) \ \triangleq \\
\qquad \wedge\, \mathit{SysNextImmediate} \\
\qquad \wedge\, \mathit{EnvNext} \Rightarrow\ \wedge\, \mathit{SysNextLater} \\
\qquad\qquad\qquad\qquad\ \ \wedge\, \mathit{Target}
\end{array}
$$

The action *SysNextImmediate* cannot mention x' in any essential way (any such mention is unrealizable, and can be rewritten with another *SysNextImmediate* that does not mention x', as we showed above). The action *SysNextLater can* mention x'. It seems that we have more "degrees of freedom" with such an operator (this operator actually corresponds to *Unzip* when $SysNextImmediate \equiv \exists x' : SysNextLater$). Is it so; in other words can we express more with this step operator than what we can using *Step*?

No. Whatever we can write with *StepHalf* is expressible with *Step*, by appropriately defining the action *SysNext* (we show this below). Let us express *StepHalf* using *Step*. Define

$$\begin{aligned} SysNext \triangleq\ & \wedge\, SysNextImmediate \\ & \wedge\, EnvNext \Rightarrow SysNextLater \end{aligned}$$

and rewrite as follows

$$\begin{aligned} & StepHalf(EnvNext,\ SysNextImmediate,\ SysNextLater) \\ \equiv\ & \wedge\, SysNextImmediate \\ & \wedge\, EnvNext \Rightarrow SysNextLater \\ & \wedge\, EnvNext \Rightarrow Target \\ \equiv\ & \wedge\, SysNext \\ & \wedge\, EnvNext \Rightarrow Target \\ \equiv\ & Step(EnvNext,\ SysNext) \end{aligned}$$

This proves that we can write any Moore interaction that we are interested in a specification form that uses *Earlier* only, and so use *Step*. The importance is that we can use the same solvers for synthesis (which are based on *Step*), after a suitable initial preprocessing of the component actions.

The part that is removed above is called the non-Cartesian part of the actions. In other words, the extra freedom that *StepHalf* specifications allow expressing is non-Cartesian, thus unrealizable by a Moore component. Only the Cartesian part is realizable (equiv. the realizable is Cartesian) and any Cartesian specification can be expressed using *Earlier* (*Step*).

Remark 15. Mealy components introduce combinational cycles, unless a mixture of different kinds of components is used [51]. We do not consider this arrangement, because it is asymmetric [54], and it involves intrastep communication. □

In summary, we answered the question of whether any behavior realizable by a Moore component is specifiable using *Earlier* (thus with *Step*), which requires that *SysNext* be independent of x'.

9.6.2.4 Earlier

Unlike *UpToNow*, when composing systems specified by *Earlier* we can deduce properties of the assembly without encountering circularity.

THEOREM
ASSUME ACTION $NextA$, ACTION $NextB$
PROVE LET
$NA \triangleq Earlier(NextB) \Rightarrow NextA$
$NB \triangleq Earlier(NextA) \Rightarrow NextB$
$A \triangleq \Box NA$
$B \triangleq \Box NB$
IN
$(A \land B) \Rightarrow \Box(NextA \land NextB)$

$\langle 1 \rangle 1.\ (A \land B) \equiv \Box(NA \land NB)$
$\langle 1 \rangle$ DEFINE
$NA \triangleq Earlier(NextB) \Rightarrow NextA$
$NB \triangleq Earlier(NextA) \Rightarrow NextB$
$N \triangleq NA \land NB$
$InvAct \triangleq NextA \land NextB$
$\langle 1 \rangle 2.\ N \equiv \land Earlier(NextB) \Rightarrow NextA$
$\land Earlier(NextA) \Rightarrow NextB$
BY DEF N, NA, NB
$\langle 1 \rangle 3.\ (\Box N) \Rightarrow InvAct$

Remember that we use RTLA$^+$in this section, so the inital condition can be an action.

BY $\langle 1 \rangle 2$ DEF *Earlier*
$\langle 1 \rangle 4.\Box((InvAct \land N) \Rightarrow InvAct)$
BY $\langle 1 \rangle 2$ DEF *Earlier*
$\langle 1 \rangle 5.\ (\Box N) \Rightarrow \Box(InvAct)$
BY $\langle 1 \rangle 3$, $\langle 1 \rangle 4$
$\langle 1 \rangle$ QED
BY $\langle 1 \rangle 5$ DEF N, NA, NB, $InvAct$

So what is the difference between Mealy and Moore implementations when the components are specified by *Earlier* properties?

9.6.2.5 Earlier and Mealy

As with *UpToNow*, we can prove more assembly properties if we consider Moore components. But the reason is that Moore implementations can be unrealizable when the same specifications are realizable by Mealy implementations. So we prove the assembly property (an upper bound) due to a false antecedent. The separate requirement of realizability fails in this case for the Moore components (the lower bound).

In other words, Mealy implementations can do more than Moore implementations (a well-known difference). We can prove the same assembly specifications from component specifications, but those component specifications can be realizable as Mealy components, and unrealizable as Moore components. This is how *Step* differs from *StepAE*.

Demonstrating this with an example

$$NextA \triangleq x' = y'$$
$$NextB \triangleq y' = x'$$

These steps are of the *StepAE* form.

$$\begin{aligned} StepA \triangleq\ & \forall\, y' :\ \exists\, x' :\ \ \wedge NextA \\ & \qquad\qquad\qquad\ \wedge NextB \Rightarrow \text{TRUE} \end{aligned}$$

With TRUE as *Target*.

$$\begin{aligned} &\equiv \forall\, y' :\ \exists\, x' : \\ &\quad \wedge x' = y' \\ &\quad \wedge (x' = y') \Rightarrow \text{TRUE} \\ &\equiv \text{TRUE} \end{aligned}$$

$$\begin{aligned} StepB \triangleq\ & \forall\, x' :\ \exists\, y' :\ \ \wedge NextB \\ & \qquad\qquad\qquad\ \wedge NextA \Rightarrow \text{TRUE} \end{aligned}$$

$$\begin{aligned} &\equiv \forall\, x' :\ \exists\, y' : \\ &\quad \wedge x' = y' \\ &\quad \wedge (x' = y') \Rightarrow \text{TRUE} \\ &\equiv \text{TRUE} \end{aligned}$$

This example is the "canonical" non-Cartesian constraint. Realizing a non-Cartesian constraint requires intrastep communication. Two Mealy components can communicate within a single step. So by using Mealy components, we are *not* reasoning about steps that involve no communication. Therefore, circular dependence can arise with Mealy implementations regarding decisions about the next state of each component.

Moreover, refining a single component step into multiple steps can lead to one component making decisions using information about the future state of another component, which may be unknown at that point (e.g., due to future environment noise).

9.6.2.6 Earlier and Moore

The non-Cartesian example is unrealizable by Moore components. So an assembly of two Moore components *would* implement $\Box(NextA \wedge NextB)$, but no Moore components exist that implement the specifications $\Box(Earlier(NextB) \Rightarrow NextA)$ and $\Box(Earlier(NextA) \Rightarrow NextB)$. In detail

$$NextA \triangleq x' = y'$$
$$NextB \triangleq y' = x'$$

These steps are of the *Step* form.

$$\begin{aligned} StepA \triangleq\ & \exists\, x' :\ \forall\, y' :\ \ \wedge NextA \\ & \qquad\qquad\qquad\quad \wedge NextB \Rightarrow \text{TRUE} \\ \equiv\ & \exists\, x' :\ \forall\, y' : \\ & \quad \wedge x' = y' \\ & \quad \wedge (x' = y') \Rightarrow \text{TRUE} \\ \equiv\ & \text{FALSE} \end{aligned}$$

$$\begin{aligned} StepB \triangleq\ & \exists\, y' :\ \forall\, x' :\ \ \wedge NextB \\ & \qquad\qquad\qquad\quad \wedge NextA \Rightarrow \text{TRUE} \\ \equiv\ & \exists\, y' :\ \forall\, x' : \\ & \quad \wedge x' = y' \\ & \quad \wedge (x' = y') \Rightarrow \text{TRUE} \\ \equiv\ & \text{FALSE} \end{aligned}$$

Remark 16. The *Step* operator is more demanding than *StepAE*. It requires that the component be able to pick a y' that works *independently* of what x' is going to be.

The Mealy versus Moore choice is absent from the specification. It is determined by what definition of realizability we choose, so what communication arrangement we assume between the components. □

Table 9.2: Comparing *Step* and *StepU*. We let $E \triangleq$ ENABLED *EnvNext* and $S \triangleq$ ENABLED *SysNext*.

		Step	*StepU*
$\neg E$	$\neg S$	FALSE	TRUE
	S	TRUE	TRUE
E	$\neg S$	FALSE	FALSE
	S	*StepU*	*StepU*

9.6.3 Comparison of *Step* to *StepU*

Algebraic manipulation leads to a simple relation between the quantified subformulas of *Step* and *StepU*

$$\begin{array}{l} \wedge\, \mathit{SysNext}(x,\, y,\, y') \\ \wedge\, \mathit{EnvNext}(x,\, y,\, x') \Rightarrow \mathit{Target}(x',\, y') \end{array}$$

$$\begin{array}{ll} \equiv & \wedge\, \mathit{SysNext}(x,\, y,\, y') \\ & \wedge\, \mathit{EnvNext}(x,\, y,\, x') \Rightarrow \begin{array}{l} \wedge\, \mathit{SysNext}(x,\, y,\, y') \\ \wedge\, \mathit{Target}(x',\, y') \end{array} \end{array}$$

$$\begin{array}{ll} \equiv & \wedge\, \mathit{SysNext}(x,\, y,\, y') \\ & \wedge\, \mathit{CPreB}(x,\, y) \end{array}$$

The relation between *Step* and *StepU* is (under the assumption that *SysNext* does not depend on x')

$$\begin{array}{ll} \mathit{Step}(x,\, y) \equiv & \wedge\, \exists\, y' :\ \mathit{SysNext}(x,\, y,\, y') \\ & \wedge\, \mathit{StepU}(x,\, y) \end{array}$$

Step differs from *StepU* only at states where the component is blocked, as shown in Table 9.2. The *Step* operator requires that the component remain unblocked while *EnvNext* has not been violated by previous steps. The choice of y' should satisfy *SysNext* independently of x', and should lead to *Target* for all those x' that satisfy *EnvNext*. Steps that violate *EnvNext* can be ignored, because those lead to satisfaction of the specification in the next step.

Due to the circularity example with two blocking Moore components, it is reasonable to require enabledness:

$$\begin{array}{lll} \exists\, y' :\ & \wedge\, \exists\, x' :\ \mathit{SysNext}(x,\, y,\, x',\, y') & \text{The component should not block.} \\ & \wedge\, \forall\, x' :\ \mathit{EnvNext} \Rightarrow \begin{array}{l} \wedge\, \mathit{SysNext}(x,\, y,\, x',\, y') \\ \wedge\, \mathit{Target} \end{array} & \end{array}$$

$$\equiv$$

$$\exists\, y' :\ \ \wedge \exists\, x' :\ SysNext(x,\ y,\ x',\ y')$$
$$\wedge \forall\, x' :\ EnvNext \Rightarrow SysNext(x,\ y,\ x',\ y')$$
$$\wedge \forall\, x' :\ EnvNext \Rightarrow Target$$

Define

$$NewSysNext(x,\ y,\ y') \triangleq$$
$$\wedge \exists\, x' :\ SysNext(x,\ y,\ x',\ y')$$
$$\wedge \forall\, x' :\ EnvNext \Rightarrow Sysnext(x,\ y,\ x',\ y')$$

Substitute the definition above

$$\exists\, y' :\ \ \wedge NewSysNext(x,\ y,\ y')$$
$$\wedge \forall\, x' :\ EnvNext \Rightarrow Target$$
$$\equiv$$
$$\exists\, y' :\ \forall\, x' :\ \ \wedge NewSysNext(x,\ y,\ y')$$
$$\wedge EnvNext \Rightarrow Target$$

So even for a *SysNext* that does contain x', after we require enabledness, we can write the resulting controllable step operator equivalently using *Step*. Thus, there is no loss of generality.

Remark 17. Any collection of inter-dependent components needs to include a Moore component to ensure that circular dependence is avoided. Also, a suitable combination of *While* and *WhilePlus* properties can avoid circular dependence, but is asymmetric. □

9.7 Hiding history preserves realizability

For a specification that includes history-determined variables, we prove that it suffices to synthesize an implementation with the history variables unhidden. More precisely

LET

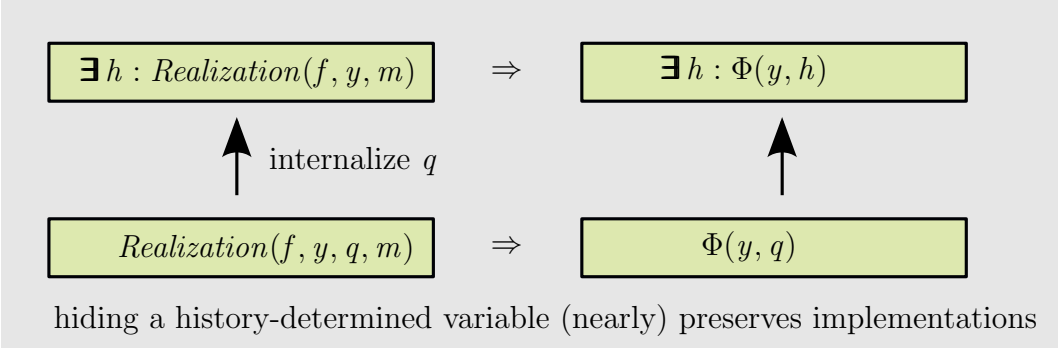
$$Spec(x,\ h) \triangleq Prop(x) \wedge History(x,\ h)$$
$$SpecH(x) \triangleq \exists\, h :\ Spec(x,\ h)$$

IN

$$IsRealizable(SpecH) \equiv IsRealizable(Spec)$$

This result is useful for using temporal synthesis algorithms that do not reason about $\exists$ (for example GR(1) synthesis), and then hiding the history variables, in order to obtain an implementation for properties that contain temporal quantification of only history variables.

Hiding history-determined variables [5, Sec. 2.4] preserves realizability, in the sense that an implementation for *Spec* also implements *SpecH*; after the hidden



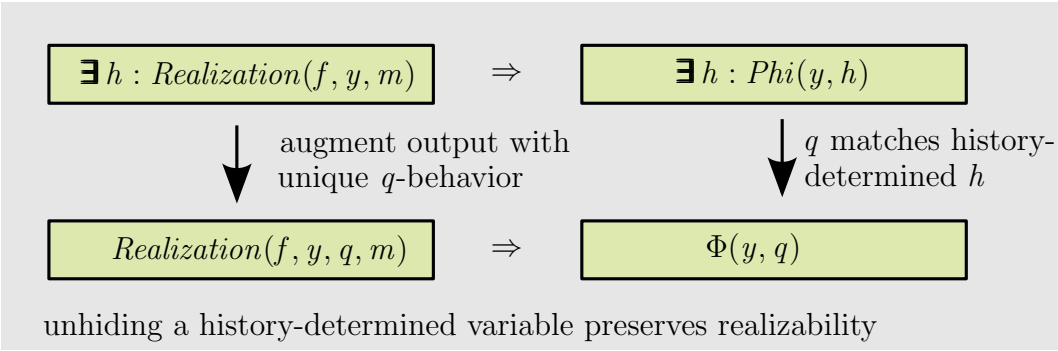


Figure 9.6: Reasoning along the two directions of the proof.

variables are turned into internal variables of the implementation (memory). Unhiding history-determined variables leads to a realizable specification because the implementation can simulate such variables using information from the already visible state.

In the context of GR(1) synthesis To implement a non-GR(1) property using GR(1) synthesis we need to use the hiding direction–the one that creates history variables. The unhiding direction is used to ensure that if GR(1) synthesis decides "unrealizable", then the non-GR(1) property too is unrealizable. The result of this section formalizes in (raw) TLA$^+$ the observation that GR(1) synthesis applies to any property specifiable by a deterministic Büchi automaton [153, p. 378]. The history variable represents the automaton's node at each state of a behavior.

An example of a property that is *not* equirealizable by expressing it in GR(1) with auxiliary variables is persistence, $\Diamond\Box P$. The auxiliary variable used to express persistence is *not* history-determined, and thus the below theorem does not apply in that case. For persistence properties, unhiding remains sound,

but not necessarily complete.

Hiding and unhiding history-determined variables History-determined variables exist [5, p. 1551] (see also [66]).

MODULE *HistoryDeterminedVar*

```
VARIABLE v,
CONSTANT Init(_, _)   corresponds to f in [5, Eq.(4)]
CONSTANT Next(_, _, _)   corresponds to g in [5, Eq.(4)]

Hist(h, v) ≜
    LET
        N ≜ ⟨h′ = Next(h, v, v′)⟩_v
    IN
        ∧ h = Init(v)
        ∧ □[N]_⟨h, v⟩
```

Temporal existential quantification below this point in this section is stutter-sensitive. By the proof of the theorem, it is possible to construct an implementation of the property *PhiH* from an implementation of the property *Phi*.

MODULE *RawHistoryDeterminedVar*

```
VARIABLE v,
CONSTANT Init(_, _)
CONSTANT Next(_, _, _)

Hist(h, v) ≜
    LET
        N ≜ h′ = Next(h, v, v′)
    IN
        ∧ h = Init(v)
        ∧ □N

THEOREM HistoryExists ≜   History-determined variables exist
also in raw TLA+, similarly to TLA+ [5].
    ∀ v : ∃ h : Hist(h, v)
```

```
THEOREM RealizingHistory ≜
    ASSUME
        CONSTANT finit, CONSTANT fnext,
```

CONSTANT $Init(\_, \_)$,

TEMPORAL $Phi(\_, \_, \_)$,

LET

$History(h, x, y) \triangleq$

$\land \quad h = finit[x, y]$

$\land \quad \Box(h' = fnext[h, x, y, x'])$

IN

$\forall\, x, y, h : \ Phi(x, y, h) \ \Rightarrow \ History(h, x, y)$

PROVE

LET

$I(x, y, h) \triangleq Init(x, y) \land (h = finit[x, y])$

$PhiH(x, y) \triangleq \exists\, h : \ Phi(x, y, h)$

IN

The controlled variables are not shown. *Phi* is realizable with h a controlled variable, and x is controlled in both cases. Variable y is part of the environment.

$IsRealizable(I, Phi) \ \equiv \ IsRealizable(Init, PhiH)$

9.8 Defining generalized reactivity

Syntax or semantics? A GR(200) formula can happen to be equivalent to a GR(1) formula. Should we call such a formula GR(1) or GR(200)? Recognizing that a GR(200) formula is equivalent to some GR(1) formula involves a step that reasons about equivalence, so this decision costs computation. For this reason we define "GR(1)" as "GR(1) formulas"; not as "formulas equivalent to some GR(1) formula". This definition is syntactic: a schema of formulas that describe liveness.

As the safety-liveness decomposition of stepwise implication properties indicates, their effect is on the safety part of a property. When a GR(1) formula occurs within a stepwise implication, it pertains to the liveness part (assuming machine-closure, otherwise it can determine also the safety part).

The complexity of temporal synthesis is governed by the nesting of fixpoints (assuming the one-step control problem isn't more expensive [51]). Categorizing synthesis as GR(k) emphasizes that the liveness subformulas determine the fixpoint nesting (though in practice computing the controllable step operator becomes a challenge in symbolic methods before the depth of fixpoint nesting).

GR(1) in the literature The syntactic nature of the definition is indicated in the literature by the phrases "class of GR(1) formulas" [153, Eq.(1), p. 365] and "implication between conjunctions of recurrence formulas" [153, §3 p. 369]. In the definition of generalized Streett[1] games [90, p. 49], the winning condition is one generalized Streett pair (formula). The safety part is defined semantically, using transition relations [90, §2, p. 40], and the model of computation is by definition interleaving [90, §3.1, p. 44].

9.8.1 Temporal quantification

Hidden variables can be used to record information. If the behavior of a hidden variable depends only on the past behavior of other variables, then it can be "revealed" by deleting the temporal quantifier, as described in Section 9.7. Otherwise an exponential increase in cost occurs (due to a subset construction) for hidden environment variables that are not history-determined [59]. So in the presence of quantification the merit of GR(1) synthesis is limited to the history-determined case.

In order to express quantifier-free non-GR(1) formulas of the form

$$Orig \triangleq Init \wedge \Box[Next]_v \wedge NonGR1Liveness$$

as quantified formulas in GR(1), we need to introduce an auxiliary variable

$$New \triangleq \exists\, h : NewInit(h) \wedge \Box[NewNext(h)]_{\langle v,h\rangle} \wedge GR1Liveness(h)$$

The history variable h adds state that needs to be reasoned about during synthesis of implementations. So synthesis is implemented with history-determined variables unhidden. The complexity depends on the number of *all* variables, both free and bound.

Chapter 10

THE EXISTENCE OF GR(1) CONTRACTS WITH FULL INFORMATION

10.1 Preserving closure and refining

In this section we investigate whether given a GR(1) property[1] φ, a contract of realizable GR(1) properties φ_1, φ_2 exists such that conjoining them ensures φ

$$(\varphi_1 \wedge \varphi_2) \Rightarrow \varphi$$

and safety not be constrained, i.e., the closure of the conjoined properties be equivalent to the closure of the assembly property

$$Cl(\varphi_1 \wedge \varphi_2) \equiv Cl(\varphi)$$

The first requirement ensures that a system assembled from components that implement φ_1 and φ_2 implements the assembly specification φ. The second requirement aims to avoid placing premature constraints on component specifications (i.e., introduce liveness instead of safety constraints). As we show below, avoiding additional safety constraints has implications on when such a decomposition is possible.

GR(1) synthesis admits the addition of history-determined variables [3], because unhiding such variables yields essentially equisynthesizable specifications [153, 23]. However, history variables also add state. For this reason, we first investigate whether *memoryless* contracts exist, i.e., contracts of specifications over the variables that φ depends on. We show that such contracts do not always exist, not even for GR(1) games with a single recurrence goal.

We then relax the requirement of remaining within GR(1), and describe a decomposition algorithm that yields "nested GR(1)" properties, which have a "request-response" structure. We next show that any such "nested GR(1)" property is equivalent to a GR(1) property with an auxiliary variable added (ranging over a set of values at worst linear in the state space—typically small).

[1] The requirement of state predicates within recurrence formulas in GR(1) is important for the results of this section.

Thus, we prove that GR(1) contracts exist, albeit at the price of adding history-determined variables (memory). In other words, *stateful* GR(1) contracts exist (unlike memoryless ones).

Even though nested GR(1) properties do not constrain safety, their realizable part constrains safety, due to the structure of the decomposition algorithm (and in general this is unavoidable). Therefore, directly constraining safety to obtain a GR(1) property without adding history variables is not restrictive.

One may wonder what would change if we dropped the requirement

$$(\varphi_1 \wedge \varphi_2) \Rightarrow \varphi.$$

We would expect that the weakest requirement that may work is to have the realizable parts $\mathcal{R}(\varphi_1)$ and $\mathcal{R}(\varphi_2)$ [4, 3, 113] of φ_1 and φ_2 in their place, i.e.,

$$(\mathcal{R}(\varphi_1) \wedge \mathcal{R}(\varphi_2)) \Rightarrow \varphi.$$

As we will see below, in absence of logical refinement, the conjunction of realizable parts does not imply φ. So there is no gain in replacing each φ_i by its realizable part.

Our results about the need for memory should not be surprising, in light of results on the need for auxiliary variables to define refinement mappings [2], and the inexistence of memoryless strategies for GR(1) games [85].

We discuss the existence of contracts in terms of two properties, but the conclusions have implications for the entire collection of (stepwise) GR(1) formulas. The safety part is mostly ignored throughout the discussion, because we work with specific counterexamples in which we leave safety constraints unchanged.

The approach is focused on the case of a specification φ with a single recurrence goal. Multiple recurrence goals can be handled by first adding a history variable that "chains" them together (equivalently defining a deterministic Büchi automaton that cycles through the goals), and thus reducing them to a single recurrence goal. The memory added by such a transformation is as large as the memory typically added during GR(1) synthesis (linear in the number of recurrence goals), so there is no loss of generality in such a transformation.

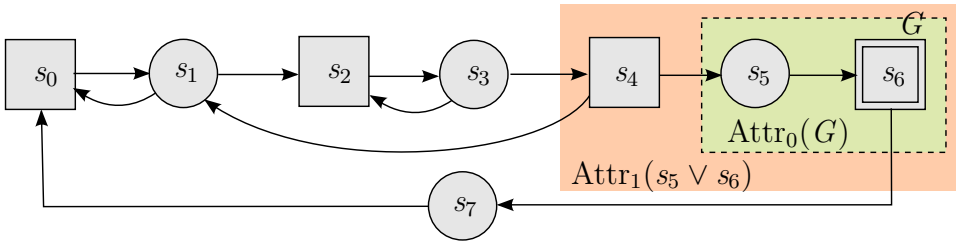


Figure 10.1: Game where a liveness assumption realizable by player 1, and sufficient for player 0, does not exist. Player 0 (player 1) moves from disks (boxes).

10.2 Memoryless contracts

10.2.1 One-sided counterexample

The example shown in Fig. 10.1 has the main feature that leads to inexistence of memoryless GR(1) contracts. The figure shows a game between two players, each player chooses the outgoing edge from nodes of the same shape. The objective is to find a GR(1) contract for the two players, so that together they implement $\varphi \triangleq \square\diamond s_6$. We start by trying to find a (generalized) Streett(1) liveness goal for the disk player that relaxes $\square\diamond s_6$, because $\square\diamond s_6$ is unrealizable by the disk player. This turns out to be impossible: for each candidate persistence goal $\diamond\square P$, either

- $(\diamond\square P) \vee \square\diamond s_6$ is unrealizable by the disk player, or
- $\diamond\square P$ is realizable by the disk player (and, clearly, P contains a cycle that omits s_6—the only P that contains a cycle with s_6 includes all nodes, so is equivalent to TRUE).

In more detail, $\square\diamond s_6$ alone is unrealizable. Any P that does not contain any cycle yields a property $(\diamond\square P) \vee \square\diamond s_6$ that is equivalent to $\square\diamond s_6$ (due to the safety constraints that represent the game graph). The smallest cycles are $C_1 \triangleq \{s_0, s_1\}$ and $C_2 \triangleq \{s_2, s_3\}$. Thus any persistence set P that contains a cycle contains at least one of these two cycles. Thus, whenever $(\diamond\square P) \vee \square\diamond s_6$ is weaker than $\square\diamond s_6$ (so possibly realizable), $\diamond\square P$ is realizable too, i.e., the disk player can choose to remain forever in C_1 or forever in C_2, instead of repeatedly visiting s_6.

More formally (the below proof sketch and the proof in [65] take a different approach than the outline above)[2]

[2]The symbol $\overset{\text{sr}}{\Rightarrow}$ is shorthand for $(\square\rho_e \wedge W_e) \overset{\text{sr}}{\Rightarrow} (\square\rho_s \wedge W_s) \triangleq \square\big((\circledcirc\boxminus\rho_e) \Rightarrow \rho_s\big) \wedge$

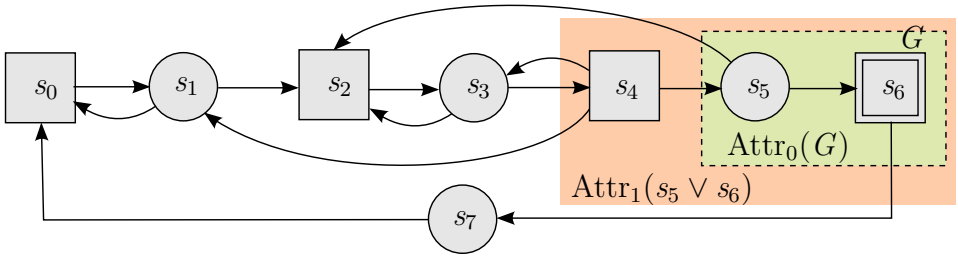


Figure 10.2: Two-sided counter-example where a GR(1) contract does not exist. Compare to Fig. 10.1.

Proposition 3. ASSUME*: Define the transition relations* ρ_0, ρ_1 *by the game graph of Fig. 10.1, the set of nodes* $V \triangleq \{s_0, \ldots, s_7\}$*, and the goal* $[\![G]\!] \triangleq \{s_6\}$ *of player 0.* PROVE*: For all sets* $[\![P]\!] \subseteq V$*, with* $\psi_1 \triangleq (\Box\rho_0) \overset{\text{sr}}{\Rightarrow} \big(\Box\rho_1 \wedge \Box\Diamond P\big)$ *and* $\psi_0 \triangleq \big(\Box\rho_1 \wedge \Box\Diamond P\big) \overset{\text{sr}}{\Rightarrow} \big(\Box\rho_0 \wedge \Box\Diamond G\big)$*, it is* $[\![\mathrm{Win}(0,\psi_0)]\!] \cap [\![\mathrm{Win}(1,\psi_1)]\!] = \emptyset$*.*

PROOF SKETCH: For any P that does not intersect $\{s_0, \ldots, s_3\}$, player 1 cannot win, because player 0 can force, and keep, the play outside of P. For similar reasons, P should intersect each of $\{s_0, s_1\}$ and $\{s_2, s_3\}$. If P intersects $\{s_0, s_1\}$ and $\{s_2, s_3\}$, then player 1 can win by always moving from s_4 to s_1, when the play comes to s_4. This forces a visit to either both s_0 and s_1, or both s_2 and s_3. So, for no P do both players have a winning strategy. A proof can be found in [65].

10.2.2 General counterexample

Assigning the goal $\Box\Diamond s_6$ to the box player in Fig. 10.1 reveals that there is a GR(1) contract $(\Diamond\Box(s_0 \vee s_1 \vee s_2 \vee s_3) \vee \Box\Diamond s_6$ for the box player, and $\Box\Diamond(s_4 \vee s_5 \vee s_6 \vee s_7)$ for the disk player).

By adding two edges to Fig. 10.1, we obtain the example of Fig. 10.2. There is no safety-preserving GR(1) contract for this example. A proof sketch is as follows. Assume that all recurrence goals of both players have the form $\Box\Diamond R$, where R contains nodes other than $\{s_6, s_7\}$. If so, then

$$\not\models (\varphi_1 \wedge \varphi_2) \Rightarrow \varphi$$

because all the recurrence goals in $\varphi_1 \wedge \varphi_2$ are satisfiable by a behavior that cycles through all nodes in $\{s_i : i \in 0..5\}$, and forever avoids s_6. Similarly, any nontrivial realizable persistence goal omits s_6.

$((\Box\rho_e \wedge W_e) \Rightarrow W_s) \equiv (\Box\rho_e) \overset{\text{sr}}{\Rightarrow} \big(\Box\rho_s \wedge (W_s \Rightarrow W_e)\big)$.

Thus, at least one component specification, φ_1 or φ_2, must include $\square\diamond s_6$ as a conjunct (or the equivalent $\square\diamond(s_6 \vee s_7)$). So it suffices to reason only about specifications where either one of the players has the goal $\square\diamond s_6$. Additional recurrence goals only strengthen the specifications, and we split cases by realizability of the persistence goals.

The goal $\square\diamond s_6$ is unrealizable without a relaxation by persistence goals. Each relaxation is either unrealizable, or if realizable, then some nontrivial persistence goal is realizable, which avoids s_6. The root cause is the same as for the one-sided example. For the box player, getting past s_5 requires assuming that "waiting" in a set of nodes that includes s_5 will lead to an eventual response by the disk player. However, such a waiting set (persistence goal) would have to contain $\{s_2, s_3, s_4\}$. The box player can remain forever in this set. Similar observations apply to the disk player. Therefore, a GR(1) contract does not exist for this example.

The claims of both the one-sided and the two-sided examples for the goal $\square\diamond s_6$ assigned to each player have been checked by a machine, using a GR(1) solver (the Python module `omega.games.gr1`), and multiple parametric persistence goals $\diamond\square P_1 \vee \diamond\square P_2 \vee \ldots \vee \diamond\square P_{64}$ (the parameters are BDD bits that are not quantified during computations).

10.2.3 What about refinement by realizable parts?

What if we replaced the requirement $(\varphi_1 \wedge \varphi_2) \Rightarrow \varphi$ with the corresponding realizable parts? A behavior σ satisfies the realizable part $\mathcal{R}(\varphi_1)$ of the property φ_1 if there exists some implementation of φ_1 that gives rise to this behavior (under some environment behavior). We show below that memoryless GR(1) contracts do not exist even if we relax the requirement in this way, using the example of Fig. 10.2.

Assume that $\not\models (\varphi_1 \wedge \varphi_2) \Rightarrow \varphi$ but

$$(\mathcal{R}(\varphi_1) \wedge \mathcal{R}(\varphi_2)) \Rightarrow \varphi$$

(a contract requires realizability, and thus $\mathcal{R}(\varphi_1)$ and $\mathcal{R}(\varphi_2)$ are not false). Pick two strategies f, g that realize φ_1 and φ_2, respectively. If either f or g realizes a goal $\diamond\square P$ that implies $\diamond\square\neg s_6$, then we are done showing that the implication fails.

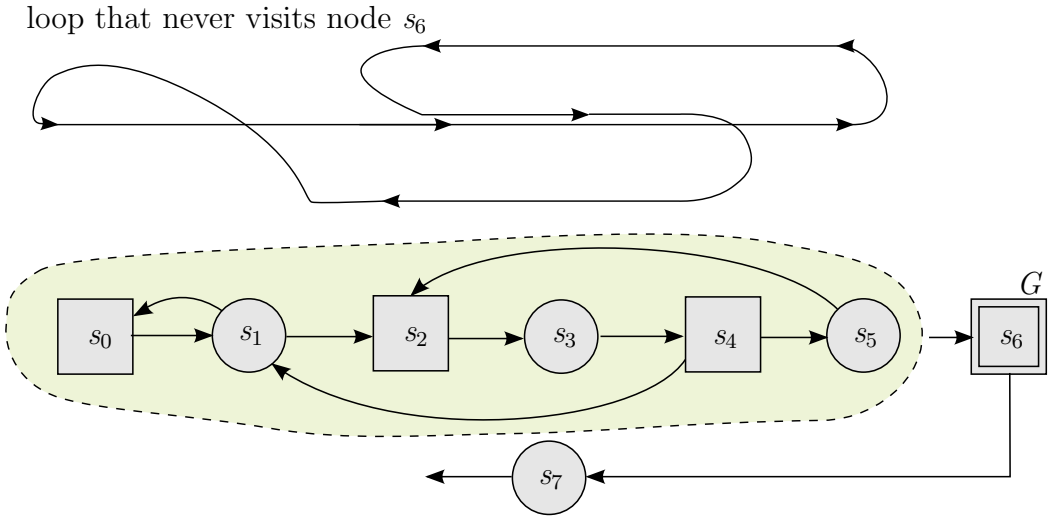


Figure 10.3: Two strategies that violate φ, thus showing that in case $\not\models (\varphi_1 \wedge \varphi_2) \Rightarrow \varphi$, conjoining realizable parts does not ensure that φ is implemented.

Otherwise $\varphi_1 \wedge \varphi_2$ contains recurrence goals that are satisfied by a behavior that satisfies $\Diamond\Box\neg s_6$ (by the assumption above). Construct two new strategies p, q for each player, by the following procedure:

- Stategy p for the disk player chooses the edge $s_5 \wedge s_2'$ and alternates between $s_1 \wedge s_0'$ and $s_1 \wedge s_2'$ on each visit to s_1, unless for 100 steps the recurrence goals within φ_1 are not visited. If so, then p switches to using f, otherwise it renews counting steps.
- Strategy q for the box player alternates between $s_4 \wedge s_5'$ and $s_4 \wedge s_1'$ on each visit to s_4, and switches to g under similar conditions to p.

The strategies p and q will never switch to f and g, because the recurrence goals in $\varphi_1 \wedge \varphi_2$ are all visited within 100 steps, by visiting all the nodes $\{s_i : i \in 0..5\}$. Thus

$$\not\models (\mathcal{R}(\varphi_1) \wedge \mathcal{R}(\varphi_2)) \Rightarrow \varphi,$$

which is a contradiction. In other words, if $\not\models (\varphi_1 \wedge \varphi_2) \Rightarrow \varphi$, then any two strategies that implement φ_1 and φ_2 can happen to "conspire" so that when assembled, the resulting behavior violates φ, as shown in Fig. 10.3.

10.3 Stateful contracts

10.3.1 Nested GR(1)

We proved earlier that memoryless safety-preserving GR(1) contracts need not always exist. To overcome this limitation, we extend the class of properties as follows, and then show that these are equivalent to GR(1) properties with auxiliary history-determined variables added.

Definition 4 (Chain)**.** *Let* $d \in Nat, H_k \in Nat, \Xi_m \in Nat$*. A* chain *condition is*

$$\begin{aligned}
&\wedge\, \forall m \in 1..d \,:\, \forall l \in 0..\Xi_m \,:\, \wedge\, P_{m-1} \Rightarrow Q_m \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge\, \xi_{ml} \Rightarrow \neg P_{m-1} \\
&\wedge\, \forall m \in 0..d \,:\, \wedge\, Q_m \Rightarrow P_m \\
&\qquad\qquad\qquad \wedge\, \forall l \in 0..H_m \,:\, \eta_{ml} \Rightarrow (P_m \wedge \neg Q_m) \\
&\qquad\qquad\qquad \wedge\, \forall l \in \Xi_m \,:\, \xi_{ml} \Rightarrow Q_m
\end{aligned}$$

Definition 5 (Nested GR(1) property [67])**.** *Assuming the chain conditions of Definition 4 hold, then*

$$\begin{aligned}
\varphi \;\triangleq\; \Box \bigwedge\nolimits_{m\in 0..d} \wedge\; &\vee\, \neg P_m \\
&\vee\, \neg \bigwedge\nolimits_{k\in m..d} \bigwedge\nolimits_{l\in 0..H_k} \Box\Diamond \neg \eta_{kl} \\
&\vee\, \Diamond Q_m \\
\wedge\, &\left(Q_m \Rightarrow \bigwedge\nolimits_{l\in 0..\Xi_m} \Box\Diamond \neg \xi_{ml}\right)
\end{aligned}$$

is a nested GR(1) *property.*

A nested GR(1) property is defined here as a liveness formula. Inserting a liveness formula of this form in a stepwise implication operator ($\stackrel{+}{\rightarrow}$, *WhilePlusHalf*, or some other choice) can yield an open-system property with this type of liveness.

A nested GR(1) property can be regarded as a request-response chain, in analogy to Rabin and Streett chains (parity) [187, p. 13] [178]. Note though that synthesis for a nested GR(1) property is similar to GR(1), whereas for parity games it is unknown whether a polynomial time algorithm exists.

10.3.2 A decomposition algorithm

We propose Algorithm 10.5 [67], which takes a recurrence goal G, and constructs nested GR(1) specifications for the individual components, forming a contract that implies $\Box\Diamond G$. This algorithm assumes that the safety closure of

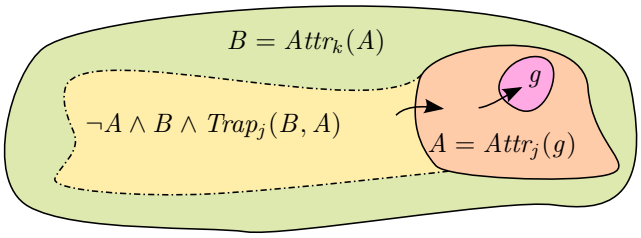


Figure 10.4: Predicates computed by UncondAsm, Algorithm 10.5.

the assembled system has already been taken, in order to constrain the component actions (in the full information setting this is always possible [33, 67]).

Theorem 6 ([67, 65])**.** Assume*: A cooperatively realizable specification* $\varphi = \square\diamond G \wedge \square Next$ *(with component actions that result from closure of* φ*).* Prove*: After* $\mathcal{O}(n\,|V|^2)$ *calls to* CPre_j*, the call* GameStack$(0, G, C, stk)$ *in Algorithm 10.5 returns in stk a contract of nested GR(1) properties (with safety part defined using stepwise implication) that refines* φ*, and each property is realizable by the corresponding component from all states in the cooperatively winning set.*

Proof Sketch: Function UncondAsm finds the largest set r from where player j can keep the play inside the k-attractor of the j-attractor of goal g, as shown in Fig. 10.4. This yields an *unconditional* assumption that player j makes about player k. Iteration l of the **while** loop produces an assumption $\square\diamond\neg\eta_{ml}$ of player j, with $\eta = \mathrm{trap} = r$, which is also a guarantee $\square\diamond\neg\xi_{m'l}$ by player k, with $\xi_{m'l} = \mathrm{trap}$. The fixpoint $\nu : \mu : \mu$ for computing r in UncondAsm has alternation depth 1 [60], so it invokes $\mathrm{CPre}_?$ $\mathcal{O}(|V|)$ times. Due to determinacy [180, 103], and the definition of the cooperatively winning set C, for every n nested calls of UncondAsm from GameStack, at least one call to UncondAsm from within the **for** loop removes a node from *uncovered*. So, GameStack makes $\mathcal{O}(n\,|V|^2)$ calls to $\mathrm{CPre}_?$. A proof can be found in [65].

Concurrent [16] and asynchronous [158] games are special cases of games with partial information, which are not determined, so the inductive argument does not hold in that case.

Example 8. Let us revisit the example of Fig. 10.1, to observe the algorithm's execution. Player 0 wants $\square\diamond G$. The first call to GameStack will call UncondAsm. Player 0 can force a visit to s_6 from the attractor

Algo. 10.5: Construction of nested GR(1) specification, for a single recurrence goal G.

```
def GameStack(j, G, uncovered, stack) :
  trap := TRUE
  goal := G
  stack := ⟨ ⟩
  (* Create unconditional assumptions η_ml *)
  while (¬ ⊨ trap = FALSE) :
    for k ≠ j :
      attr, trap := UncondAsm(j, k, goal)
      goal := attr ∨ trap
      assumptions := assumptions ∪ {⟨k, □◇¬trap⟩}
      if ¬ ⊨ trap = FALSE :
        break
  game := ⟨j, goal, G, assumptions⟩
  stack.append(game)
  uncovered := uncovered ∧ ¬goal
  (* Covered cooperatively winning set? *)
  if ⊨ uncovered = FALSE
    return (* Construct a nested game *)
  GameStack(j ⊕ 1, goal, uncovered, stack)

def UncondAsm(j, k, g) :
  A := Attr_j(g)
  B := Attr_k(A)
  r := ¬A ∧ B ∧ Trap_j(B, A)
  return A, r
```

$A = Attr_0(s_6) = s_5 \vee s_6$. Player 1 can force A from $B = Attr_1(A) = s_4 \vee s_5 \vee s_6$. But $r =$ FALSE, because player 1 can escape to s_1. So, a nested game is constructed over $s_0 \vee s_1 \vee s_2 \vee s_3 \vee s_4$, with player 1 wanting $\diamond(s_5 \vee s_6)$. In the nested game, $A = Attr_1(s_5 \vee s_6) = s_4 \vee s_5 \vee s_6$. The attractor $B = Attr_0(s_4 \vee s_5 \vee s_6) =$ TRUE, and player 0 can keep player 1 in there, until player 0 visits $s_4 \vee s_5 \vee s_6$. So, in the nested game, player 1 makes the assumption $\square\diamond\neg(s_0 \vee s_1 \vee s_2 \vee s_3)$. This covers the cooperative winning set, which in this example is the entire game graph. □

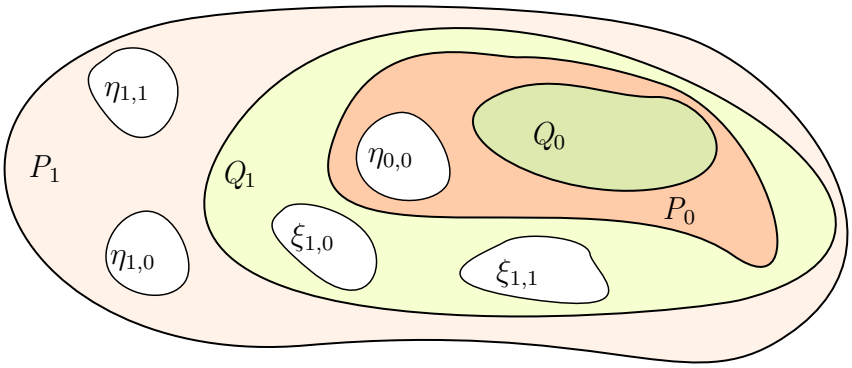


Figure 10.6: An example of a chain condition, with states satisfying each predicate depicted by patches. Containment means implication, e.g. $\models Q_0 \Rightarrow P_0$. Note that Q_0 is the desired recurrence goal, so the algorithm never produces any ξ inside Q_0, because we have arrived at the goal.

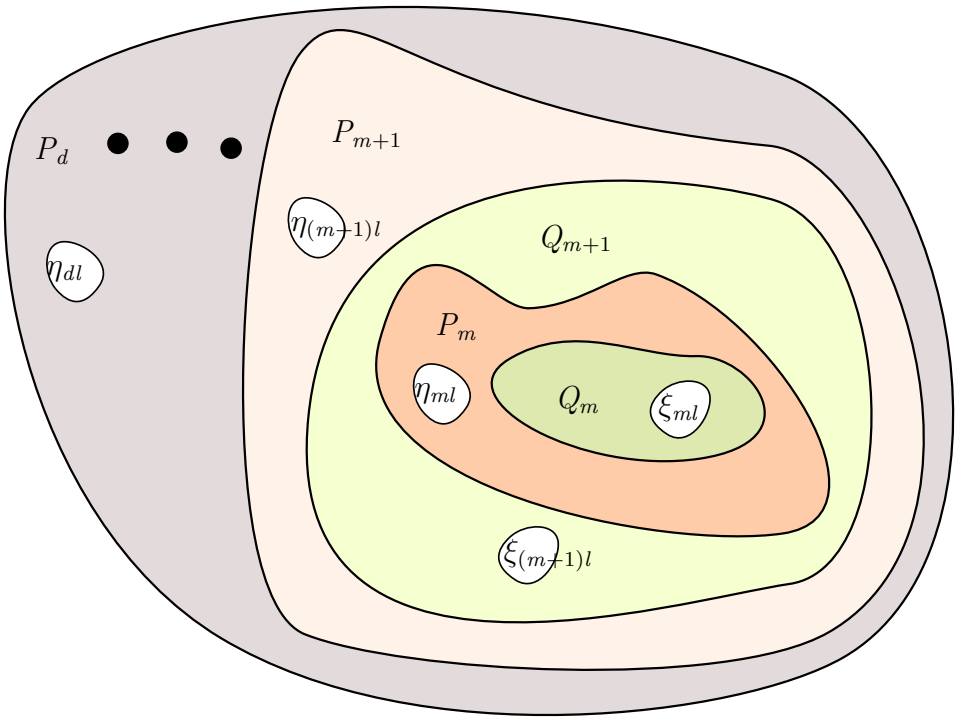


Figure 10.7: A chain condition schematically (Fig. 10.6 is a particular instance).

10.3.3 Equivalence to GR(1) with linear memory increase via auxiliary variables

We prove that nested GR(1) properties can be expressed as GR(1) properties using an auxiliary hidden variable that ranges over at most $d+2$ values, where d in the nesting depth, which is typically small (in the worst case d is bounded by the number of states). This result shows that the original algorithm still produces GR(1) properties, which preserve all safe behaviors (although not all realizable).

Theorem 7 (Flattening nested GR(1) to GR(1))**.** *Assume that φ is a nested GR(1) formula, and σ a behavior. Prove*

$$
\begin{aligned}
(\sigma \models \varphi) \equiv \sigma \models \exists p \; : \; & \wedge \; p = d + 1 \\
& \wedge \; \Box(p' = \text{CHOOSE } r \in 0..(d+1) \; : \\
& \qquad \wedge \; \forall m \in 0..d \; : \; \vee \; (r \leq m) \vee Q_m \\
& \qquad\qquad\qquad\qquad \vee \; (p > m) \wedge \neg P_m \\
& \qquad \wedge \; (r \leq d) \Rightarrow \big(\neg Q_r \wedge (P_r \vee (p \leq r))\big)\big) \\
& \wedge \; \vee \; \neg \forall m \in 0..d, l \in 0..H_m \; : \; \Box\Diamond\neg\eta_{ml} \\
& \quad \vee \; \wedge \; \Box\Diamond(p = d+1) \\
& \qquad \wedge \; \forall m \in 0..d, l \in 0..\Xi_m \; : \; \Box\Diamond\neg\xi_{ml}
\end{aligned}
$$

Finally, we can observe that a nested GR(1) property does not constrain safety compared to the given assembly property φ, but the *realizable part* of each component specification can constrain safety. In particular, "going backwards" after entering a set of states P_m would lead outside the attractor of Q_m (by construction via the decomposition algorithm), even though the response $\Diamond Q_m$ is required by a nested GR(1) property after entering P_m (provided the liveness assumptions η_{kl} hold). Therefore, a component realizing such a property would have to avoid such backwards transitions that exit the attractor of Q_m, which is a safety constraint.

Chapter 11

TIME

Time is absent from TLA$^+$ and LTL. Behaviors represent *changes*, not time. (TLA$^+$ could also be called the "logic of changes".) Physical time can be modeled by a variable (that takes discrete or continuous values throughout a behavior) [5, 118, 118], [117, Ch.9].

Why is there no time in *temporal* logic? The main reasons are:

- To abstract away from specifics. If timing is irrelevant to the purpose of a specification, then it clutters it, reduces readability, and increases the likelihood of errors.
- To increase reusability. Reusing a component's design, or replacing one component by another in a deployed assembly should be insensitive to timing, unless time is of essence to the application.
- To enable machines reason to about specifications. Timed specifications involve additional state (in the form of clocks and times) that quickly leads to intractable reasoning problems.
- To simplify refinement within the object logic. Instead of constraining how many steps should occur, TLA$^+$ layers a system into different *time scales*. We need not count steps, instead liveness ensures that changes eventually do happen.

Liveness in temporal logic models time flow in the physical world.

Remark 18. The abstraction into time scales hinges on an implicit assumption: that steps in the faster time scale never sum to an amount of time comparable to the slower time scale. For example, the sum of ALU steps needed for performing some arithmetic operation may in all cases be negligible compared to the time scale of communicating over a network. If this assumption does not hold, then liveness is not a good model, and the two time scales should be merged. The TLA$^+$ approach to time scales is reminiscent of singular perturbation theory [98]. □

We *can* write timed specifications without liveness in TLA$^+$, despite the stutter-invariance of properties, by having any nonstuttering step increment time. However, this approach means that absolute time is tied to change in lock step with changes of variables at a particular level of refinement [111]. This prevents from letting several "fast" nonstuttering steps to occur between changes that happen at a slower pace.

11.1 Who controls time?

Time cannot be a variable controlled by any component under design. Allowing a component to control time enables it to force its environment to violate an assumption, if the assumption contains timing constraints. For example, if the assumption specifies that the environment eventually responds to some requests, and that those responses consume time, then a component that controls time can implement a specification with such an assumption by freezing time. Freezing time freezes the environment, and ensures that the assumption is violated. This meaning is not what we want when specifying timed systems.

Suppose that the environment controls time. In a specification that abstracts time from faster time scales, many untimed steps may be prerequisite for a change that does consume time. The environment can then "race" the clock ahead, by incrementing it in each behavior step, even though those steps are in a "fast" time scale. Until the component has taken all the necessary fast steps, the deadline it was working towards has elapsed. Avoiding this situation requires using a fine-grain time scale, but that increases the computational burden.

There is a distinction between measuring time and counting steps. This distinction gives rise to the above issues of too relaxed and too demanding timed specifications. In LTL, this situation can be avoided by incrementing time in each behavior step. However, doing so precludes easy step refinement, as in TLA$^+$.

Modeling time with stutter-invariant open-system specifications requires using two variables:

1. A time variable controlled by the environment, which can be thought of as modeling physical time.

2. A variable controlled by the system that can *pause* time in the slower time scale.

The situation is symmetric. In summary, some component controls time, but is allowed to advance time *only* under agreement by other components. (There are special cases where time constrains the environment only via safety, not liveness, and in those cases GR(1) specifications still suffice, but in general we need to use GR(2).)

What about verification? Time needs to be an environment variable in the context of synthesis due to the alternation of quantification ($\exists f : \forall x, y, \ldots$). In verification quantification is uniform, so the unrealizability issues discussed above are absent (in other words, in verification the assembled components are treated as a single entity that allows some behaviors; there is no situation there of a component freezing time to hinder other components). One example of timed open-system specifications in a verification context is [5].

11.2 Assuming time does progress

Writing timed assume-guarantee specifications in the presence of alternating quantification is tricky. A first attempt might be

VARIABLES now, env, controlled by the environment
$pause$, sys controlled by the component

. . . specification . . .

$$TickHappens \triangleq \langle \text{TRUE} \rangle_{now}$$
$$UnfreezeTime \triangleq pause = \text{FALSE}$$

$$Se \triangleq EnvInit \wedge \Box EnvNext$$
$$Le \triangleq \Box[\neg pause]_{now} \wedge \Box\Diamond TickHappens$$
$$Env \triangleq Se \wedge Le$$

$$Sm \triangleq SysInit \wedge \Box SysNext$$
$$Lm \triangleq \wedge \Box\Diamond Goal$$
$$\wedge \Box\Diamond UnfreezeTime$$
$$Sys \triangleq Sm \wedge Lm$$

$$WrongSpec \triangleq WhilePlusHalf(Env, Sys)$$

This specification is wrong. Assuming each of the pairs Se, Le and Sm, Lm is machine-closed, we can shift the liveness to the second argument of *WhilePlusHalf* to obtain

$$\begin{array}{lll} WrongL & \triangleq & \vee \Diamond\Box\neg TickHappens \\ & & \vee \ \wedge \Box\Diamond Goal \\ & & \quad \wedge \Box\Diamond UnfreezeTime \end{array}$$

$WrongSpec \equiv WhilePlusHalf(Se,\ Sm \wedge Wrong)$

A component that satisfies $\Box(pause = \text{FALSE})$ does implement this specification. Clearly, *WrongSpec* is not what we want to specify.

Instead, we should require that time flow resumes *unconditionally* of whether time flow, as follows

$$\begin{array}{lll} NewL & \triangleq & \wedge \Box\Diamond UnfreezeTime \\ & & \wedge \ \vee \neg\Box\Diamond TickHappens \\ & & \quad \vee \Box\Diamond Goal \end{array}$$

$$NewLe \triangleq (\Box\Diamond UnfreezeTime) \Rightarrow \Box\Diamond TickHappens$$

$$NewLm \triangleq \Box\Diamond Goal \wedge \Box\Diamond UnfreezeTime$$

PROPOSITION $NewL \equiv (NewLe \Rightarrow NewLm)$

$Spec \triangleq WhilePlusHalf(Se,\ Sm \wedge NewL)$

The liveness formula *NewL* is in GR(2), not GR(1). Again assuming machine closure as needed, we can shift the antecedent to the first argument to obtain the specification

$Spec \equiv WhilePlusHalf(Se \wedge NewLe,\ \ Sm \wedge NewLm)$

The above reasoning shows that in general timed, stutter-invariant open-system specifications need to include GR(2) liveness formulas. As a consequence, synthesis from such specifications is more expensive, but nevertheless low in the temporal hierarchy.

Chapter 12

CONCLUSIONS

We developed an approach for decomposing the temporal logic specification of an assembled system to open-system component specifications that form a contract, in the sense that they are implementable and conjoined imply the given overall system specification. The decomposition approach relies on generating liveness requirements for individual components in a way that leads to acyclic dependencies. In the context of full information, we developed an algorithm that decomposes any given GR(1) property into GR(1) properties with auxiliary variables added, or equivalently to nested GR(1) properties. 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. We investigated under what conditions GR(1) contracts exist, and showed that decomposition of a GR(1) property into a contract of GR(1) properties in general leads to introducing safety constraints. We implemented these algorithms in Python and Cython packages that we developed.

We formalized realizability in TLA^+, and used it to define formally the notion of a contract between components. We defined a closed-system property as one that implies a type invariant that bounds all variables of interest, and an open-system property as one that is not closed. We observed that open-system properties defined using the "while-plus" operator of TLA^+ are unrealizable with our definition of realizability, when not expected. To ensure realizability in those cases, we defined a variant of the "while-plus" operator, called *WhilePlusHalf*. Using the operator *WhilePlusHalf*, we defined an operator for forming open-systems from closed-systems, called *Unzip*. The *Unzip* operator has desirable properties related to avoiding accidental vacuity. We studied stepwise implication operators from the literature, and established relations between forms with and without temporal quantification, the latter expressed in raw TLA^+ with past operators. We showed that in symmetric systems, Moore components specified by strictly causal stepwise implication ensure that circularity is avoided.

We decompiled the generated specifications from BDDs by using a symbolic minimal covering algorithm originally proposed for two-level logic minimization. We specified in TLA$^+$ the cyclic core computation of this algorithm, and proved that the specification implies safety properties of interest using a proof assistant. We implemented this minimal covering algorithm for the case of integer variables. In addition, we specified and proved correct an algorithm for constructing the set of minimal covers of the input covering problem from the set of minimal covers of the cyclic core.

Future directions One direction for future investigation is reducing the amount of information that is visible to other components in the contracts that result from decomposition. It is conceivable that the structure of a specification can be utilized to tractably identify cases where more variables can be hidden. We described how noninterleaving specifications can be handled. Noninterleaving, together with hiding can be used to relax the scheduling assumption to specify asynchronous interaction.

The parametric analysis relied on constant parameters. This choice was motivated by the aim to eliminate variables from specifications, by not mentioning them at all. The same computational approach can be used to design communication strategies that select when to send information, by letting the constants be variables controlled by components. The strategies can then take into account factors other than feasibility, for example a cost of communication.

Another direction is to find whether fewer liveness assumptions suffice. The current version of the algorithm computes as many traps as it can find, and each trap is a greatest fixpoint. It may be possible to use smaller traps and still obtain a contract. One guiding criterion for selecting among different contracts is whether some of them are expressible with more readable specifications than others.

The minimal covering problem bridges the gap between semantic methods and humans, enabling automation without rendering the result unreadable. Besides the criterion of minimality, there is also the problem of structuring a specification. As the planar syntax of TLA demonstrates, structuring mechanisms play an important role, so identifying structure automatically is key for humans to understand what a machine computes.

Appendix A

SUPPLEMENTAL MATERIAL: PROOFS AND METHODS

The theorems and proofs in support of this thesis are organized as appendices in separate documents, available at: `http://resolver.caltech.edu/CaltechTHESIS:07202018-115217471`, and organized as follows:

- Modules related to temporal logic and stepwise implication (relevant to Chapter 9, Chapter 4, and Chapter 3). This appendix includes the modules:

  TLASemantics
  TemporalLogic
  TemporalQuantification
  WhilePlusTheorems (Section 9.1.2)
  WhilePlusHalfTheorems (Secs. 9.3, 4.2.2)
  UnzipTheorems (Sections 4.2.2 and 9.4.2)
  Realizability (Section 3.3)
  HistoryIsRealizable (Section 9.7)
  Representation (Section 9.2)
  StepComparison (Section 9.6).

- Modules related to minimal covering, and the cyclic core computation (Chapter 7). This appendix includes the modules:

  FiniteSetFacts *Lattices* *CyclicCore*
  Optimization *MinCover* *StrongReduction*.

  The proofs in these modules have been checked with the proof assistant TLAPS (version 1.4.3[1]) [35], in the presence of the theorem prover Zenon (v0.7.2) [24], the SMT solver CVC3 (v2.4.1) [19], the proof assistant Isabelle (v2011-1) [145], and the propositional linear temporal logic prover LS4 (commit `c5e907eb3be9d454b3365e747c05100bdffa939c`) [176].

- Proofs about the relation between nested GR(1) properties and GR(1) properties (Section 10.3.3).

Other proofs relevant to Chapter 10 can be found in [65]. Proofs relevant to defining realizability (Section 3.3) can be found in [66].

[1] TLAPS version 1.4.3 is available at: `http://tla.msr-inria.inria.fr/tlaps/dist/current/tlaps-1.4.3.tar.gz`.

Appendix B

MISCELLANEOUS NOTES

B.1 History-determined variables as modal witnesses

Synthesis can be viewed as finding the "value" of a history-determined variable. In non-modal logic a witness for existential quantification is a single value. In analogy, a witness for temporal quantification is an infinite sequence of values. *One* way to describe this sequence is by a function that is applied recursively (the recursion loop closes via the history-determined variable itself).

The form of static synthesis in absence of bound variables other than the existentially quantified variable y is:

$$\exists y \,:\, P(y)$$

and a constant value c is the (rigid) witness. Similarly, for temporal quantification:

$$\mathbf{\exists}\, y \,:\, P(y)$$

the "modal witness" is a function $f[y]$. This function allows us to "embed" the witness in the object theory itself.

$$\mathbf{\exists}\, y \,:\, (y = y_0) \wedge \Box[y' = f[y]]_y \wedge P(y)$$

There is no value in TLA$^+$ that spans multiple steps. Using a function f in this way compensates for this absence. This limitation does not exist in the metatheory. There, we *can* write an infinite sequence of values for a variable, by defining a behavior.

B.2 Where are the bounded quantifiers ?

Consider a quantifier-free formula *Spec* that specifies a system. We have been talking about boundedness in the sense of Δ_0, via a type invariant conjunct within *Spec*. But where are the bounded quantifiers?

Bounded quantification arises when reasoning about the properties implied by *Spec*. Suppose that we can prove the following theorem.

THEOREM $TypeOK \triangleq Spec \Rightarrow TypeInv$

where *TypeInv* is a type invariant that bounds all the declared variables that occur in *Spec*. Suppose that next we want to prove that the implication

$$Spec \Rightarrow Prop$$

is valid. This is a formula with free variables. It is equivalent to the sentence

$$\forall\, x, y, \ldots, z \;:\; Spec \Rightarrow Prop$$

where $x, y, \ldots, z$ are all the variable identifiers that are declared in the context where the formula $Spec \Rightarrow Prop$ appears.

THEOREM
  $Spec \Rightarrow Prop$
PROOF
$\langle 1 \rangle 1$. $Spec \Rightarrow TypeInv$
  BY *TypeOK*
$\langle 1 \rangle 2$. $Spec \equiv (Spec \land TypeInv)$
  BY $\langle 1 \rangle 1$
$\langle 1 \rangle 3$. SUFFICES $(Spec \land TypeInv) \Rightarrow Prop$
  BY $\langle 1 \rangle 2$
$\langle 1 \rangle 4$. SUFFICES $TypeInv \Rightarrow (Spec \Rightarrow Prop)$
  BY $\langle 1 \rangle 3$
$\langle 1 \rangle 5$. SUFFICES $\forall\, x,\ y,\ z :\ TypeInv \Rightarrow (Spec \Rightarrow Prop)$
  BY $\langle 1 \rangle 4$ DEF *Spec*, *Prop* universal closure
  x, y, z should be all declared variables
$\langle 1 \rangle$ QED
  OMITTED The goal from step $\langle 1 \rangle 5$ is a sentence with bounded quantifiers.

So boundedness is relevant to the sentence form of temporal claims that we want to prove. This is the modal analog of bounded universal rigid quantification.

TLA$^+$ does not include bounded temporal quantification (a form of bounded modal quantification). Wondering how bounded temporal quantifiers would look like if present, we arrive at the following form by analogy to bounded rigid quantification

$\forall\, x \in S :\ P$

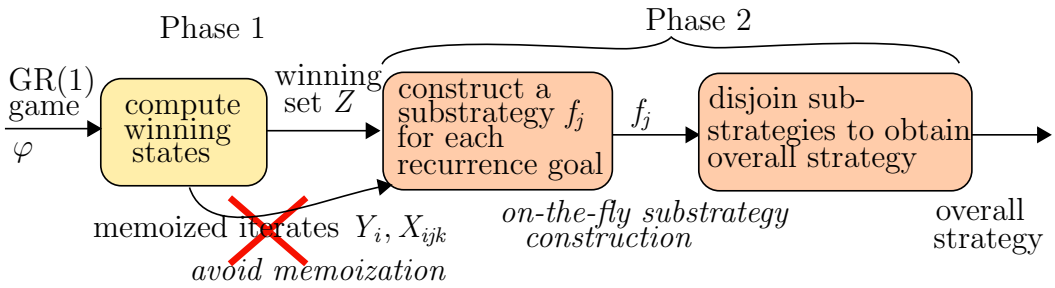


Figure B.1: Phases of solving a GR(1) game. The main changes proposed here are to avoid memoization of fixpoint iterates in phase 1, and construct substrategies on-the-fly in phase 2, to avoid memoization also in phase 2.

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

$$\forall x \in S,\, y \in Q : \ P$$
$$\triangleq \ \forall x \in S : \ \forall y \in Q : \ P$$
$$\equiv \forall x,\, y : \ \Box((x \in S) \wedge (y \in Q)) \ \Rightarrow \ P$$

This form is exactly what role a type invariant plays.

Interestingly, $\exists f \ : \ \forall x, y \ : \ \mathit{Realization}(f, x, y) \Rightarrow \mathit{WPH}(x, y)$ is a bounded formula, because $\exists$ is bounded to a type invariance constraint (within the guarantee), and $\forall$ to a negated type invariance constraint (within the assumption).

B.3 Improvements on efficiency of symbolic synthesis algorithms

A puzzling question about symbolic implementations of synthesis is why widly varying behavior is observed for benchmarks of different sizes [23]. Motivated by observations while developing a language for describing synthesis problems [68] led to experimenting further, and to developing a solver that scales well with the number of recurrence goals in a given GR(1) problem. The main conclusions from this effort are summarized below.

B.3.1 Changes

The modifications applied to the synthesis algorithm have been driven primarily by the following observations:

- Reordering is essential in synthesis, especially during assembly of substrategies to construct the overall solution.

- In practice with CUDD [174], reordering by sifting is effective only for BDD managers with relatively few nodes. A manager with millions of nodes can lead to hours of reordering, and little active time of non-reordering computation. Besides, for managers with large BDDs, reordering can have highly variable duration [195].
- Recomputing the same results can be expensive, but overall faster than reordering of a manager with large BDDs.
- Deferring interaction between BDDs of different parts of a problem can keep variables less coupled. In other words, a manager with many smaller BDDs is observed to hinder reordering less than a manager with a few larger BDDs.

Based on the above, we applied the following design principles (listed in thematic order):

- Avoid BDD memoization when possible.
  1. No memoization in phase 1, as in GR1C [129].
  2. On-the-fly strategy construction from iterates in phase 2.
- Dereference BDDs as informed by profiling.
- Dynamic reordering enabled during strategy construction [68].
- Group sifting as reordering algorithm.
- Conjoin and abstract simultaneously (ANDABSTRACT, hereafter denoted by $\wedge_\exists$) [174].
- Isolation: use separate BDD managers for the fixpoint computation and strategy assembly. One motivation is to isolate the manager where controllable predecessors are computed (and cause high peak compared to total nodes in a CUDD manager) from the manager where strategies are assembled.
- Defer interaction of substrategies, by disjoining them at the end.
- Factor out the system's action *SysNext* during strategy construction.

Below, we discuss some of these changes in more detail.

B.3.2 Avoiding memoization

Except for the solver GR1C, other tools memoize iterates during phase 1. We do not memoize those iterates in phase 1. We also avoid accumulating iterates in phase 2. Instead, we construct each recurrence strategy on-the-fly, meaning that iterates are disjoined to the strategy as soon as they are computed, and dereferenced immediately afterward. Otherwise significantly more time is spent reordering, rather than for BDD computations.

B.3.3 On-the-fly substrategy construction

We avoid memoization in phase 2. Instead of accumulating the fixpoint iterates, and then constructing each substrategy, we assemble each substrategy as the iterates are being computed. At the level of program statements, we have rearranged the operations to reduce how many nodes need to be referenced simultaneously within the BDD manager of phase 2.

B.3.4 Multiple BDD managers

We use two BDD managers:

1. Manager 1 is used exclusively for the fixpoint computation, and is where AndExists is computed.

2. Manager 2 is used for assembling slices of substrategies to obtain each substrategy, storing substrategies (i.e., strategies associated with individual recurrence goals R_i), and finally disjoining the accumulated substrategies to obtain the final strategy.

Using two managers involves transfers of BDDs. Our experiments showed that no significant difference is caused by these transfers, similarly to an observation in [42]. The transfer times are small, compared to AndExists (which takes 80% of total runtime).

Maintaining two managers increases overhead, but isolates iterate computations in phases 1 and 2 from the ongoing strategy construction.

B.3.4.1 Simultaneous conjunction and quantification twice

BDD operations can be implemented as a sequence of more elementary operations, or in one recursive pass. The latter approach is more efficient, because

several large intermediary BDDs are never created. Simultaneous conjunction and quantification (AndAbstract in Cudd [174], denoted $\wedge_\exists$) implements $\exists x : u \wedge v$ in one pass. We apply AndAbstract twice for computing the controllable step operator as follows (x and y denote environment and system variables), and T the target set) $CPre(Target) \triangleq (\exists y' : SysNext(x, y, y') \wedge \forall x' : (EnvNext(x, y, x') \Rightarrow T(x', y'))) \equiv (SysNext(x, y, y') \wedge_{\exists y'} \neg(EnvNext(x, y, x') \wedge_{\exists x'} \neg Target))$.

B.4 Proof of Theorem 1

Proof:

$\langle 1 \rangle 1.$ $IsRealizable_2(\Box\Diamond\neg D)$

$\langle 2 \rangle 1.$ $IsRealizable_2(\Box(D \Rightarrow \Diamond(U \vee Out)))$

$\langle 3 \rangle 1.$ DEFINE $Z \triangleq Attr_2(U \vee Out)$

$\langle 3 \rangle 2.$ $D \Rightarrow Z$

BY DEF D

$\langle 3 \rangle 3.$ $IsRealizable_2(\Box(Z \Rightarrow \Diamond(U \vee Out)))$

BY DEF $Attr$

$\langle 3 \rangle 4.$ QED

BY $\langle 3 \rangle 2$, $\langle 3 \rangle 3$

$\langle 2 \rangle 2.$ $(U \vee Out) \Rightarrow \neg D$

$\langle 3 \rangle 1.$ $D \Rightarrow \neg Out$

$\langle 4 \rangle 1.$ $Basin \Rightarrow \neg Out$

BY DEF Out

$\langle 4 \rangle 2.$ $D \Rightarrow Basin$

BY DEF D

$\langle 4 \rangle 3.$ QED

BY $\langle 4 \rangle 1$, $\langle 4 \rangle 2$

$\langle 3 \rangle 2.$ $D \Rightarrow \neg U$

BY DEF D

$\langle 3 \rangle 3.$ QED

BY $\langle 3 \rangle 1$, $\langle 3 \rangle 2$

$\langle 2 \rangle 3.$ $IsRealizable_2(\Box(D \Rightarrow \Diamond\neg D))$

BY $\langle 2 \rangle 1$, $\langle 2 \rangle 2$

$\langle 2 \rangle 4.$ $\Box(D \Rightarrow \Diamond\neg D) \equiv \Box\Diamond\neg D$

PROOF:

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

$\langle 2\rangle 5$. QED
BY $\langle 2\rangle 3$, $\langle 2\rangle 4$

$\langle 1\rangle 2$. $IsRealizable_1(\Box \lor \neg(T \lor A) \lor \Diamond A \lor \Diamond\Box T)$
$\langle 2\rangle 1$. DEFINE $Z \triangleq Trap_1(Stay, A)$
$\langle 2\rangle 2$. $IsRealizable_1(\Box(Z \Rightarrow \lor \Diamond A \lor \Diamond\Box(Z \land \neg A)))$
BY DEFS Z, $Trap$
$\langle 2\rangle 3$. $T \equiv Z \land \neg A$
BY DEFS T, Z
$\langle 2\rangle 4$. $(T \lor A) \Rightarrow Z$
$\langle 3\rangle 1$. $(T \lor A) \Rightarrow ((Z \land \neg A) \lor A)$
BY $\langle 2\rangle 3$
$\langle 3\rangle 2$. $(T \lor A) \Rightarrow (Z \lor A)$
BY $\langle 3\rangle 1$
$\langle 3\rangle 3$. $(Z \lor A) \Rightarrow Z$
$\langle 4\rangle 1$. $A \Rightarrow Z$
BY DEF Z, $Trap$
$\langle 4\rangle 2$. QED
BY $\langle 4\rangle 1$
$\langle 3\rangle 4$. QED
BY $\langle 3\rangle 2$, $\langle 3\rangle 3$
$\langle 2\rangle 5$. QED
BY $\langle 2\rangle 2$, $\langle 2\rangle 3$, $\langle 2\rangle 4$

$\langle 1\rangle 3$. $(Inv \land T) \Rightarrow D$
$\langle 2\rangle 1$. $T \equiv Trap_1(Stay, A) \land \neg A$
BY DEF T
$\langle 2\rangle 2$. $Trap_1(Stay, A) \Rightarrow (Stay \lor A)$
BY DEF $Trap$
$\langle 2\rangle 3$. $T \Rightarrow (Stay \land \neg A)$
BY $\langle 2\rangle 1$, $\langle 2\rangle 2$
$\langle 2\rangle 4$. $(Inv \land Stay) \Rightarrow D$

BY DEFS $Stay, Obs_1$

⟨2⟩5. QED

BY ⟨2⟩3, ⟨2⟩4

⟨1⟩4. QED

BY ⟨1⟩1, ⟨1⟩2, ⟨1⟩3

BIBLIOGRAPHY

[1] 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. DOI: 10.1016/0020-0190(91)90168-H [p. 121]

[2] M. Abadi and L. Lamport, "The existence of refinement mappings," *TCS*, vol. 82, no. 2, pp. 253–284, 1991. DOI: 10.1016/0304-3975(91)90224-P [pp. 36, 160]

[3] M. Abadi and L. Lamport, "An old-fashioned recipe for real time," in *Real-Time: Theory in Practice*. Springer, 1992, pp. 1–27. DOI: 10.1007/BFb0031985 [pp. 159, 160]

[4] M. Abadi and L. Lamport, "Composing specifications," *TOPLAS*, vol. 15, no. 1, pp. 73–132, 1993. DOI: 10.1145/151646.151649 [p. 160]

[5] M. Abadi and L. Lamport, "An old-fashioned recipe for real time," *TOPLAS*, vol. 16, no. 5, pp. 1543–1571, 1994. DOI: 10.1145/186025.186058 [pp. 154, 156, 170, 172]

[6] M. Abadi and L. Lamport, "Open systems in TLA," in *PODC*, 1994, pp. 81–90. DOI: 10.1145/197917.197960 [pp. 7, 29]

[7] M. Abadi and L. Lamport, "Conjoining specifications," *TOPLAS*, vol. 17, no. 3, pp. 507–535, 1995. DOI: 10.1145/203095.201069 [pp. 6, 9, 12, 30, 33, 34, 36, 47, 120, 132]

[8] M. Abadi, L. Lamport, and P. Wolper, "Realizable and unrealizable specifications of reactive systems," in *ICALP*, 1989, pp. 1–17. DOI: 10.1007/BFb0035748 [p. 19]

[9] M. Abadi, F. McSherry, D. G. Murray, and T. L. Rodeheffer, "Formal analysis of a distributed algorithm for tracking progress," in *Formal Techniques for Distributed Systems*, 2013, pp. 5–19. DOI: 10.1007/978-3-642-38592-6_2 [p. 71]

[10] M. Abadi and S. Merz, "An abstract account of composition," in *Int. Symp. on Mathematical Foundations of Computer Science (MFCS)*, 1995, pp. 499–508. DOI: 10.1007/3-540-60246-1_155 [pp. 30, 32, 47]

[11] 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: Computer and Systems Sciences, vol. 152. Springer, 1996, pp. 235–272. Available at: https://members.loria.fr/SMerz/papers/mod94.html [pp. 7, 16, 30, 31, 32, 34, 36, 47, 107, 134]

[12] B. Alpern and F. B. Schneider, "Defining liveness," *IPL*, vol. 21, no. 4, pp. 181–185, 1985. DOI: 10.1016/0020-0190(85)90056-0 [pp. 16, 36, 107]

[13] B. Alpern and F. B. Schneider, "Recognizing safety and liveness," *Distributed Computing*, vol. 2, no. 3, pp. 117–126, 1987. DOI: 10.1007/BF01782772 [p. 36]

[14] R. Alur, L. de Alfaro, T. A. Henzinger, and F. Y. C. Mang, "Automating modular verification," in *CONCUR*, 1999, pp. 82–97. DOI: 10.1007/3-540-48320-9_8 [p. 11]

[15] R. Alur and T. A. Henzinger, "Reactive modules," *FMSD*, vol. 15, no. 1, pp. 7–48, 1999. DOI: 10.1023/A:1008739929481 [pp. 7, 10]

[16] R. Alur, T. A. Henzinger, and O. Kupferman, "Alternating-time temporal logic," *JACM*, vol. 49, no. 5, pp. 672–713, 2002. DOI: 10.1145/585265.585270 [pp. 11, 33, 166]

[17] R. Alur, S. Moarref, and U. Topcu, "Counter-strategy guided refinement of GR(1) temporal logic specifications," in *FMCAD*, 2013, pp. 26–33. DOI: 10.1109/FMCAD.2013.6679387 [p. 12]

[18] R. Alur, S. Moarref, and U. Topcu, "Pattern-based refinement of assume-guarantee specifications in reactive synthesis," in *TACAS*, 2015, pp. 501–516. DOI: 10.1007/978-3-662-46681-0_49 [p. 12]

[19] C. Barrett and C. Tinelli, "CVC3," in *CAV*, 2007, pp. 298–302. DOI: 10.1007/978-3-540-73368-3_34. Available at: https://cs.nyu.edu/acsys/cvc3/ [p. 176]

[20] S. S. Bauer, A. David, R. Hennicker, K. G. Larsen, A. Legay, U. Nyman, and A. Wąsowski, "Moving from specifications to contracts in component-based design," in *FASE*, 2012, pp. 43–58. DOI: 10.1007/978-3-642-28872-2_3 [pp. 9, 26]

[21] 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 system design," *Foundations and Trends® in Electronic Design Automation*, vol. 12, no. 2–3, pp. 124–400, 2018. DOI: 10.1561/1000000053 [pp. 8, 26]

[22] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K. Larsen, "Contracts for systems design," INRIA, Tech. Rep. 8147, 2012. Available at: https://hal.inria.fr/hal-00757488 [pp. 8, 9, 26]

[23] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," *JCSS*, vol. 78, no. 3, pp. 911–938, 2012. DOI: 10.1016/j.jcss.2011.08.007 [pp. 20, 21, 31, 32, 33, 55, 159, 179]

[24] R. Bonichon, D. Delahaye, and D. Doligez, "Zenon: An extensible automated theorem prover producing checkable proofs," in *LPAR*, 2007, pp. 151–165. DOI: 10.1007/978-3-540-75560-9_13. Available at: http://zenon.inria.fr [p. 176]

[25] F. Boniol, V. Wiels, Y. Ait Ameur, and K.-D. Schewe, Eds., *ABZ 2014: The landing gear case study.* Springer, 2014. DOI: 10.1007/978-3-319-07512-9 [p. 94]

[26] 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. DOI: 10.1007/978-3-642-31374-5_3 [p. 2]

[27] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic minimization algorithms for VLSI synthesis.* Kluwer, 1984. DOI: 10.1007/978-1-4613-2821-6 [p. 67]

[28] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. Gritzner, and R. Weber, "The design of distributed systems—An introduction to FOCUS," Technische Universität München, Tech. Rep., 1992. [p. 10]

[29] M. Broy and K. Stølen, *Specification and development of interactive systems: Focus on streams, interfaces, and refinement.* Springer, 2001. DOI: 10.1007/978-1-4613-0091-5 [p. 10]

[30] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *TC*, vol. 35, no. 8, pp. 677–691, 1986. DOI: 10.1109/TC.1986.1676819 [pp. 4, 23, 66]

[31] 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. DOI: 10.1109/12.73590 [p. 66]

[32] K. Chatterjee and T. A. Henzinger, "Assume-guarantee synthesis," *TACAS*, pp. 261–275, 2007. DOI: 10.1007/978-3-540-71209-1_21 [p. 11]

[33] K. Chatterjee, T. A. Henzinger, and B. Jobstmann, "Environment assumptions for synthesis," in *CONCUR*, 2008, pp. 147–161. DOI: 10.1007/978-3-540-85361-9_14 [pp. 12, 37, 166]

[34] K. Chatterjee, T. A. Henzinger, J. Otop, and A. Pavlogiannis, "Distributed synthesis for LTL fragments," in *FMCAD*, 2013, pp. 18–25. DOI: 10.1109/FMCAD.2013.6679386 [p. 11]

[35] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz, "A TLA$^+$ proof system," in *Proceedings of the LPAR Workshops, Knowledge Exchange: Automated Provers and Proof Assistants (KEAPPA) Workshop*, vol. 418, 2008, pp. 17–37. Available at: http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-418/paper2.pdf [pp. 71, 176]

[36] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz, "Verifying safety properties with the TLA$^+$ proof system," in *IJCAR*, 2010, pp. 142–148. DOI: 10.1007/978-3-642-14203-1_12 [p. 71]

[37] A. Cimatti, M. Dorigatti, and S. Tonetta, "OCRA: A tool for checking the refinement of temporal contracts," in *ASE*, 2013, pp. 702–705. DOI: 10.1109/ASE.2013.6693137 [p. 8]

[38] A. Cimatti and S. Tonetta, "A property-based proof system for contract-based design," in *EUROMICRO*, 2012, pp. 21–28. DOI: 10.1109/SEAA.2012.68 [p. 8]

[39] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 1999. [pp. 4, 66]

[40] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu, "Learning assumptions for compositional verification," in *TACAS*, 2003, pp. 331–346. DOI: 10.1007/3-540-36577-X_24 [p. 12]

[41] A. Cohen and K. S. Namjoshi, "Local proofs for global safety properties," *FMSD*, vol. 34, no. 2, pp. 104–125, 2009. DOI: 10.1007/s10703-008-0063-8 [p. 12]

[42] A. Cohen, K. S. Namjoshi, Y. Sa'ar, L. D. Zuck, and K. I. Kisyova, "Parallelizing a symbolic compositional model-checking algorithm," in *HVC*, 2010, pp. 46–59. DOI: 10.1007/978-3-642-19583-9_9 [p. 181]

[43] O. Coudert and J. C. Madre, "Implicit and incremental computation of primes and essential primes of Boolean functions," in *DAC*, 1992, pp. 36–39. DOI: 10.1109/DAC.1992.227866 [p. 67]

[44] O. Coudert, "Two-level logic minimization: An overview," *Integration, the VLSI Journal*, vol. 17, no. 2, pp. 97–140, 1994. DOI: 10.1016/0167-9260(94)00007-7 [pp. 4, 67, 68, 70, 76, 78, 79, 83, 92]

[45] 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. DOI: 10.1145/157485.165071 [p. 67]

[46] 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. Available at: http://sasimi.jp [p. 67]

[47] O. Coudert and J. C. Madre, "New ideas for solving covering problems," in *DAC*, 1995, pp. 641–646. DOI: 10.1145/217474.217603 [p. 67]

[48] D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, and H. Vanzetto, "TLA$^+$ proofs," in *Formal Methods (FM)*, 2012, pp. 147–154. DOI: 10.1007/978-3-642-32759-9_14 [p. 71]

[49] L. de Alfaro and M. Faella, "Information flow in concurrent games," in *ICALP*, 2003, pp. 1038–1053. DOI: 10.1007/3-540-45061-0_80 [p. 11]

[50] L. de Alfaro and T. A. Henzinger, "Interface automata," in *ESEC/FSE*, 2001, pp. 109–120. DOI: 10.1145/503271.503226 [pp. 8, 40]

[51] L. de Alfaro, T. A. Henzinger, and F. Y. C. Mang, "The control of synchronous systems," in *CONCUR*, 2000, pp. 458–473. DOI: 10.1007/3-540-44618-4_33 [pp. 11, 21, 33, 149, 157]

[52] M. de Wulf, L. Doyen, and J.-F. Raskin, "A lattice theory for solving games of imperfect information," in *HSCC*, 2006, pp. 153–168. DOI: 10.1007/11730637_14 [pp. 3, 11]

[53] F. Dederichs and R. Weber, "Safety and liveness from a methodological point of view," *IPL*, vol. 36, no. 1, pp. 25–30, 1990. DOI: 10.1016/0020-0190(90)90181-V [p. 121]

[54] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *CACM*, vol. 8, no. 9, p. 569, 1965. DOI: 10.1145/365559.365617 [pp. 33, 149]

[55] I. Dillig, T. Dillig, K. L. McMillan, and A. Aiken, "Minimum satisfying assignments for SMT," in *CAV*, 2012, pp. 394–409. DOI: 10.1007/978-3-642-31424-7_30 [p. 13]

[56] H.-D. Ebbinghaus, *Ernst Zermelo: An approach to his life and work*. Springer, 2007. DOI: 10.1007/978-3-540-49553-6 [pp. 14, 19]

[57] R. Ehlers, R. Könighofer, and R. Bloem, "Synthesizing cooperative reactive mission plans," in *IROS*, 2015, pp. 3478–3485. DOI: 10.1109/IROS.2015.7353862 [p. 12]

[58] R. Ehlers and V. Raman, "Low-effort specification debugging and analysis," *EPTCS*, vol. 157, pp. 117–133, 2014. DOI: 10.4204/EPTCS.157.12 [p. 13]

[59] R. Ehlers and U. Topcu, "Estimator-based reactive synthesis under incomplete information," in *HSCC*, 2015, pp. 249–258. DOI: 10.1145/2728606.2728626 [pp. 3, 12, 158]

[60] E. A. Emerson and C.-L. Lei, "Model checking in the propositional Mu-calculus," Department of Computer Sciences, University of Texas at Austin, Tech. Rep. TR-86-06, 1986. Available at: http://catalog.lib.utexas.edu/record=b7285480~S29 [p. 166]

[61] I. Filippidis, "Contract construction implementation." Available at: https://github.com/johnyf/contract_maker [p. 104]

[62] I. Filippidis, "`dd`: Decision diagrams (PYTHON package)." Available at: https://github.com/tulip-control/dd [pp. 5, 71]

[63] I. Filippidis, "`omega`: PYTHON package for specification of systems and synthesis of implementations using symbolic algorithms." Available at: https://github.com/tulip-control/omega [pp. 5, 71]

[64] 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. DOI: 10.1109/CCA.2016.7587949 [p. 22]

[65] I. Filippidis and R. M. Murray, "Symbolic construction of GR(1) contracts for synchronous systems with full information," California Institute of Technology, Tech. Rep. arXiv:1508.02705, 2015. [pp. 50, 106, 161, 162, 166, 176]

[66] I. Filippidis and R. M. Murray, "Formalizing synthesis in TLA$^+$," California Institute of Technology, Tech. Rep. CaltechCDSTR:2016.004, 2016. Available at: http://resolver.caltech.edu/CaltechCDSTR:2016.004 [pp. 19, 29, 30, 31, 34, 121, 129, 148, 156, 176]

[67] I. Filippidis and R. M. Murray, "Symbolic construction of GR(1) contracts for systems with full information," in *ACC*, 2016, pp. 782–789. DOI: 10.1109/ACC.2016.7525009 [pp. 24, 32, 37, 48, 50, 94, 106, 165, 166]

[68] I. Filippidis, R. M. Murray, and G. J. Holzmann, "A multi-paradigm language for reactive synthesis," in *4th Workshop on Synthesis (SYNT)*, 2015, pp. 73–97. DOI: 10.4204/EPTCS.202.6 [pp. 68, 179, 180]

[69] B. Finkbeiner and S. Schewe, "Uniform distributed synthesis," in *LICS*, 2005, pp. 321–330. DOI: 10.1109/LICS.2005.53 [p. 11]

[70] B. Finkbeiner and S. Schewe, "Bounded synthesis," *STTT*, vol. 15, no. 5, pp. 519–539, 2013. DOI: 10.1007/s10009-012-0228-z [p. 11]

[71] R. W. Floyd, "Assigning meanings to programs," in *Symposia in Applied Mathematics*, ser. Aspects of Computer Science, vol. 19. AMS, 1967, pp. 19–32. DOI: 10.1090/psapm/019/0235771 [p. 6]

[72] N. Francez and A. Pnueli, "A proof method for cyclic programs," *Acta Informatica*, vol. 9, no. 2, pp. 133–157, 1978. DOI: 10.1007/BF00289074 [p. 6]

[73] J. Fu and U. Topcu, "Integrating active sensing into reactive synthesis with temporal logic constraints under partial observations," in *ACC*, 2015, pp. 2408–2413. DOI: 10.1109/ACC.2015.7171093 [p. 12]

[74] 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.* The Institute for Advanced Study, 1947, vol. 1. Available at: https://library.ias.edu/files/pdfs/ecp/planningcodingof0103inst.pdf [p. 6]

[75] G. D. Hachtel and F. Somenzi, *Logic synthesis and verification algorithms*. Kluwer, 1996. DOI: 10.1007/0-387-31005-3 [p. 66]

[76] B. Hayes and J. A. Shah, "Improving robot controller transparency through autonomous policy explanation," in *Human-Robot Interaction*, 2017, pp. 303–312. DOI: 10.1145/2909824.3020233 [p. 12]

[77] T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "You assume, we guarantee: Methodology and case studies," in *CAV*, 1998, pp. 440–451. DOI: 10.1007/BFb0028765 [p. 10]

[78] T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "Decomposing refinement proofs using assume-guarantee reasoning," in *ICCAD*, 2000, pp. 245–253. DOI: 10.1109/ICCAD.2000.896481 [pp. 6, 10]

[79] D. Hilbert and P. Bernays, *Grundlagen der Mathematik II*. Springer, 1970. DOI: 10.1007/978-3-642-86896-2 [p. 15]

[80] C. A. R. Hoare, "An axiomatic basis for computer programming," *CACM*, vol. 12, no. 10, pp. 576–580, 1969. DOI: 10.1145/363235.363259 [p. 6]

[81] 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. DOI: 10.7873/DATE.2014.167 [p. 9]

[82] A. Iannopollo, S. Tripakis, and A. Sangiovanni-Vincentelli, "Constrained synthesis from component libraries," in *FACS*, 2016, pp. 1–6. DOI: 10.1007/978-3-319-57666-4_7 [p. 9]

[83] A. Iannopollo, S. Tripakis, and A. Sangiovanni-Vincentelli, "Specification decomposition for synthesis from libraries of LTL Assume/Guarantee contracts," in *DATE*, 2018, pp. 1574–1579. DOI: 10.23919/DATE.2018.8342266 [p. 9]

[84] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem, "Anzu: A tool for property synthesis," in *CAV*, 2007, pp. 258–262. DOI: 10.1007/978-3-540-73368-3_29 [p. 66]

[85] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program repair as a game," in *CAV*, 2005, pp. 226–238. DOI: 10.1007/11513988_23 [pp. 21, 160]

[86] C. B. Jones, "Specification and design of (parallel) programs," in *Information Processing*, 1983, pp. 321–332. [pp. 1, 6, 40]

[87] 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, 2003. DOI: 10.1109/MAHC.2003.1203057 [p. 6]

[88] B. Jonsson and Y.-K. Tsay, "Assumption/guarantee specifications in linear-time temporal logic," *TCS*, vol. 167, no. 1, pp. 47–72, 1996. DOI: 10.1016/0304-3975(96)00069-2 [pp. 9, 30, 32, 106, 107]

[89] M. Katara and T. Mikkonen, "Design approach for real-time reactive systems," in *International Workshop on Real-Time Constraints*, 1999. Available at: https://www.cs.ccu.edu.tw/~pahsiung/cp99-rtc/proceedings.html [p. 12]

[90] Y. Kesten, N. Piterman, and A. Pnueli, "Bridging the gap between fair simulation and trace inclusion," *Information and Computation*, vol. 200, no. 1, pp. 35–61, 2005. DOI: 10.1016/j.ic.2005.01.006 [pp. 20, 29, 33, 158]

[91] Y. Kesten and A. Pnueli, "Complete proof system for QPTL," *Journal of Logic and Computation*, vol. 12, no. 5, pp. 701–745, 2002. DOI: 10.1093/logcom/12.5.701 [p. 18]

[92] Y. Kesten, A. Pnueli, and L.-o. Raviv, "Algorithmic verification of linear temporal logic specifications," in *ICALP*, 1998, pp. 1–16. DOI: 10.1007/BFb0055036 [pp. 4, 66]

[93] S. C. Kleene, *Introduction to metamathematics*. North-Holland, 1971. [pp. 25, 34, 45]

[94] U. Klein, "Topics in formal synthesis and modeling," Ph.D. dissertation, Courant Institute of Mathematical Sciences, New York University, 2011. Available at: https://cs.nyu.edu/media/publications/klein_uri.pdf [p. 12]

[95] U. Klein, N. Piterman, and A. Pnueli, "Effective synthesis of asynchronous systems from GR(1) specifications," in *VMCAI*, 2012, pp. 283–298. DOI: 10.1007/978-3-642-27940-9_19 [pp. 3, 12, 18]

[96] U. Klein and A. Pnueli, "Revisiting synthesis of GR(1) specifications," in *HVC*, 2010, pp. 161–181. DOI: 10.1007/978-3-642-19583-9_16 [pp. 9, 30, 32, 105, 111, 119, 121]

[97] D. E. Knuth, "Robert W Floyd, In memoriam," *ACM SIGACT News*, vol. 34, no. 4, pp. 3–13, 2003. DOI: 10.1145/954092.954488 [p. 6]

[98] P. Kokotović, H. K. Khalil, and J. O'Reilly, *Singular perturbation methods in control: Analysis and design*. SIAM, 1999. DOI: 10.1137/1.9781611971118 [p. 170]

[99] R. Könighofer, G. Hofferek, and R. Bloem, "Debugging formal specifications: A practical approach using model-based diagnosis and counterstrategies," *STTT*, vol. 15, no. 5, pp. 563–583, 2013. DOI: 10.1007/s10009-011-0221-y [pp. 12, 33]

[100] K. Kunen, *The foundations of mathematics*, ser. Studies in Logic. College Publications, 2012, vol. 19. [pp. 30, 122, 132]

[101] K. Kunen, *Set theory*, ser. Studies in Logic. College Publications, 2013, vol. 34. [p. 19]

[102] O. Kupferman and M. Y. Vardi, "Safraless decision procedures," in *FOCS*, 2005, pp. 531–540. DOI: 10.1109/SFCS.2005.66 [pp. 3, 11]

[103] O. Kupferman, Y. Lustig, M. Y. Vardi, and M. Yannakakis, "Temporal synthesis for bounded systems and environments," in *STACS*, vol. 9, 2011, pp. 615–626. DOI: 10.4230/LIPIcs.STACS.2011.615 [p. 166]

[104] O. Kupferman and M. Y. Vardi, "Synthesis with incomplete informatio," in *Advances in Temporal Logic*, vol. 16. Springer, 2000, pp. 109–127. DOI: 10.1007/978-94-015-9586-5_6 [p. 11]

[105] R. Kurki-Suonio, "Component and interface refinement in closed-system specifications," in *Formal Methods (FM)*, 1999, pp. 134–154. DOI: 10.1007/3-540-48119-2_10 [pp. 2, 12]

[106] R. P. Kurshan and L. Lamport, "Verification of a multiplier: 64 bits and beyond," in *CAV*, 1993, pp. 166–179. DOI: 10.1007/3-540-56922-7_14 [p. 2]

[107] H. Lamouchi and J. Thistle, "Effective control synthesis for DES under partial observations," in *CDC*, vol. 1, 2000, pp. 22–28. DOI: 10.1109/CDC.2000.912726 [p. 11]

[108] L. Lamport, "Proving the correctness of multiprocess programs," *TSE*, vol. 3, no. 2, pp. 125–143, 1977. DOI: 10.1109/TSE.1977.229904 [p. 7]

[109] L. Lamport, "The "Hoare logic" of concurrent programs," *Acta Informatica*, vol. 14, pp. 21–37, 1980. DOI: 10.1007/BF00289062 [p. 6]

[110] L. Lamport, "Specifying concurrent program modules," *TOPLAS*, vol. 5, no. 2, pp. 190–222, 1983. DOI: 10.1145/69624.357207 [pp. 2, 6, 7, 8, 20, 110]

[111] L. Lamport, "What good is temporal logic?" in *Information Processing*, 1983, pp. 657–668. Available at: http://lamport.azurewebsites.net/pubs/what-good.pdf [pp. 2, 16, 171]

[112] L. Lamport, "A temporal logic of actions," Systems Research Center of Digital Equipment Corporation, Research Report 47, Apr 1990. Available at: https://lamport.azurewebsites.net/pubs/old-tla-src.pdf [pp. 22, 134]

[113] L. Lamport, "Miscellany," 21 April 1991, sent to TLA mailing list. Available at: http://lamport.azurewebsites.net/tla/notes/91-04-21.txt [pp. 2, 20, 107, 160]

[114] L. Lamport, "The temporal logic of actions," *TOPLAS*, vol. 16, no. 3, pp. 872–923, 1994. DOI: 10.1145/177492.177726 [pp. 14, 18, 29, 37, 62, 134]

[115] L. Lamport, "Composition: A way to make proofs harder," in *COMPOS*, 1997, pp. 402–423. DOI: 10.1007/3-540-49213-5_15 [pp. 2, 35]

[116] L. Lamport, "Proving possibility properties," *TCS*, vol. 206, no. 1, pp. 341–352, 1998. DOI: 10.1016/S0304-3975(98)00129-7 [p. 47]

[117] L. Lamport, *Specifying systems: The TLA+ language and tools for hardware and software engineers*. Addison-Wesley, 2002. [pp. 2, 14, 15, 16, 18, 20, 22, 31, 34, 44, 47, 62, 121, 122, 170]

[118] L. Lamport, "Real time is really simple," Microsoft Research, Tech. Rep. MSR-TR-2005-30, 2005. Available at: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2005-30.pdf [p. 170]

[119] L. Lamport, "The PlusCal algorithm language," in *ICTAC*, 2009, pp. 36–60. DOI: 10.1007/978-3-642-03466-4_2 [pp. viii, 60]

[120] L. Lamport, "How to write a 21st century proof," *Journal of fixed point theory and applications*, vol. 11, no. 1, pp. 43–63, 2012. DOI: 10.1007/s11784-012-0071-6 [p. 59]

[121] L. Lamport, "TLA^{+2}: A preliminary guide," Tech. Rep., 15 Jan 2014. Available at: https://lamport.azurewebsites.net/tla/tla2-guide.pdf [pp. 14, 22, 59]

[122] L. Lamport, "Who builds a house without drawing blueprints?" *CACM*, vol. 58, no. 4, pp. 38–41, 2015. DOI: 10.1145/2736348 [p. 1]

[123] L. Lamport and L. C. Paulson, "Should your specification language be typed?" *TOPLAS*, vol. 21, no. 3, pp. 502–526, 1999. DOI: 10.1145/319301.319317 [p. 23]

[124] L. Lamport and F. B. Schneider, "The "Hoare Logic" of CSP, and all that," *TOPLAS*, vol. 6, no. 2, pp. 281–296, 1984. DOI: 10.1145/2993.357247 [p. 6]

[125] T. T. H. Le, R. Passerone, U. Fahrenberg, and A. Legay, "Contract-based requirement modularization via synthesis of correct decompositions," *TECS*, vol. 15, no. 2, pp. 33:1–33:26, 2016. DOI: 10.1145/2885752 [p. 8]

[126] A. C. Leisenring, *Mathematical logic and Hilbert's ε-symbol*. MacDonald Technical & Scientific, 1969. [p. 15]

[127] W. Li, L. Dworkin, and S. A. Seshia, "Mining assumptions for synthesis," in *MEMOCODE*, 2011, pp. 43–50. DOI: 10.1109/MEMCOD.2011.5970509 [p. 12]

[128] O. Lichtenstein, A. Pnueli, and L. Zuck, "The glory of the past," in *Conference on logics of programs*, ser. LNCS, vol. 193. Springer, 1985, pp. 196–218. DOI: 10.1007/3-540-15648-8_16 [pp. 16, 105]

[129] S. C. Livingston, "Incremental control synthesis for robotics in the presence of temporal logic specifications," Ph.D. dissertation, California Institute of Technology, 2016. DOI: 10.7907/Z94Q7RW3. Available at: http://resolver.caltech.edu/CaltechTHESIS:12312015-131513787 [pp. 20, 180]

[130] Z. Manna and A. Pnueli, "A hierarchy of temporal properties," in *PODC*, 1990, pp. 377–410. DOI: 10.1145/93385.93442 [pp. 16, 20]

[131] S. Maoz and J. O. Ringert, "On well-separation of GR(1) specifications," in *FSE*, 2016, pp. 362–372. DOI: 10.1145/2950290.2950300 [pp. 119, 120]

[132] P. C. McGeer, J. V. Sanghavi, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "ESPRESSO-SIGNATURE: A new exact minimizer for logic functions," *TVLSI*, vol. 1, no. 4, pp. 432–440, 1993. DOI: 10.1109/92.250190 [p. 67]

[133] K. L. McMillan, "Circular compositional reasoning about liveness," in *CHARME*, 1999, pp. 342–346. DOI: 10.1007/3-540-48153-2_30 [p. 8]

[134] S. Merz, "From TLT modules to stream processing functions," Institut für Informatik, Technische Universität München, Tech. Rep., Feb 1996. Available at: http://www4.in.tum.de/proj/korsys/public/tltfocus.ps.gz [p. 19]

[135] S. Merz, "Rules for abstraction," in *Advances in Computing Science—ASIAN'97*, 1997, pp. 32–45. DOI: 10.1007/3-540-63875-X_41 [p. 14]

[136] S. Merz and H. Vanzetto, "Harnessing SMT solvers for TLA$^+$ proofs," in *12th International Workshop on Automated Verification of Critical Systems (AVoCS 2012)*, ser. ECEASST, vol. 53. EASST, 2012. DOI: 10.14279/tuj.eceasst.53.766.794 [p. 71]

[137] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, 1992. DOI: 10.1109/2.161279 [p. 8]

[138] 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. DOI: 10.1109/ACC.2014.6859229 [p. 12]

[139] T. Mikkonen, "Abstractions and logical layers in specifications of reactive systems," Ph.D. dissertation, Tampere University of Technology, 1999. [p. 12]

[140] A. Mishchenko, "An introduction to zero-suppressed binary decision diagrams," University of California, Berkeley, Tech. Rep., 2014, see also EXTRA library v1.3. Available at: http://www.eecs.berkeley.edu/~alanmi/research/extra [p. 68]

[141] J. Misra and K. M. Chandy, "Proofs of networks of processes," *TSE*, vol. 7, no. 4, pp. 417–426, 1981. DOI: 10.1109/TSE.1981.230844 [pp. 6, 30]

[142] S. Moarref, "Compositional reactive synthesis for multi-agent systems," Ph.D. dissertation, University of Pennsylvania, 2016. Available at: https://repository.upenn.edu/edissertations/1902 [pp. 12, 44, 62]

[143] 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, 1984. DOI: 10.1109/MAHC.1984.10017 [p. 6]

[144] W. Nam and R. Alur, "Learning-based symbolic assume-guarantee reasoning with automatic decomposition," in *ATVA*, 2006, pp. 170–185. DOI: 10.1007/11901914_15 [p. 12]

[145] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A proof assistant for Higher-Order Logic*. Springer, 2002. DOI: 10.1007/3-540-45949-9. Available at: https://www.cl.cam.ac.uk/research/hvg/Isabelle/ [p. 176]

[146] P. Nuzzo, "Compositional design of cyber-physical systems using contracts," Ph.D. dissertation, EECS Department, University of California, Berkeley, 2015. Available at: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-189.html [p. 8]

[147] P. Nuzzo, A. Iannopollo, S. Tripakis, and A. Sangiovanni-Vincentelli, "Are interface theories equivalent to contract theories?" in *MEMOCODE*, 2014, pp. 104–113. DOI: 10.1109/MEMCOD.2014.6961848 [p. 8]

[148] 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. DOI: 10.1109/ACCESS.2013.2295764 [p. 8]

[149] S. Owicki and L. Lamport, "Proving liveness properties of concurrent programs," *TOPLAS*, vol. 4, no. 3, pp. 455–495, 1982. DOI: 10.1145/357172.357178 [pp. 7, 47]

[150] N. Ozay, U. Topcu, and R. M. Murray, "Distributed power allocation for vehicle management systems," in *CDC*, 2011, pp. 4841–4848. DOI: 10.1109/CDC.2011.6161470 [pp. 12, 20]

[151] N. Ozay, U. Topcu, R. M. Murray, and T. Wongpiromsarn, "Distributed synthesis of control protocols for smart camera networks," in *ICCPS*, 2011, pp. 45–54. DOI: 10.1109/ICCPS.2011.22 [p. 20]

[152] N. Piterman and A. Pnueli, "Faster solutions of Rabin and Streett games," in *LICS*, 2006, pp. 275–284. DOI: 10.1109/LICS.2006.23 [pp. 3, 20]

[153] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," in *VMCAI*, 2006, pp. 364–380. DOI: 10.1007/11609773_24 [pp. 3, 5, 11, 20, 21, 106, 111, 121, 155, 158, 159]

[154] A. Pnueli and R. Rosner, "A framework for the synthesis of reactive modules," in *CONCURRENCY*, 1988, pp. 4–17. DOI: 10.1007/3-540-50403-6_28 [p. 19]

[155] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *POPL*, 1989, pp. 179–190. DOI: 10.1145/75277.75293 [pp. 6, 11, 19]

[156] A. Pnueli, "The temporal logic of programs," in *FOCS*, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32 [p. 6]

[157] 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, vol. 13. Springer, 1985, pp. 123–144. DOI: 10.1007/978-3-642-82453-1_5 [pp. 6, 7, 47]

[158] A. Pnueli and U. Klein, "Synthesis of programs from temporal property specifications," in *MEMOCODE*, 2009, pp. 1–7. DOI: 10.1109/MEMCOD.2009.5185372 [pp. 3, 11, 12, 20, 166]

[159] A. Pnueli and R. Rosner, "Distributed reactive systems are hard to synthesize," in *FOCS*, vol. 2, 1990, pp. 746–757. DOI: 10.1109/FSCS.1990.89597 [pp. 3, 11]

[160] A. Pnueli, Y. Sa'ar, and L. D. Zuck, "JTLV: A framework for developing verification algorithms," in *CAV*, 2010, pp. 171–174. DOI: 10.1007/978-3-642-14295-6_18 [p. 66]

[161] V. Preoteasa, I. Dragomir, and S. Tripakis, "Type inference of Simulink hierarchical block diagrams in Isabelle," in *FORTE*, 2017, pp. 194–209. DOI: 10.1007/978-3-319-60225-7_14 [p. 10]

[162] V. Preoteasa and S. Tripakis, "Refinement calculus of reactive systems," in *EMSOFT*, 2014, pp. 2:1–2:10. DOI: 10.1145/2656045.2656068 [p. 9]

[163] V. Preoteasa and S. Tripakis, "Towards compositional feedback in non-deterministic and non-input-receptive systems," in *LICS*, 2016, pp. 768–777. DOI: 10.1145/2933575.2934503 [p. 10]

[164] V. Raman, "Explaining unsynthesizability of high-level robot behaviors," Ph.D. dissertation, Cornell University, 2013. Available at: http://hdl.handle.net/1813/34373 [p. 20]

[165] G. Rock, W. Stephan, and A. Wolpers, *Modular reasoning about structured TLA specifications*, ser. Advances in Computing Science. Springer, 1999, pp. 217–229. DOI: 10.1007/978-3-7091-6355-9_16 [p. 7]

[166] T. L. Rodeheffer, "The Naiad clock protocol: Specification, model checking, and correctness proof," Microsoft Research, Tech. Rep. MSR-TR-2013-20, Feb 2013. Available at: https://www.microsoft.com/en-us/research/publication/the-naiad-clock-protocol-specification-model-checking-and-correctness-proof/ [p. 71]

[167] R. Rosner, "Modular synthesis of reactive systems," Ph.D. dissertation, Department of Applied Mathematics and Computer Science, Weizmann Institute of Science, 1991. Available at: http://lib-phds1.weizmann.ac.il/Dissertations/rosner_roni.pdf [p. 3]

[168] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *ICCAD*, 1993, pp. 42–47. DOI: 10.1109/ICCAD.1993.580029 [p. 69]

[169] R. L. Rudell, "Logic synthesis for VLSI design," Ph.D. dissertation, EECS Department, University of California, Berkeley, 1989. [p. 67]

[170] A. Sangiovanni-Vincentelli, 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. DOI: 10.3166/ejc.18.217-238 [pp. 8, 9, 26]

[171] S. Schewe and B. Finkbeiner, "Synthesis of asynchronous systems," in *LOPSTR*, 2007, pp. 127–142. DOI: 10.1007/978-3-540-71410-1_10 [p. 11]

[172] F. B. Schneider, *On concurrent programming*. Springer, 1997. DOI: 10.1007/978-1-4612-1830-2 [pp. 6, 16]

[173] S. Singh and M. D. Wagh, "Robot path planning using intersecting convex shapes: Analysis and simulation," *TRO*, vol. RA-3, no. 2, 1987. DOI: 10.1109/ROBOT.1986.1087448 [p. 12]

[174] F. Somenzi, "CUDD: CU Decision Diagram package release 3.0.0," Department of Electrical, Computer, and Energy Engineering, University of Colorado at Boulder, 2016. [pp. 180, 182]

[175] E. W. Stark, "A proof technique for rely/guarantee properties," *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, vol. 206, pp. 369–391, 1985. DOI: 10.1007/3-540-16042-6_21 [pp. 7, 8, 47]

[176] M. Suda and C. Weidenbach, "A PLTL-prover based on labelled superposition with partial model guidance," in *IJCAR*, 2012, pp. 537–543. DOI: 10.1007/978-3-642-31365-3_42. Available at: https://github.com/quickbeam123/ls4 [p. 176]

[177] H. W. Thimbleby and P. B. Ladkin, "From logic to manuals again," *IEE Proceedings - Software Engineering*, vol. 144, no. 3, 1997. DOI: 10.1049/ip-sen:19971104 [p. 12]

[178] W. Thomas, "On the synthesis of strategies in infinite games," in *STACS*, 1995, pp. 1–13. DOI: 10.1007/3-540-59042-0_57 [pp. 21, 22, 165]

[179] W. Thomas, "Solution of Church's problem: A tutorial," *New Perspectives on Games and Interaction*, vol. 4, pp. 211–236, 2008. DOI: 10.2307/j.ctt46mwfz.14 [pp. 3, 21, 22]

[180] W. Thomas and H. Lescow, "Logical specifications of infinite computations," in *A Decade of Concurrency Reflections and Perspectives*, 1993, pp. 583–621. DOI: 10.1007/3-540-58043-3_29 [p. 166]

[181] S. Tripakis, "Undecidable problems of decentralized observation and control," in *CDC*, vol. 5, 2001, pp. 4104–4109. DOI: 10.1109/CDC.2001.980822 [p. 11]

[182] S. Tripakis, B. Lickly, T. A. Henzinger, and E. A. Lee, "A theory of synchronous relational interfaces," *TOPLAS*, vol. 33, no. 4, pp. 14:1–14:41, 2011. DOI: 10.1145/1985342.1985345 [p. 9]

[183] Y.-K. Tsay, "Compositional verification in linear-time temporal logic," in *FOSSACS*, 2000, pp. 344–358. DOI: 10.1007/3-540-46432-8_23 [p. 143]

[184] A. M. Turing, "Checking a large routine," in *Report of a Conference on High Speed Automatic Calculating Machines*, 24 Jun 1949, pp. 67–69. Available at: http://www.turingarchive.org/browse.php/B/8 [p. 6]

[185] H. Vanzetto, "Proof automation and type synthesis for set theory in the context of TLA$^+$," Ph.D. dissertation, Computer Science, Université de Lorraine, 2014. Available at: https://hal.inria.fr/tel-01096518 [p. 23]

[186] M. Y. Vardi, "Verification of open systems," in *FSTTCS*, 1997, pp. 250–266. DOI: 10.1007/BFb0058035 [p. 33]

[187] P. T. M. Varghese, "Parity and generalized Büchi automata: Determinisation and complementation," Ph.D. dissertation, University of Liverpool, 2014. Available at: http://livrepository.liverpool.ac.uk/id/eprint/2027479 [p. 165]

[188] T. Villa, T. Kam, R. K. Brayton, and A. Sangiovanni-Vincentelli, *Synthesis of finite state machines: Logic optimization*. Springer, 1997. DOI: 10.1007/978-1-4615-6155-2 [p. 66]

[189] A. Walker and L. Ryzhyk, "Predicate abstraction for reactive synthesis," in *FMCAD*, 2014, pp. 219–226. DOI: 10.1109/FMCAD.2014.6987617 [pp. 3, 13]

[190] J. C. Willems, "The behavioral approach to open and interconnected systems: Modeling by tearing, zooming, and linking," *IEEE Control Systems*, vol. 27, no. 6, pp. 46–99, Dec 2007. DOI: 10.1109/MCS.2007.906923 [p. 2]

[191] N. Wirth, "Program development by stepwise refinement," *CACM*, vol. 14, no. 4, pp. 221–227, 1971. DOI: 10.1145/362575.362577 [pp. 1, 3]

[192] A. Wolpers and W. Stephan, "Modular verification of programmable logic controllers with TLA," in *9th IFAC Symposium on Information Control in Manufacturing 1998 (INCOM '98)*, vol. 31, no. 15, 1998, pp. 159–164. DOI: 10.1016/S1474-6670(17)40546-5 [p. 7]

[193] T. Wongpiromsarn, "Formal methods for design and verification of embedded control systems: Application to an autonomous vehicle," Ph.D. dissertation, California Institute of Technology, 2010. Available at: http://resolver.caltech.edu/CaltechTHESIS:05272010-153304667 [p. 20]

[194] T. Wongpiromsarn, U. Topcu, and R. M. Murray, "Formal synthesis of embedded control software: Application to vehicle management systems," in *Infotech@Aerospace*, 2011. DOI: 10.2514/6.2011-1506 [p. 20]

[195] B. Yang, R. E. Bryant, D. R. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. K. Ranjan, and F. Somenzi, "A performance study of BDD-based model checking," in *FMCAD*, 1998, pp. 255–289. DOI: 10.1007/3-540-49519-3_18 [p. 180]