Appendix

Ioannis Filippidis Richard M. Murray

{ifilippi,murray}@caltech.edu

Control and Dynamical Systems
California Institute of Technology*

1 Open systems and logic

1.1 Logic and set theory

We study problems where two or more players control variables, without necessarily knowing how other players will move next. The freedom to design what moves a player picks is formalized with existential quantifiers ($\exists$), and the lack of control over other players with universal quantifiers ($\forall$). Alternation of quantifiers motivates calling mathematical models of this form *games* [1, 2].

We use the (raw) temporal logic of actions [3, 4] with some minor modifications that accommodate for a smoother connection to the literature on games and reactive synthesis. The differences are mainly that we use the raw logic (whose syntax does not enforce stutter invariance) and discuss about assignments only to variables that occur in the specification ("partial" states, as opposed to TLA$^+$ states, which assign values to all variable names, including variables that are not declared in a specification).

TLA$^+$ is based on Zermelo-Fraenkel (ZF) set theory, which is regarded as a foundation for mathematics [5]. so everything is a set (also called a "value"). A difference to ZF is that TLA$^+$ includes a formalization of function syntax. Functions can be defined with the syntax $[x \in S \mapsto e]$, where e some suitable expression [4, p.303, p.71]. A set f that satisfies the axiom schema

$$f = [x \in \text{DOMAIN } f \mapsto f[x]]$$

is a function. For convenience, let

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

A function f maps each element in the set DOMAIN f to the set $f[x]$. For any $x \notin$ DOMAIN f, the expression $f[x]$ is *some* set, but we don't know *which* set. The collection of functions with domain S and range $R \subseteq T$ forms a set, denoted by

$$[S \to T]$$

An *operator* g is a mapping without a domain that maps x to $g(x)$. Brackets (parentheses) signify a function (an operator). Operators are metatheoretic constructs, not sets. Every operator symbol is replaced by the expression from its definition before the semantics of an expression is defined. The notation $F(x, G(\_))$ denotes an operator F that takes as arguments a nullary operator x and a unary operator G (in Python the argument G would be a callable object, e.g., a function). Unnamed operators are built with the construct LAMBDA [6]. For example

$$\text{LAMBDA } x, y \,:\, x + y$$

We abbreviate by λ (as in λ-calculus).

The operator *Nat* is defined as the set of natural numbers in the standard TLA$^+$ module *Naturals* [4, §18.6, p.348]. The set of integers from 0 to n is denoted by the schema

$$0..n \;\triangleq\; \{i \in Nat \,:\, 0 \leq i \wedge i \leq n\}$$

A function f with DOMAIN $f = 1..n$ for some $n \in Nat$ is called a *tuple* and denoted with angle brackets. For example, the tuple $\langle a, b \rangle$ is a function with domain 1..2 that maps 1 to a and 2 to b, i.e.,

$$\langle a, b \rangle = [x \in 1..2 \mapsto \text{IF } x = 1 \text{ THEN } a \text{ ELSE } b]$$

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 essentially a suitable antecedent. The operator $\wedge$ denotes conjunction, $\vee$ disjunction, $\neg$ negation. Substitution of expression e_1 for (a rigid or flexible) variable x in the expression e is expressed with

$$\text{LET } x \;\triangleq\; e_1 \text{ IN } \quad e$$

1.2 Modal semantics

Assume a suitable metatheory, for example ZF with TLA$^+$ syntax (excluding the modal operators). A *state* s is an assignment of values to *all* variables. Let *VarNames* be the set of all variable names. The cardinality of variable names in TLA$^+$ is infinite and countable. A state can be defined as any function s that satisfies (within the metatheory)

$$\mathit{IsAFunction}(s) \;\wedge\; (\text{DOMAIN } s = \mathit{VarNames})$$

If x is the name of a variable, then the function s maps "x" to the set $s[\text{"}x\text{"}]$ (the value $s[\text{"}x\text{"}]$ is denoted also by $s[\![x]\!]$).

In practice algorithms reason about only finitely many variables that take values from a finite set. Real specifications mention finitely many variables and values, so there is no loss of usefulness. This makes restricted assignments ("partial" states) relevant. Define *SpecVars* as the set of variable names that occur in a given specification. Define *RelevantValues* as the set of values relevant to the specification (usually *RelevantValues* = BOOLEAN , for reasons explained in Section 1.7). In this case the set of "partial" states is

$$[var \in SpecVars \to RelevantValues]$$

A *step* is a 2-tuple of states $\langle s_1, s_2 \rangle$. A *behavior* σ is an infinite sequence of states (a function from *Nat* to states). An *action* (*state predicate*) is a Boolean-valued formula over steps (states). Given a step $\langle s_1, s_2 \rangle$, the formulae x and x' denote the values $s_1[\![x]\!]$ and $s_2[\![x]\!]$, respectively. A state predicate can be interpreted also as an action.

We will use few temporal operators: $\Box$ "always" and $\Diamond$ "eventually". If formula f is TRUE in every (some) state of behavior σ, then we write

$$\sigma \models \Box f$$

($\sigma \models \Diamond f$) is TRUE. Formal semantics are defined in [4, §16.2.4].

A property is the 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 step refinement [7]. To ensure stutter-invariance, TLA$^+$ includes the shorthands

$$[A]_v \triangleq A \vee (v' = v)$$

and

$$\langle A \rangle_v \triangleq \neg[A]_v \equiv A \wedge (v' \neq v)$$

So, $\Box[x = x + 1]_x$ is satisfied by a behavior with steps that either increment x by one, or leave x unchanged. Raw TLA$^+$ does not enforce stutter invariance, so one can write expressions of the form $\Box(x' = x + 1)$ where the operator $\Box$ is applied to an action.

If $\neg(\sigma \models \varphi)$ implies that for some $n \in Nat$, the (finite) subsequence

$$front(\sigma, n) \triangleq [i \in 0..n \mapsto \sigma[i]]$$

cannot be extended to satisfy φ, then φ is a *safety* property. If for all $n \in Nat$ the finite sequence $front(\sigma, n)$ can be extended to a behavior that satisfies φ, then φ is a *liveness* property [8]. A property of the form $\Box\Diamond p$ ($\Diamond\Box p$) is called *recurrence* (*persistence*) [9].

1.3 Collection vs set

Not every statement in axiomatic set theory defines a set. Some statements describe collections that are too large to be sets. In naive set theory this phenomenon gives rise to Rusell's and other paradoxes. A collection that is not a set is called a (proper)

class. 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. This proves that the collection of states that satisfy a formula is not a set (within the theory that the semantics is discussed).

For the above reason, we should use the term "collection" instead of "set", to accommodate for TLA$^+$ semantics. However, to use common terminology and for brevity, we will refer to "sets" of states, even when "collection" would be appropriate.

1.4 State vs predicate

We can articulate the concepts either in terms of states or state predicates (Boolean-valued expressions). We choose state predicates because global state involves all variables, so we would need to restrict global to local states (assignments to variables visible to each component) in order to discuss in terms of states. A local state represents the viewpoint of a component. It is simpler to capture the same viewpoint using a state predicate that mentions only variables visible to the component under consideration.

1.5 Realizability and synthesis

A temporal property can be less specific than the final implementation. Synthesis is the algorithmic construction of a controller that ensures the component satifies a given temporal property. Let x be uncontrolled and y controlled variables, and

$$\begin{array}{lcl} \mathit{Realization}(f, g, \mathit{mem}_0) & \triangleq & \land\ \mathit{mem} = \mathit{mem}_0 \\ & & \land\ \Box \land\ y' = f[\langle \mathit{mem}, x, y \rangle] \\ & & \qquad \land\ \mathit{mem}' = g[\langle \mathit{mem}, x, y \rangle] \end{array}$$

where *IsFiniteSet* requires finite cardinality [4, p.341]. Assume that φ is a temporal logic formula over the variables x, y. An implementation (program, circuit, etc.) of φ is a pair $\langle f, g \rangle$ of a controller function f and a memory update function g, and an initial memory value mem_0, such that

$$\models \big(\exists\, \mathit{mem} \,:\, \mathit{Realization}(f, g, \mathit{mem}_0)\big) \Rightarrow \varphi$$

be valid (true for all behaviors σ), formalized as follows.

Definition 1 ASSUME*: 1. No variable symbols other than x, y occur in formula φ*
2. The symbols f, g, mem_0 do not occur in formula φ
Let

$$\begin{array}{lcl} \mathit{IsRealizable}(\varphi) & \triangleq & \exists f, g, m_0 \,:\, \forall\, x, y \,:\, \land\ \mathit{IsAFunction}(f) \land \mathit{IsFiniteSet}(\text{DOMAIN } f) \\ & & \qquad \land\ \mathit{IsAFunction}(g) \land \mathit{IsFiniteSet}(\text{DOMAIN } g) \\ & & \qquad \land\ \big(\exists\, \mathit{mem} \,:\, \mathit{Realization}(f, g, \mathit{mem}_0) \Rightarrow \varphi\big) \end{array}$$

So *IsRealizable*(φ) denotes existence of an imlementation.

We note in passing that the index j identifies the component (so x, y). The initial condition *Init* forms part of φ, for example $\varphi \triangleq \mathit{Init} \Rightarrow \Box \mathit{Next}$. A stutter-invariant definition in TLA$^+$ is possible [10, 11] but the above suffices here. Establishing a smoother connection with previous results on games is another reason for using the above definition.

The function f actuates on variables of player j that are mentioned in φ. In contrast, function g modifies the *memory* variable *mem* that is used by the strategy to remember information. In general adding memory can be necessary, for example if φ contains multiple recurrence goals $\Box\Diamond R_k$.

1.6 Tractable liveness

A formula described by the schema

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

defines a liveness property categorized as generalized Streett(1), or GR(1) [12]. Conjoining k Streett pairs yields a liveness property called GR(k) [13]. Each additional Streett pair increases the nesting of fixpoint iterations, leading to factorial complexity of synthesis in the number of Streett pairs [14]. This is why GR(1) properties are preferred to write synthesis specifications.

An implementation that satisfies a GR(1) property can be computed in time cubic in the number of relevant states, and linear in the product of recurrence and persistence goals. For example, a GR(1) specification with N recurrence and M persistence goals that is expressed using 10 bits can be synthesized in at most $NM(2^{10})^3 = NM2^{30}$ controllable predecessor operations [15, 16, 17]. In practice, runtime is much smaller, due to the symbolic implementation [18].

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 [19], so exponentially more expensive than GR(1) synthesis [12]. For this reason GR(1) synthesis algorithms unconditionally add a memory variable that ranges over $1..n$. To simplify the algorithms and presentation that follows, we will embed the memory $1..n$ a priori in the state space, which is common practice in the literature on games, for example reductions to parity games. By the previous remarks, we do not lose generality, because we would anyway not be searching for memoryless strategies.

1.7 Relevant values in untyped logic

Game solving involves reasoning about (sets of) states. Symbolic methods are used to avoid enumerating states, by representing sets of states using binary decision diagrams (BDDs). Reasoning about variables that take integer values is reduced to reasoning about variables that take Boolean values, so that we can use BDDs. In an untyped logic, as TLA$^+$ is, variables can take any value. For example, any integer can be the value of variable k.

Clearly, we cannot directly reason about an infinity of values using BDDs. What allows using BDDs (or other finitistic methods) to reason about properties expressed in an untyped logic is that for the properties we are interested in only finitely many (integer) values are relevant. For all other values, it is straightforward to prove (syntactically, not using BDDs) that they are irrelevant (for example, because they violate a type invariant in the assumption or guarantee formulas).

How do we know which values are relevant? We write the specification formulae according to a syntactic schema (pattern) that makes it easy to recognize the type invariants. (This is no less general than using a typed logic. In a typed logic, instead of this schematic requirement, we have the equally restrictive requirement of declaring the same information as variable types.) Taking into consideration the proof about what values are relevant, and the information extracted from the type invariants that appear in the specification formulae, we use this information as *type hints* [20]. Using these type hints, the algorithm can choose a sufficient number of new "bit" variables to represent each original variable, so that reasoning about Boolean assignments to the new variables suffice to reach conclusions about the original variables. The BDDs are used to represent the specification after it is expressed in terms of the "bit" variables, based on the refinement mapping [21] [4, §10.8]. that associates the original and "bit" variables.

The "bit" variables are untyped, as the original variables. The point is that reasoning for the original formulae would require manipulating directly integer values, which would require using a data structure different than BDDs (for example, algebraic decision diagrams), or enumeration. Rewriting the problem with formulas for which reasoning about Boolean assignments suffices, we can use BDDs. (To emphasize the point, we could reuse the original variable names to name some[1] of the "bit" variables, though doing so would be confusing.)

Provided the type hints match the type invariants that appear within assumptions and guarantees of the specification's temporal properties, the BDD representation is adequate for reasoning about the origina formulas.

1.8 Interleaving

Let i be a variable used to express the assumption that players move in turns. The variable i tracks the player whose turn it is to move, thus expressing the modeling assumption of interleaving. For readability below i is omitted. Otherwise, i needs to be constrained to the player's turn and i' to the turn of the next player, for example

$$\begin{array}{l} \mathit{SysCPre}(x, y, i) \triangleq \\ \quad \land\ i = \mathit{SysTurn} \\ \quad \land\ \exists\, y' : \forall\, h : \lor\ \neg \mathit{Inv}(x, y, h, i) \\ \qquad\qquad\qquad\quad \lor\ \land\ \mathit{SysNext}(x, y, h, y') \\ \qquad\qquad\qquad\qquad\ \land\ \mathit{Target}(x, y', h, \mathit{EnvTurn}) \end{array}$$

where usually $\mathit{EnvTurn} \triangleq \mathit{SysTurn} \oplus_2 1$ (where $i \oplus_n j \triangleq ((i + j)\%n)$. Finer detail can be found in the implementation code. Observe that i expresses a mod-

[1] We would need to add names because there are "bit" variables than original variables.

eling assumption about interleaving, thus about a lower level protocol that ensures interleaving execution. The use of i above lets it be a neutral variable. Attributing changes of i to an environment player (other than the components involved in decomposition) is a specification style [22] noninterleaving wrt the environment, interleaving among the components, and with disjoint state, because each variable is controlled by a fixed entity throughout each entire behavior. Attributing changes of i to the system (by letting i be controlled by the system by defining via actions μ [23, 4, 10] how state control is shared among different players) leads to a specification style that does not necessitate an environment external to the components, is shared-state, and interleaving. The definitions of interleaving and disjoint state used here are those defined in [10] based on [23, 4]. The above remarks indicate that expressing in TLA$^+$ an assumption that models interleaving of some components results in either

1. disjoint state and noninterleaving with an external environment, or
2. shared state and interleaving.

The first style is expressed by the *SysCPre* given above. The *SysCPre* below expresses the second style

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

where

$$\begin{array}{l} \models \mathit{SysNext}(x, y, h, y') \Rightarrow \land\ i = \mathit{SysTurn} \\ \qquad\qquad\qquad\qquad\qquad\quad \land\ i' = \mathit{EnvTurn} \end{array}$$

Both descriptions yield the same realizability results. So, they are modeling styles and not mathematically essential.

2 Contracts

A single temporal property φ specifies the behavior of a player. To specify the behavior of two (or more) players, we can use a collection of properties φ_0, φ_1. Just listing the properties φ_0, φ_1 does not say much about the relation between them. To be useful, the properties should be implementable from the same initial conditions, motivating the following definitions.

Definition 2 (Contract [24])
ASSUME*:* *1.* ZF NEW $n \in \mathbb{N}$ *(number of players)*

2. IsRealizable(j, ψ_j) *as defined in Section 1.5, with* j *used to select the variables of player* j*.*

A contract *is*

1. a partition of variables among players in $1..n$

2. *a collection of properties* $\psi_1, \ldots, \psi_n$ *such that (*$\forall$ *here is metasyntax that serves as conjunction)*

$$\models \forall\, j \in 1..n \,:\, \mathit{IsRealizable}(j, \mu_j)$$

Definition 3 (GR(1) contract) *A GR(1) contract contains only assume-guarantee GR(1) properties.*

A property ψ refines a property φ if [23, 4]

$$\models \psi \Rightarrow \varphi$$

Remark 4 (No set of open-system behaviors) *If Init bounds all initial variable values to be contained in sets, then the collection of behaviors (within the metatheory ZF) that satisfy the assume-guarantee property* $\mathit{AsmGrnt}(\mathit{Init}, \ldots)$ *is not a set (it is too large to be a set [4, p.66, p.311]) [10]. This collection is a proper class (informally in ZF, formally in metatheories that include classes). So we cannot write* $[\![\psi]\!]$ *for any set in the metatheory, unless we restrict the semantics to define behaviors only for a specific set of values.*

Definition 5 (Contract refinement) *A contract* $\psi_1, \ldots, \psi_n$*, refines a property* φ *if, and only if,*

$$\models (\forall\, j \in 1..n \,:\, \psi_j) \;\Rightarrow\; \varphi$$

In this definition, initial conditions are part of ψ_j and φ. Non-vacuity of ψ_j is ensured by the definition of *IsRealizable*. Compatibility of assumptions in ψ_j (non-triviality) is ensured by the violation it would cause of φ, by allowing arbitrary behavior for the variables of some player.

3 Overall algorithm

The algorithm for decomposition that accounts for more than two players is shown in Algs. 1 and 2. The main difference of calls to MakePinfoAssumption from OneRoundOfGoals, compared to DecomposeTeam is passing the keyword argument *pred*, which propagates to be used in $\mathit{ProjHiddenVrs}_{team}(a, \mathit{pred}) \triangleq \text{LET } \mathit{Inv} \triangleq \mathit{pred} \text{ IN } \neg \mathit{Obs}_{team}(\neg a)$.

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

Figure 1: Overall algorithm.

```
def DecomposeGR1Property(
      goals, player, team, spc, pred := NONE) :
   DeclareMasksFor(spc.players \ team)
   inv := Closure(aut)
   spc.inv := inv
   PreserveInvariant(inv, spc)
   aut.observer := player
   spc.visible := {player}
   z := TRUE
   z := NONE
   while z ≠ z_old
      z_old := z
      u := TRUE
      for goal ∈ goals :
      u := u ∧ ProcedureOneRoundOfGoals(z, goal, team, spc)
      z := z ∧ u
```

Figure 2: Overall algorithm (continued).

```
def OneRoundOfGoals(z, goal, team, spc) :
  obs_target := Obs(target, spc.observer, team, spc)
  assumptions := {}
  others := spc.players \ {player}
  covered := obs_target
  covered_old := NONE
  while covered ≠ covered_old :
    covered_old := covered
    (* Can others help as a team? *)
    attr, η_player, η_team, new_spc := MakePinfoAssumption(
      covered, player, team, spc)
    (* Can others help w/o circularity? *)
    if Len(others) > 1 :
      sft := spc.inv ∧ η_player
      goal := ¬sft
      dep_player, traps := DecomposeTeam(
        goal, sft, new_spc, others)
    else : (* Terminal case *)
      traps := {}
    assumptions := assumptions ∪ traps ∪ {⟨player, dep_player, η_player⟩}
    covered := covered ∨ attr ∨ η_player
  return covered

def DecomposeTeam(goal, pred, spc, players) :
  player := PickFrom(players)
  others := players \ {player}
  DeclareMasksFor(spc.players \ team)
  team.observer := Pick(team)
  spc.visible := team
  obs_target := Obs(goal, player, spc)
  assumptions := {}
  covered := obs_target
  (* Least fixpoint *)
  while covered ≠ covered_old :
    covered_old := covered
    attr, η_player, η_team, new_spc := MakePinfoAssumption(
      covered, player, team, spc, pred := pred)
    if Len(others) > 1 :
      sft := spc.inv ∧ η_player
      goal := ¬sft
      dep_player, traps := DecomposeTeam(
        goal, sft, new_spc, others)
    else :
      traps := {}
    assumptions := assumptions ∪ traps ∪ {⟨player, dep_player, η_player⟩}
    covered := covered ∨ attr ∨ η_player
  return player, assumptions
```

References

[1] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer, "Alternation," *JACM*, vol. 28, no. 1, pp. 114–133, 1981.

[2] I. Walukiewicz, "A landscape with games in the background," *LICS*, pp. 356–366, 2004.

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

[4] ——, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

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

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

[7] ——, "What good is temporal logic?" in *Information Processing*, vol. 83, 1983, pp. 657–668.

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

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

[10] I. Filippidis and R. M. Murray, "Formalizing synthesis in TLA$^+$," Caltech, Tech. Rep. CaltechCDSTR:2016.004, 2016.

[11] L. Lamport, "Miscellany," 21 April 1991, note sent to TLA mailing list.

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

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

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

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

[16] W. Thomas and H. Lescow, "Logical specifications of infinite computations," in *A decade of concurrency reflections and perspectives*, 1993, pp. 583–621.

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

[18] F. Somenzi, "CUDD: CU Decision Diagram package - v2.5.0," *Colorado University at Boulder*, 2012.

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

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

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

[22] S. Merz, "A user's guide to TLA," in *Modélisation et vérification des processus parallèles: Actes de l'école d'été*. Nantes, France: Ecole centrale de Nantes, 1998, pp. 29–44.

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

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