Comparative Case Studies of Reactive Synthesis and Supervisory Control

Downloaded from: https://research.chalmers.se, 2025-06-18 04:33 UTC

Citation for the original published paper (version of record):
Ramezani, Z., Krook, J., Fei, Z. et al (2019). Comparative Case Studies of Reactive Synthesis and Supervisory Control. 2019 18th European Control Conference, ECC 2019: 1752-1759. http://dx.doi.org/10.23919/ECC.2019.8795696

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

2019 18th European Control Conference (ECC)
Napoli, Italy, June 25-28, 2019

Comparative Case Studies of Reactive Synthesis and Supervisory Control

Zahra Ramezani Jonas Krook Zhennan Fei Martin Fabian Knut Åkesson

***Abstract*— *Reactive Synthesis* and *Supervisory Control Theory* are both systematic approaches for the automatic construction of controllers from requirements. However, their underlying technicalities differ significantly. This paper provides an empirical comparison between these two approaches from the modelling perspective through case studies. Using the synthesis tools TuLiP and Supremica, two examples are modelled in the typical modelling formalism supported by each tool, and the algorithms are applied to synthesize controllers. Based on the obtained models and experiences, we compare how the models are derived, and how the characteristics of the examples and the underlying synthesis algorithms influence the modelling choices.**

I. Introduction

Reactive Synthesis (RS) [1], developed in the field of computer science, is an automated methodology for synthesizing correct-by-construction controllers, referred to as *reactive modules*, from requirements describing the intended behavior. A reactive module evolves in computation cycles and engages in an ongoing interaction with its *environment*. The reactive module alternately reads inputs from the environment and assigns values to its outputs, possibly affecting the environment. Recent research on RS includes environments with assumptions [2] and synthesis with plants [3], [4]. For a comprehensive introduction to RS, refer to [5].

Supervisory Control Theory (SCT) [6], [7] is a model-based approach for control of Discrete Event System (DES). A DES is a dynamic system that can be characterized by a set of states whose transitions are triggered by occurrences of *events*. Given a DES to be controlled, the *plant*, and a *specification* describing the desired behavior, a control entity, called *supervisor*, can be automatically synthesized to dynamically restrict the behavior of the plant, such that the closed-loop system satisfies the specification.

RS and SCT both employ a systematic approach for automatically synthesizing controllers that are *resilient* in the sense that the closed-loop behavior fulfills given requirements in an open environment; the behavior is affected by an environment that cannot be restricted by the controllers. However, their underlying technicalities differ significantly [8], [9]. In [8], the focus is on the basic synthesis problem and a comprehensive comparison on how properties such as *safety*, *non-blockingness* and *maximal permissiveness* are treated algorithmically within RS and SCT. Building upon these connections, a formal procedure is introduced to reduce the basic SCT problem to a specific variant of RS problem with plants and maximal permissiveness requirements. In [9], the authors take this comparison one step further and identify the conditions under which the algorithm of one field can be tailored to address the synthesis problem considered in the other. To facilitate the analysis, ω-language is used as the common base, and hence a branch of SCT that targets the synthesis problem for ω-languages [10] is considered throughout the comparison by [9].

Besides the synthesis problems and underlying algorithms, the difference between RS and SCT also lies in the modelling tools and the corresponding modelling techniques. In RS, *Linear Temporal Logic* (LTL) [11] is one widely used formalism to express the specifications. An LTL fragment that is relevant to this work is General Reactivity of Rank 1 (GR(1)) [12]. In practice, GR(1) synthesis is particularly useful thanks to the efficient polynomial-time synthesis algorithm [12]. In SCT, on the other hand, plants and specifications are typically modelled by Finite Automata (FA) [7], and/or an extension thereof called Extended Finite Automata (EFA) [13] in which FA are augmented with variables.

Inspired by the aforementioned line of research [8], [9], this paper complements those works by comparing RS and SCT from a modelling perspective. We use case studies to reveal some fundamental differences between the synthesis approaches by modelling the case studies in two synthesis tools; TuLiP [14] that implements RS synthesis and Supremica [15] that implements SCT synthesis.

The contributions of this paper are: (i) We present complete modelling solutions for two examples, each of them modelled in TuLiP and Supremica using the provided modelling formalism of each tool. (ii) We compare these two synthesis approaches empirically based on the obtained models and experiences. Specifically, we compare how the models are derived, and how the characteristics of the examples and the underlying synthesis algorithms influence the modelling choices. The comparison confirms what has been concluded from [8], [9] but emphasizes the differences in terms of modelling. The presented results are intended to provide practitioners with guidelines on selecting the suitable method and tool that fit the scope of the considered synthesis problem.

The rest of the paper is organized as follows. After recalling the necessary preliminaries in Section II, Section III and Section IV detail the modelling of two selected examples in TuLiP and Supremica, respectively. Section V compares

The authors are with the Department of Electrical Engineering, Chalmers University of Technology, Gothenburg, Sweden. Email: {rzahra, krookj, zhennan, fabian, knut}@chalmers.se. This work was supported by Vetenskapsrådet SyTeC, and the Wallenberg AI, Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation.

Authorized licensed use limited to: Chalmers University of Technology Sweden. Downloaded on June 05,2020 at 09:29:27 UTC from IEEE Xplore. Restrictions apply.

the models obtained from the previous sections and reveals differences and similarities between RS and SCT. Finally, Section VI concludes the paper by summarizing the contributions. The complete set of models can be found online[1].

II. Preliminaries

A. Reactive Synthesis

Reactive Synthesis aims to automatically synthesize a controller, referred to as a *reactive module*, that satisfies the desired *guarantees* ϕ_s, under the *assumptions* of the environment ϕ_e. In other words, the synthesized controller satisfies the formula $\phi_e \rightarrow \phi_s$ [2]. One way to model the RS problem is to consider the controller to be synthesized and the environment as adversaries that play a finite-state game and take turns to provide input to each other [16]. Then, an iterative process can be adopted to find a fix-point of a subset of states and transitions that solves the RS problem. The states, transitions, inputs and outputs can be modelled by a Kripke structure.

Definition 1. A Kripke Structure is a tuple

$$M = \langle S, I, R, AP, L \rangle,$$

where S is a set of states, $I \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a transition relation, AP is a set of atomic propositions, and $L : S \rightarrow 2^{AP}$ is a labelling function that defines the atomic propositions that are true in each state. AP is divided into two disjoint subsets AP_e and AP_s, representing the propositions of the environment and the controller, respectively.

The atomic propositions in AP_e are seen as the inputs to the controller, while AP_s are its outputs. The atomic propositions in AP can be composed of relational operators on functions of discrete finite domain variables as they can be considered either true or false, given a certain valuation in a certain state.

LTL formulae can be evaluated over infinite runs on a Kripke structure. In addition to standard propositional logic operators, LTL includes temporal operators [11]. The temporal operators $'$ (next), $\Box$ (always), and $\Diamond$ (eventually) are used in this paper. A run π of a Kripke structure M is an infinite sequence of states $\{\pi_0, \pi_1, \dots\}$, where $\pi_0 \in I$ and $(\pi_i, \pi_{i+1}) \in R$. Let $\pi[i]$ represent the infinite run starting from state π_i. Let τ and θ be LTL formulae, and ψ be an atomic proposition. The definition of when a run π satisfies a formula is given inductively:

- $\pi \vDash \tau$ iff $\pi[0] \vDash \tau$
- $\pi[i] \vDash \psi$ iff $\psi \in L(\pi_i)$
- $\pi[i] \vDash \neg\tau$ iff $\pi[i] \nvDash \tau$
- $\pi[i] \vDash \tau \vee \theta$ iff $\pi[i] \vDash \tau$ or $\pi[i] \vDash \theta$
- $\pi[i] \vDash \tau'$ iff $\pi[i+1] \vDash \tau$
- $\pi[i] \vDash \Box\tau$ iff $\pi[k] \vDash \tau$ for all $k \geq i$
- $\pi[i] \vDash \Diamond\tau$ iff $\pi[k] \vDash \tau$ for some $k \geq i$

The Kripke Structure M satisfies a formula τ if every possible run π satisfies the formula.

[1] https://github.com/krooken/ComparativeCaseStudies

Given an LTL specification φ for the environment and the system, TuLiP synthesizes a controller such that the system satisfies φ, if such a controller exists. The LTL specification φ has to be in GR(1) form [12]:

$$\varphi \triangleq \big((\psi^e_{init} \wedge \Box\psi^e_{safe} \wedge \bigwedge_{0<i\leq J} \Box\Diamond\psi^e_{live,i}) \rightarrow (\psi^s_{init} \wedge \Box\psi^s_{safe} \wedge \bigwedge_{0<i\leq N} \Box\Diamond\psi^s_{live,i})\big), \quad (1)$$

where ψ^e_{init} and ψ^s_{init} contain all the initial conditions of the environment and the controller, respectively. ψ^e_{safe} and $\psi^e_{live,i}$ (J sub-specifications) are the *safety* and *liveness* assumptions on the environment. ψ^s_{safe} and $\psi^s_{live,i}$ (N sub-specifications) are the safety and liveness properties of the system.

Note that TuLiP interprets the implication in (1) as a *strict realizability* implication [17].

B. Supervisory Control Theory

Given a plant G and a specification K of the desired controlled behavior, the SCT [6] makes it possible to automatically synthesize a supervisor S that controls the plant such that the specification is satisfied.

The supervisor and the plant form an asymmetric *closed-loop* system where the supervisor restricts the event generation of the plant. As the plant generates events enabled by the supervisor, the supervisor observes the generated sequences of events and determines, at each state, which of the currently possible events that should be enabled. Some of the events cannot be disabled by the supervisor. These events are *uncontrollable*. The supervisor is required to be *controllable*, i.e. it must never try to disable an uncontrollable event. By disabling controllable events, the supervisor can confine the plant to a subset of its possible states so that the closed-loop system only visits states that are considered "good" by the specification. At the same time, the supervisor guarantees that from any closed-loop state, the system can always continue to a desired marked state, it is *non-blocking*. Finally, the supervisor disables as few events as possible, it is *maximally permissive*.

Basically, supervisor synthesis is an iterative removal of states and/or transitions of an initially calculated supervisor "candidate". Practically, this candidate is calculated from $G||K$ where $||$ denotes the full synchronous composition operator [7]. The iterative algorithm removes from $G||K$ the states that break the controllability and/or the non-blocking properties. Iteration is necessary since enforcing one property may break the other. The iteration will eventually reach a fix-point, and what then is obtained is the maximally permissive supervisor. In Supremica, one way to model this supervisor candidate is to use EFA.

EFA are automata extended with bounded discrete variables, and updates defining logical conditions over, and/or assignments of, those variables. Let $V = \langle v_1, v_2, \dots, v_n \rangle$ be an ordered set of variables, with each variable v_i associated to a finite discrete domain, $dom(v_i) = \{\hat{v_i}^1, \hat{v_i}^2, \dots, \hat{v_i}^j\}$, having an initial value $\hat{v_i}^\circ \in dom(v_i)$, and a set of marked

Authorized licensed use limited to: Chalmers University of Technology Sweden. Downloaded on June 05,2020 at 09:29:27 UTC from IEEE Xplore. Restrictions apply.

values $\hat{V}_i^m = \{\hat{v_i}^{m_1}, \hat{v_i}^{m_2}, \ldots, \hat{v_i}^{m_k}\} \subseteq dom(v_i)$. Define the domain of V as $dom(V) = dom(v_1) \times dom(v_2) \times \ldots dom(v_m)$.

Definition 2. An extended finite automaton (EFA) is a 7-tuple:

$$E = \langle L, V, \Sigma, \rightarrow, L^i, L^m, \hat{V}^m \rangle$$

where L is a finite set of locations; V is as above. Σ denotes a finite set of events; $L^i \subseteq L$ and $L^m \subseteq L$ denote the set of initial locations and marked locations, respectively; $\hat{V}^m = \{\hat{V}_i^m : i = 1, \ldots, n\}$ is the set of marked values; $\rightarrow \subseteq L \times \Sigma \times \Pi \times L$ is the extended transition relation where Π denotes the set of updates, which are formulas consisting of variables, integer constants, Boolean literals, as well as propositional logic and discrete arithmetic connectives. A *state* of an EFA is a tuple $(\ell, \hat{V})$ where $\ell \in L$ and $\hat{V} \in dom(V)$. Hence, the initial states are defined as $Q^i = L^i \times \hat{V}^\circ$ and the marked states as $Q^m = L^m \times \hat{V}^m$.

A transition between locations ℓ, ℓ' with event $\sigma \in \Sigma$ and update $p \in \Pi$ is written as $\ell \overset{\sigma:p}{\rightarrow} \ell'$. The transition can be fired if E is at location ℓ and the update p evaluates to true; consequently, E changes its location to ℓ' while updating the variables in p; variables not in p remain unchanged.

For SCT, the alphabet Σ of an EFA is partitioned into the disjoint sets of the controllable, Σ_c, and uncontrollable, Σ_u, event sets; $\Sigma = \Sigma_c \dot{\cup} \Sigma_u$.

III. Stick-Picking Game

The stick-picking game is a simple game in which two players take turns drawing one, two or three sticks from a pile, which initially contains seven sticks. Players are not allowed to skip their turns, and the player who draws the last stick loses the game.

There is a winning strategy for player one who starts the game by picking 2 sticks. Regardless of the number of sticks picked by player two, player one can pick a number of sticks next such that only 1 stick remains, offering player two no choice but losing the game.

A. TuLiP

The formulae required for the synthesis are structured into the format of (1) in the following way:

$$\psi^e_{init} \triangleq \phi^e_{init,1} \qquad \psi^s_{init} \triangleq (\phi^s_{init,1} \wedge \phi^s_{init,2})$$
$$\psi^e_{safe} \triangleq \bigwedge_{i=1}^{3} \phi^e_{safe,i} \qquad \psi^s_{safe} \triangleq \bigwedge_{i=1}^{7} \phi^s_{safe,i}$$
$$\psi^e_{live} \triangleq \top \qquad \psi^s_{live} \triangleq \top,$$

where the subformulae are given below.

When modelling the stick-picking game in TuLiP, four different things need to be kept track of (each with a separate variable): the number of remaining sticks, whose turn it is, and how many sticks each of the two players pick, respectively, during their turn. A strategy for player one is sought, so player two's behavior is part of the environment and player one is part of the system. The dynamics for the number of sticks and the player turn do not involve taking decisions, so these dynamics can either be assumptions or assertions.

The integer variable *sticks* represents the number of sticks, and the binary variable *player_turn* models the alternation of turn between the players. Both of the variables need to be initialized to their initial values.

$$\phi^s_{init,1} \triangleq (sticks = 7 \wedge player\_turn = 1) \tag{2}$$

Player one is the player who starts the game, so player one may pick 1, 2 or 3 sticks initially, and player two picks 0 sticks waiting for player one's turn to end. The variables *player_1_picks* and *player_2_picks* denote the number of sticks the players pick.

$$\phi^s_{init,2} \triangleq player\_1\_picks \in \{1, 2, 3\} \tag{3}$$
$$\phi^e_{init,1} \triangleq player\_2\_picks = 0 \tag{4}$$

As the game progresses, player one and player two pick sticks according to whose turn it is. Additionally, they cannot pick more sticks than what remains. Below, the formulae for player one are given; the formulae $\phi^e_{safe,1}$, $\phi^e_{safe,2}$, and $\phi^e_{safe,3}$ for player two are defined analogously.

$$\phi^s_{safe,1} \triangleq \big((sticks > 0 \wedge player\_turn = 1) \rightarrow (player\_1\_picks \in \{1, 2, 3\})\big) \tag{5}$$
$$\phi^s_{safe,2} \triangleq \big((sticks = 0 \vee player\_turn \neq 1) \rightarrow (player\_1\_picks = 0)\big) \tag{6}$$
$$\phi^s_{safe,3} \triangleq (player\_1\_picks \leq sticks) \tag{7}$$

$\phi^s_{safe,2}$ is not strictly necessary, but makes the RS solution easier to interpret and compare with the SCT solution. If not used, four possible transitions (one for each value in the domain of $player\_2\_picks$) will be considered in the synthesis when it is player one's turn, but all four transitions would be equivalent with this specification.

The number of remaining sticks decreases as the players pick them. Note that an unprimed variable name refers to that variable's valuation in state π_i, while a primed variable name refers to that variable's valuation in state π_{i+1}.

$$\phi^s_{safe,4} \triangleq (player\_turn = 1) \rightarrow (sticks' = sticks - player\_1\_picks) \tag{8}$$

$\phi^s_{safe,5}$ for player two is similar.

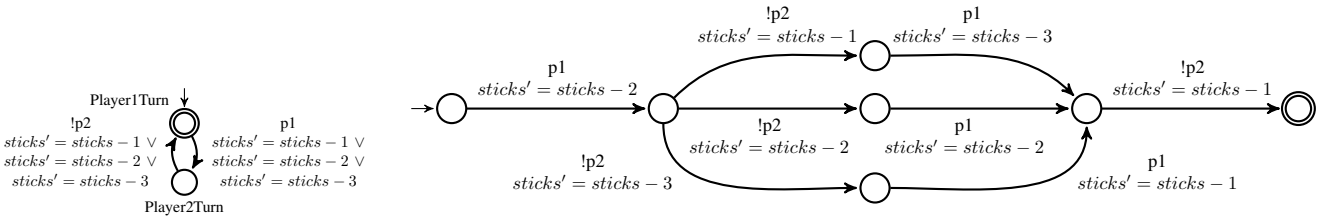
While there are still sticks left to pick, the player turn alternates between player one and player two.

$$\phi^s_{safe,6} \triangleq (sticks > 0 \rightarrow player\_turn' \neq player\_turn) \tag{9}$$

These are the rules of the game.

There are several ways to express the winning condition for player one. Here are three examples that all produce

Authorized licensed use limited to: Chalmers University of Technology Sweden. Downloaded on June 05,2020 at 09:29:27 UTC from IEEE Xplore. Restrictions apply.



(a) EFA for alternating the turn between the players.

(b) EFA enforcing the winning strategy for player one.

Fig. 1: EFAs for the plant and supervisor in Supremica.

equivalent synthesized strategies:

$$\phi^s_{safe,7a} \triangleq \big((sticks > 0 \wedge player\_turn = 1) \rightarrow (player\_1\_picks < sticks)\big) \tag{10}$$

$$\phi^s_{safe,7b} \triangleq \big((sticks > 0 \wedge sticks' = 0) \rightarrow (player\_turn = 2)\big) \tag{11}$$

$$\phi^s_{safe,7c} \triangleq \big((sticks > 0 \wedge sticks' = 0) \rightarrow (player\_turn \neq 1)\big) \tag{12}$$

$\phi^s_{safe,7a}$ means that for player one to win, as long as there are sticks left he or she must not pick all remaining sticks. $\phi^s_{safe,7b}$ means that when the last stick will be picked, it is player two's turn to pick. $\phi^s_{safe,7c}$ means that when the last stick will be picked, it is not player one's turn to pick.

B. Supremica

When modelling the stick-picking game in Supremica, the player turn is encoded as two locations, and the number of sticks is encoded by a variable $sticks$ defined as follows.

$$sticks \in \{0, \ldots, 7\}, \quad sticks^{\circ} = 7, \quad sticks^{m} = \{0\}.$$

The different player moves are encoded as events $p1$ and $p2$, one for each player. Since the strategy for player one is sought, event $p1$ is controllable while event $p2$ is uncontrollable. When an event is triggered, the player has the choice to remove 1, 2 or 3 sticks, which is encoded as updates on the corresponding transition. The *sticks* variable cannot be negative, so the updates are not fulfilled if the subtraction would yield a negative number. If no update is fulfilled, the event on that transition is disabled. Hence, no event can occur when there are zero sticks left.

Figure 1(a) shows the EFA that keeps track of player turn. Uncontrollable event such as $p2$ is prefixed by ! as a convention. When in the *Player1Turn* location, only the event associated to player one is enabled, and analogously for player two. When the event indicating that player one picks sticks is triggered, the plant transitions to player two's turn (*Player2Turn*).

When the last stick is picked, the variable *sticks* becomes 0. If the last stick was picked by player two, then the EFA in Figure 1(a) is in the location *Player1Turn*. Combining these two observations means that marking *sticks=0* and *Player1Turn* as accepting locations form a necessary and sufficient requirement for player one to win. The winning synthesized strategy is shown in Figure 1(b).

The winning strategy synthesized by Supremica is the same as the strategy synthesized by TuLiP. For this example, there is exactly one winning strategy and thus both of the tools have to generate the same strategy. Note that, in general, an SCT tool generates *all* winning strategies while RS tools only generate *one* winning strategy.

IV. Autonomous Driving

The second example designed specifically in this paper for the comparison is a simplified autonomous driving (AD) model involving two vehicles: an AD vehicle, called *ego-vehicle*, and a manually driven vehicle, called *env-vehicle*.

The two vehicles are travelling on a circular road with two lanes in the same direction. The problem is to design a controller for *ego-vehicle* that at all times guarantees safe distance between the two vehicles when they are in the same lane. That is to say, *ego-vehicle* observes the behavior of *env-vehicle*, and adjusts its own actions accordingly, such that collisions are always avoided.

Unsurprisingly, if *env-vehicle* is capable of doing any arbitrary action, an non-zero control strategy cannot prevent collisions and therefore reasonable assumptions must be made on the behaviors of the vehicles. These assumptions, which are detailed below, arise from two aspects, (i) odometry constraints that a typical vehicle has, and (ii) the basic traffic rules vehicles must obey.

Some simplifications have been made. Firstly, vehicle dimensions are not considered so the vehicles are modelled as singular points. Secondly, the velocities of the vehicles are abstracted as integers with a defined range $\{0, 1, 2\}$ in increasing order of velocity. Thirdly, the two lanes of the circular road are simplified to have the same length. Finally, inertia has no affect on the vehicles, thus accelerating or decelerating to a specified speed can be done without delay.

Initially both vehicles stand still in the right lane, with *env-vehicle* ten units ahead of *ego-vehicle*, which starts at the origin. The safety distance between the two vehicles is set to be two units. All the assumptions put on *env-vehicle* are also put as guarantees on *ego-vehicle* to uphold. Thus, for brevity, only the assumptions on *env-vehicle* are given:

1) When in the same lane, the longitudinal distance between the vehicles is larger than the safety distance.

Authorized licensed use limited to: Chalmers University of Technology Sweden. Downloaded on June 05,2020 at 09:29:27 UTC from IEEE Xplore. Restrictions apply.

2) When *env-vehicle* changes to the left (right) lane, it activates the left (right) turn indicator before changing lane, and deactivates it after completing the lane change.
3) When *env-vehicle* is in the left (right) lane, the turn indicator is either not used or indicating right (left).
4) The new position of *env-vehicle* is updated by the old position plus the velocity and then modulo the length of the road.

These assumptions make sure that *env-vehicle* (and *ego-vehicle*) behave 'well' while driving. However, they do not state that *ego-vehicle* should be able to drive. It is reasonable to expect an autonomous vehicle to be able to drive, and not end up in situations where it is not possible to continue. Therefore, *ego-vehicle* is, in addition to the above, expected to drive at some point. (This is not expected of *env-vehicle* because *ego-vehicle* will need to take all possible actions of *env-vehicle* into account, including driving all the time or standing still forever.)

A. TuLiP

The synthesis problem of the AD example fits well in the RS framework given its inherent characteristics. This section demonstrates the modelling of the example by expressing the natural language specifications as a GR(1) LTL formulae according to (1). To this end, the input and output variables are declared, which correspond to the environment and controller variables, respectively, in TuLiP (with $\star \in \{env, ego\}$):

- $lane\_\star \in \{\textit{left}, \textit{right}\}$
- $pos\_\star \in \{0, \ldots, 29\}$
- $vlc\_\star \in \{0, 1, 2\}$
- $ind\_\star \in \{\textit{none}, \textit{left}, \textit{right}\}$

The variables $lane\_\star$, $pos\_\star$, $vlc\_\star$, and $ind\_\star$, respectively hold the current lane, the longitudinal position, the velocity, and the direction of the turn indicators.

With the variables defined, the assumptions of (1) can be formalized. The initial condition, $\psi^e_{\textit{init}}$, is defined as

$$\begin{aligned}\psi^e_{\textit{init}} \triangleq (&vlc\_env = 0 \ \wedge \ pos\_env = 10 \\ &\wedge \ lane\_env = \textit{right}).\end{aligned} \tag{13}$$

The formula $\psi^e_{\textit{safe}}$ in (1) is defined as the conjunction of sub-formulae ϕ_i where $i \in \{1, \ldots, 4\}$, with the indices corresponding to the natural language assumptions given above.

$$\psi^e_{\textit{safe}} \triangleq \bigwedge_{i=1}^{4} \phi^e_i. \tag{14}$$

Assumption 1, which regards safe distance, is formalized as ϕ^e_1:

$$\begin{aligned}\phi^e_1 \triangleq &\ lane\_env = lane\_ego \rightarrow \\ &|pos\_env - pos\_ego| > 2,\end{aligned} \tag{15}$$

Assumption 2 is split into two sub-formulae, one for changing lanes from right to left, and one for the other direction. In essence, the formulae express that if the vehicle is about to change lanes, then the turn indicator has to be activated and the vehicle must not be stationary.

$$\begin{aligned}\phi^e_{2,a} \triangleq &\ (lane\_env = \textit{right} \ \wedge \ lane\_env' = \textit{left}) \rightarrow \\ &(ind\_env = \textit{left} \ \wedge \ ind\_env' = \textit{none} \\ &\wedge \ vlc\_env > 0),\end{aligned} \tag{16}$$

and vice versa for left to right ($\phi^e_{2,b}$). Then $\phi^e_2 \triangleq \phi^e_{2,a} \wedge \phi^e_{2,b}$. Assumption 3 is also split in two, where $\phi^e_{3,a}$ express that the left indicator cannot be active when driving in the left lane, and vice versa for $\phi^e_{3,b}$.

$$\begin{aligned}\phi^e_{3,a} \triangleq &\ \big(lane\_env = \textit{left} \rightarrow \\ &(ind\_env = none \ \vee \ ind\_env = \textit{right})\big),\end{aligned} \tag{17}$$

Then $\phi^e_3 \triangleq \phi^e_{3,a} \wedge \phi^e_{3,b}$.

Finally, the position update in assumption 4 is expressed with ϕ^e_4.

$$\phi^e_4 \triangleq pos\_env' = (pos\_env + vlc\_env) \mod 30. \tag{18}$$

The implementation of ϕ^e_4 is more elaborate because TuLiP lacks a modulo operator, but this can be implemented by using relational operators and simple arithmetic, see GitHub[1]. With 4 formalized, all assumptions of the GR(1) formula (1) are defined.

Due to lack of space and since the requirements ψ^s_{init} and ψ^s_{safe} on *ego-vehicle* are in principle the same as the assumptions ψ^e_{init} and ψ^e_{safe} made on *env-vehicle*, they are not included here, but the complete model is available on GitHub[1]. However, the expectation on *ego-vehicle* to be able to drive needs to be formalized. The GR(1) fragment provides one way of ensuring such progress, and that is to specify the liveness properties $\psi_{live,i}$ of (1). When a liveness property is included in the specification, it is assumed or guaranteed that $\psi_{live,i}$ is fulfilled infinitely often, or in other words, again and again, with any finite number of states in between. In this example, being able to drive is interpreted as guaranteeing that *ego-vehicle* will drive infinitely many laps, which is formalized as

$$\psi^s_{\textit{live},1} \triangleq (pos\_ego = 0), \quad \psi^s_{\textit{live},2} \triangleq (vlc\_ego > 0). \tag{19}$$

ψ^s_{init}, ψ^s_{safe}, $\psi^s_{\textit{live},1}$, and $\psi^s_{\textit{live},2}$ form the formulae for the guarantees of (1). The full formula (1) is the input to TuLiP.

B. Supremica

The model of the AD example using EFA in Supremica allows the usage of the same variables as defined and used in the TuLiP model. Hence, the initial and marked values for the EFA variables that describe the partial initial and marked states of both vehicles in the AD examples can be defined:

- $lane\_env^\circ = \textit{right}$.
- $pos\_env^\circ = 10$.
- $vlc\_env^\circ = 0$.
- $ind\_env^\circ = \textit{none}$, $ind\_env^m = \{\textit{none}\}$.
- $lane\_ego^\circ = \textit{right}$.
- $pos\_ego^\circ = 0$, $pos\_ego^m = \{0\}$.
- $vlc\_ego^\circ = 0$.

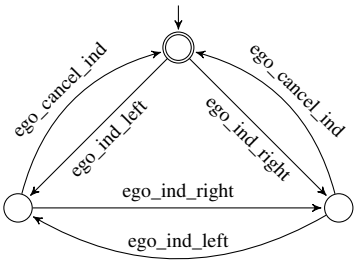
Authorized licensed use limited to: Chalmers University of Technology Sweden. Downloaded on June 05,2020 at 09:29:27 UTC from IEEE Xplore. Restrictions apply.

- $ind\_ego^{\circ} = \textit{none}$, $ind\_ego^{m} = \{\textit{none}\}$.

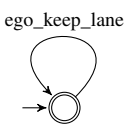
Note that if the set of marked values for a variable v is undefined, then $v^m = dom(v)$. With the variables defined, the plants and specifications can be modelled by EFA.

Plant In the SCT, the plant models all possible behavior. In this example the plant is defined by the synchronous composition of four sub-plants: the behavior of *ego-vehicle*, G_{ego}, the behavior of *env-vehicle*, G_{env}, together with the assumptions G_{asp} on *env-vehicle*, and the configuration G_{conf} describing the alternations of actions between the two vehicles.

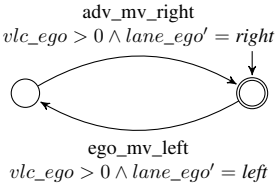
The sub-plant G_{ego} is the synchronous composition of the EFAs shown in Figure 2(a)-(c). As a plant, G_{ego} models all physically possible behavior of *ego-vehicle*.



(a) EFA of *ego-vehicle* indicator.



(b) EFA of *ego-vehicle* driving forward.



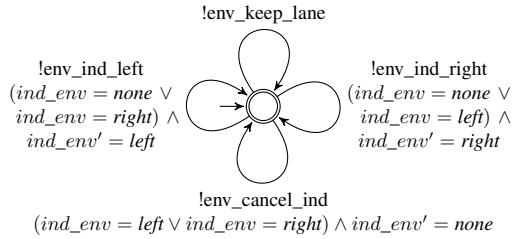
(c) EFA of *ego-vehicle* changing lanes.

Fig. 2: Plant G_{ego} modelling all physically possible behavior of *ego-vehicle*.

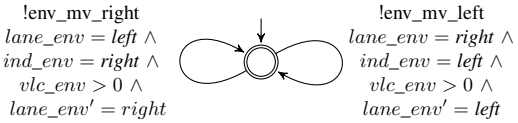
Similarly, the sub-plant G_{env}, which is the synchronous composition of the EFAs shown in Figure 3(a)-(b), models all physically possible behavior of *env-vehicle* in the example. This sub-plant is different from G_{ego} in two aspects. First, all events generated by G_{env} are uncontrollable, so the supervisor cannot disable them. Secondly, G_{env} only has a single location and all transitions are self-loops. This is because the variables for *env-vehicle* may change arbitrarily rather than being logically regulated. Having transitions as self-loops enables all possible events to occur.

The EFAs in Figure 4(a)-(b) together represent the sub-plant G_{asp}, which models the assumptions on *env-vehicle* regarding the longitudinal distance from *ego-vehicle*. In particular, the EFA of Figure 4(a) states that any event of *env-vehicle* can only occur if its current distance to *ego-vehicle* is more than the safety distance, or the vehicles are in different lanes. The EFA of Figure 4(b), on the other hand, states that when executing an event that can bring *env-vehicle* into the same lane as *ego-vehicle*, the updated distance is more than the safety distance if such event occurs.

The last sub-plant in the Supremica model is G_{conf}, shown

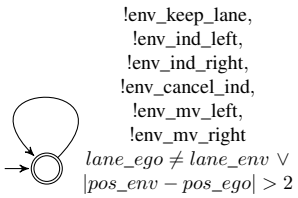


(a) EFA of *env-vehicle* staying in the same lane.

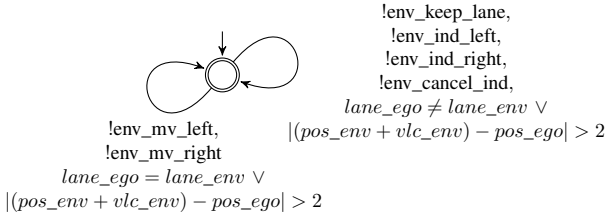


(b) EFA of *env-vehicle* changing lanes.

Fig. 3: Plant G_{env} modelling all the physically possible behavior of *env-vehicle*.



(a) EFA of *env-vehicle* checking the current longitudinal distance.



(b) EFA of *env-vehicle* checking the next longitudinal distance.

Fig. 4: Plant G_{asp} modelling the assumptions of *env-vehicle*.

in Figure 5, which models the configuration of each computation cycle. Upon the occurrence of the event set_vlc_env, variable vlc_env is set to an arbitrary value in its domain. Next, only one of the events of *env-vehicle* can occur and the longitudinal position in the circular road increases according to vlc_env. The two remaining transitions provide the same functionality, but for *ego-vehicle*.

Specification The Supremica model of the AD example includes two specifications K_{ind} and K_{dis}, where K_{ind} models the desired behavior of the indicators when changing lanes, and K_{dis} models the check of the safety distance. The EFAs representing K_{dis} are similar to those of G_{asp} in Figure 4 and are thus omitted. The EFA representing K_{ind} is shown in Figure 6; it states that a lane change of *ego-vehicle* should always be preceded by an activation and followed by a deactivation of the indicator.

Authorized licensed use limited to: Chalmers University of Technology Sweden. Downloaded on June 05,2020 at 09:29:27 UTC from IEEE Xplore. Restrictions apply.

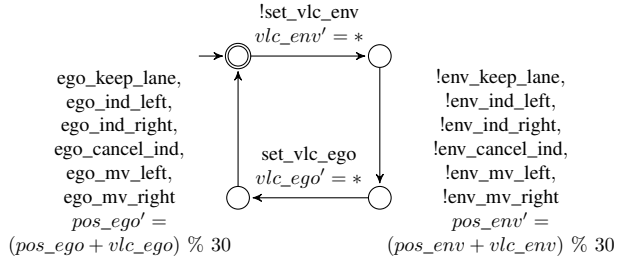


Fig. 5: Plant G_{conf} modelling the alternations of actions between the vehicles in each computation cycle.

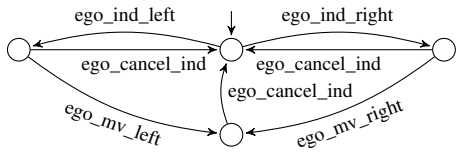


Fig. 6: Specification K_{ind} modelling the desired behavior of *ego-vehicle* indicating and lane changing.

V. Comparison

This section provides an empirical comparison between RS and SCT, based on the two case studies presented in Section III and Section IV. Admittedly there are many ways to model the same example and different ways have their different gains and drawbacks. We only compare the described models and make no claims that these are the "best" ways to model the examples.

Specification In RS, two involved components are the *environemnt* and the *system*, while in SCT they are the *plant* and the *specification*. The term *specification* has different meanings in the two formalisms. In SCT, the *specification* is a formalization of only the requirements; it is a component of the model that defines the (un-)desired behavior of the plant. In SCT, a *specification* can either be expressed by one or more automata, or simply by marking (or forbidding) some states of the plant. For example, the Supremica model of the stick-picking game does not have any automaton to express that player two should pick the last stick in order to let player one win. Instead, this desired behavior is expressed by marking $sticks = 0$ and the location *Player1Turn* in the EFA of Figure 1(a).

In RS however, *specification* is a synonym for the model, thus encompassing all dynamics and the entire set of requirements that the controller must fulfill. The LTL formulae in (16) and (17) directly express the requirements on how *ego-vehicle* shall activate the turn indicator in order to change lanes. In Supremica, these requirements are represented in two parts: the plant composed by the EFAs in Figure 2(a) and (c), and a specification EFA shown in Figure 6. In particular, the plant describes all possible behavior in terms of indicators and lane changing, which includes undesired sequences such as changing lane without first using the indicator.

Modelling write access to a shared variable The model of the stick-picking game in Supremica is quite compact in the sense that two 'variables' are used in the model ($sticks$ and an automaton). For TuLiP, four variables are used. This might be caused by how Supremica allows read/write access to the shared variable $sticks$. While TuLiP grants player two read access to $sticks$, it requires the variable $player\_2\_picks$ (or similar) as a mean for player two to get write access to $sticks$. In SCT, access to variables is determined by the controllability of the events of the transition where a variable appears, while in RS, access (particularly write access) to variables is determined by which of AP_e and AP_s contains the variable. This means that in situations similar to the stick-picking game, access to $sticks$ can be granted directly to either agent by the plant in SCT, while only one of the agents can have direct write access to $sticks$ in RS, since AP_e and AP_s cannot change based on state.

Modelling cycles The plant G_{conf} in Figure 5 is unparalleled in the TuLiP specification. It provides similar functionality as the plant in Figure 1(a); shifting turns between two agents (where the top two locations indicate that it is *env-vehicle*'s turn, and the bottom two indicate *ego-vehicle*'s turn). This concurrent behavior is an inherent property of the algorithm used by TuLiP to solve the synthesis problem, which can be solved by letting two agents play a turn-based game. However, in the stick-picking game, it is apparently difficult to let this inherent property deal with the player turns directly, likely because the need of shared and restricted write access (like a mutex) of the variable $sticks$.

Liveness Since the plant G_{conf} of Figure 5 is supposed to mimic TuLiP's infinite cycles of concurrency of *ego-vehicle* and *env-vehicle*, it needs a liveness property. To overcome the lack of liveness properties in SCT, G_{conf} is structured in a way such that a supervisor cannot prevent further execution cycles. This is achieved by having one marked location with an uncontrollable event on the outgoing transition. So, if the plant fires the uncontrollable event (which the supervisor cannot prevent), then the supervisor also has to allow the plant to fire the remaining events to complete the cycle (because of the marking). Although this technique does not produce infinite runs or liveness, it makes sure that, if the plant so chooses, there is always the possibility for another cycle.

The expectation in the AD example that *ego-vehicle* is able to drive is handled differently in TuLiP and Supremica. Liveness properties are required to make *ego-vehicle* move at all in TuLiP. The formulae in (19) are mere one way to make *ego-vehicle* move, but without any liveness properties TuLiP produces a solution where *ego-vehicle* stays at the initial position and never moves. When the liveness properties in (19) is included, however, TuLiP will notify the user if *ego-vehicle* would not be able to move.

In the Supremica model the expected movement of *ego-vehicle* is handled by marking $pos\_ego^m = \{0\}$ and relying on the non-blocking property of the SCT solution. This approach has the potential to give *ego-vehicle* a much richer set of behaviors to chose its actions from, since *ego-vehicle* might drive one step and then remain there forever as long as it is possible to reach $ego\_pos = 0$. The caveat, however,

Authorized licensed use limited to: Chalmers University of Technology Sweden. Downloaded on June 05,2020 at 09:29:27 UTC from IEEE Xplore. Restrictions apply.

is that, in general, a supervisor might be created that does not allow movement at all. In contrast to TuLiP's behavior, Supremica would not necessarily notify the user since movement is not required by any specification. However, movement is possible in this example, which is easy to check by simulating the model with its supervisor and observe that it is possible for *ego-vehicle* to move one step. In this particular case it might actually be possible to force progress by letting $vlc\_ego^m = \{1, 2\}$.

Contrary to the AD example, the TuLiP specification for the stick-picking game has no need for liveness properties although $sticks = 0$ is desired to be satisfied eventually. Because of RS's infinite runs and the formulae that makes sure that $sticks$ is decreased in every state, a liveness property would be superfluous. On the other hand, in Supremica, $(Player1Turn, sticks = 0)$ has to be the only marked state, otherwise player one may chose to stop playing if the next move is a losing move; it has to be possible to reach $sticks = 0$, if the plant desires.

Maximally Permissive Another key difference between Supremica and TuLiP is that in Supremica the solution is maximally permissive while in TuLiP any strategy that fulfills the specification is sufficient. In the stick-picking example there is exactly one winning strategy, thus TuLiP and Supremica synthesize the same strategy. For the autonomous drive example there are multiple strategies, but Supremica always generates the maximally permissive solution. This fact is not evident from the examples, but it is at least possible to ascertain that the strategy synthesized by TuLiP has one and only one behavior of *ego-vehicle* for each input sequence of *env-vehicle*, while the supervisor synthesized by Supremica often give several actions to chose from; Supremica's supervisor is more permissive than TuLiP's controller.

Analyzing solutions: Both TuLiP and Supremica let the user inspect the synthesized solution graphically and through simulation. However, in practice, these methods of inspection rapidly become infeasible as the number of locations or size of variable domains increase, even only slightly. For instance, even if the AD-example is scaled down to a single lane road with five cells, the synthesis result is difficult to manually interpret fully. This difficulty is a major obstacle for using SCT and RS by themselves, but also when comparing them.

VI. Conclusion

This paper provides a comparison between Reactive Synthesis and Supervisory Control Theory from a modelling perspective by considering two case studies, a stick-picking game and an autonomous driving example. Both RS and SCT allow for control synthesis of dynamic discrete-event systems, and they both have their individual strengths.

As a complement to previously published comparisons of SCT and RS that have focused on establishing a formal connection between SCT and RS, this paper's focus is on comparing two specific case studies. The problems have been modelled both in the RS synthesis tool TuLiP and the SCT synthesis tool Supremica. While the models are very different, the solutions generated by the two approaches still have many similarities.

Since RS and SCT have developed as separate subjects in different communities it is our aim to show, using concrete examples, what can be achieved with available tools from each community. We hope that this might be useful for researchers from both communities to learn more about the approach used by the other community and that this might stimulate new ideas in both research fields.

References

[1] O. Kupferman, P. Madhusudan, P. Thiagarajan, and M. Vardi, "Open systems in reactive environments: Control and synthesis," in *CONCUR 2000 — Concurrency Theory*, ser. Lecture Notes in Computer Science, vol. 1877. Springer, Berlin, Heidelberg, 2000.

[2] R. Bloem, R. Ehlers, S. Jacobs, and R. Könighofer, "How to handle assumptions in synthesis," in *Proceedings 3rd Workshop on Synthesis*, ser. Electronic Proceedings in Theoretical Computer Science, K. Chatterjee, R. Ehlers, and S. Jha, Eds., vol. 157. Vienna, Austria: Open Publishing Association, July 2014, pp. 34–50.

[3] P. Madhusudan, "Control and synthesis of open reactive systems," Ph.D. dissertation, University of Madras, 2001.

[4] T. Wongpiromsarn, U. Topcu, and R. M. Murray, "Receding horizon temporal logic planning," *IEEE Transactions on Automatic Control*, vol. 57, no. 11, pp. 2817–2830, 2012.

[5] B. Finkbeiner, "Synthesis of reactive systems," in *Dependable Software Systems Engineering*, vol. 45, 2016, pp. 72–98.

[6] P. J. G. Ramadge and W. M. Wonham, "The control of discrete event systems," *Proceedings of the IEEE*, vol. 77, no. 1, pp. 81–98, Jan 1989.

[7] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, 2nd ed. Springer, 2008.

[8] R. Ehlers, S. Lafortune, S. Tripakis, and M. Y. Vardi, "Supervisory control and reactive synthesis: A comparative introduction," *Discrete Event Dynamic Systems*, vol. 27, no. 2, pp. 209–260, Jun 2017.

[9] A.-K. Schmuck, T. Moor, and R. Majumdar, "On the relation between reactive synthesis and supervisory control of input/output behaviours," in *14th IFAC Workshop on Discrete Event Systems (WODES)*, vol. 51, 01 2018, pp. 31–38.

[10] P. J. G. Ramadge, "Some tractable supervisory control problems for discrete-event systems modeled by buchi automata," *IEEE Transactions on Automatic Control*, vol. 34, no. 1, pp. 10–19, Jan 1989.

[11] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science*, Oct 1977, pp. 46–57.

[12] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of Reactive(1) designs," in *Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, E. Emerson and K. Namjoshi, Eds., vol. 3855. Springer, Jan 2006, pp. 364 – 380.

[13] M. Sköldstam, K. Åkesson, and M. Fabian, "Supervisory control applied to automata extended with variables - revised," Chalmers University of Technology, Tech. Rep., 2008.

[14] 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 *IEEE Conference on Control Applications (CCA)*, Sept 2016, pp. 1030–1041.

[15] R. Malik, K. Åkesson, H. Flordal, and M. Fabian, "Supremica – an efficient tool for large-scale discrete event systems," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 5794 – 5799, 2017, 20th IFAC World Congress.

[16] T. Wongpiromsarn, U. Topcu, N. Ozay, H. Xu, and R. Murray, "TuLiP: A software toolbox for receding horizon temporal logic planning," in *Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control*, 04 2011, pp. 313–314.

[17] U. Klein and A. Pnueli, "Revisiting synthesis of GR(1) specifications," in *Hardware and Software: Verification and Testing*, S. Barner, I. Harris, D. Kroening, and O. Raz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 161–181.

Authorized licensed use limited to: Chalmers University of Technology Sweden. Downloaded on June 05,2020 at 09:29:27 UTC from IEEE Xplore. Restrictions apply.