Automatic Behavior Composition Synthesis

Giuseppe De Giacomo[a], Fabio Patrizi[a], Sebastian Sardiña[b]

[a]*Dipartimento di Informatica e Sistemistica - Sapienza Università di Roma - Rome, Italy.*
[b]*School of Computer Science and IT - RMIT University - Melbourne, Australia.*

Abstract

The behavior composition problem amounts to realizing a virtual desired module (e.g., a surveillance agent system) by suitably coordinating (and re-purposing) the execution of a set of available modules (e.g., a video camera, vacuum cleaner, a robot, etc.) In particular, we investigate techniques to synthesize a controller implementing a fully controllable target behavior by suitably coordinating available partially controllable behaviors that are to execute within a shared, fully observable, but partially predictable (i.e., non-deterministic), environment. Both behaviors and environment are represented as arbitrary finite state transition systems. The technique we propose is directly based on the idea that the controller job is to coordinate the concurrent execution of the available behaviors so as to "mimic" the target behavior. To this end, we exploit a variant of the formal notion of *simulation* to formally capture the notion of "mimicking," and we show that the technique proposed is sound and complete, optimal with respect to computational complexity, and robust for different kind of system failures. In addition, we demonstrate that the technique is well suited for highly efficient implementation based on synthesis by model checking technologies, by relating the problem to that of finding a winning strategy in a special safety game and explaining how to actually solve it using an existing verification tool.

Key words: Knowledge representation and reasoning, Intelligent agents, Reasoning about actions and change, Automated planning, Synthesis of reactive systems

1. Introduction

In this paper, we provide a thorough investigation—from theory to implementation—of the behavior composition problem, that is, the problem of how to realize an abstract desired target behavior module by reusing and re-purposing a set of accessible modules implementing certain concrete behaviors. More concretely, we are interested in *synthesizing* a sort of controller that coordinates the available existing behaviors in order to replicate a given desired target behavior [29, 77, 78]. Generally speaking, a behavior stands for the logic of any artifact that is able to operate in the environment, such as devices, agents, software or hardware components, or workflows. For example, consider a painting blocks-world scenario in which blocks are painted and processed by different robotic arms; different behaviors stand for different types of arms (e.g., a gripper, a painting arm, a cleaner arm, etc.), all acting

in the same environment. The aim is to realize a desired (intelligent) virtual painting system by suitably "combining" the available arms.

Behavior composition is of particular interest in agents and multi-agent settings. A (desired) intelligent system may be built, for example, from a variety of existing different modules operating (that is, performing actions) on a common environment and whose logic is only partially known. These modules may, in turn, be other agents themselves. A set of RoboCup players with different capabilities can be put together to form an (abstract) more sophisticated "team" player. Similarly, a BDI (Belief-Desire-Intention) agent may implement a desired deterministic plan (which was probably obtained via planning or agent communication) by appealing to the set of available user pre-defined non-deterministic plans [34, 73]. In robot-ecologies and ambient intelligence, advanced functionalities, such as a home surveillance agent, are achieved through the composition of many *simple* robotic devices, such as a vacuum cleaner, a lamp, or a video camera [74, 17].

Our work is really a form of process synthesis as studied in Computer Science [68, 1, 87, 49]. However, while most literature on synthesis concentrates on synthesizing a process satisfying a certain specification from scratch, behavior composition focuses on synthesizing a process (the controller) starting from available components [52]. This idea of composing and reusing components has been strongly put forward by Service Oriented Computing, under the name of "service composition" [2, 40, 61, 84]. Indeed, service composition aims at composing complex services by orchestrating (i.e., controlling and coordinating) services that are already at disposal. When service composition takes into account the behavior of the component service, as in [20, 82, 16] for instance, it becomes intimately related to what we call here "behavior composition."

When we look at behavior composition from an Artificial Intelligence perspective, the issue of actual controllability of the available behaviors becomes prominent. While one can instruct a behavior module to carry out an action, the actual outcome of the action may not be always foreseen a priori, though it can be possibly observed after execution. Our work here is based on revisiting a certain stream of work in service composition [13, 14, 15], called "Roman Model" in [40, 84], but keeping the need of dealing with partial controllability central. In particular, we consider the problem of synthesizing a *fully controllable* target behavior from a library of available *partially controllable* behaviors that are to execute within a shared, fully observable, but *partially predictable* environment [29, 77].

Technically, we abstract behaviors and the environment as finite state *transition systems*. More precisely, each available module is represented as a nondeterministic transition system (to model partial controllability); the target behavior is represented as a deterministic transition system (to model full controllability); and the environment is represented as a non-deterministic transition system (to model partial predictability). The environment's states are fully accessible by the other transition systems. Working with finite state transition systems allows us to leverage on research in Verification and Synthesis in Computer Science [67, 85, 48, 3, 23].

Once we settle for a formal specification of the problem of concern, we then develop a novel sound and complete, and computationally optimal technique to generate so-called *compositions*. The technique is directly based on the idea that a composition amounts to a controller that coordinates the concurrent execution of the avail-

able modules so as to "mimic" the desired target behavior. We capture "mimicking" through the formal notion of *simulation* [58, 39]. Obviously, we need to consider that available behaviors as well as the environment are only partially controllable (i.e., non-deterministic), and therefore a special variant of the classical notion of simulation ought to be devised.

The proposed technique has several interesting features:

- The technique is *sound and complete*, in a very strong sense: it allows to synthesize a sort of meta-controller, called *controller generator*, that represents all possible compositions. While the set of possible compositions is infinite (in fact uncountable) in general, the controller generator is unique.

- The technique gives us a very precise characterization of the sources of *complexity* in the problem. Whereas behaviour composition is known to be EXPTIME-hard even for deterministic available behaviors running in a stateless environment [59], the technique proposed here allows for computing the controller generator in time exponential in the number of available behavior, but not in the number of their states. In other words, computing the controller generator (i.e., an implicit representation of *all* compositions) is EXPTIME-complete and indeed exponential only in the number of available behaviors.

- Due to its "universality," the controller generator can be used to generate a sort of *lazy composition* on-the-fly, possibly adapting reactively based on runtime feedback.

  In particular, we shall argue that the composition solutions obtained are *robust* to behavior failures in two ways. First, they can handle *(a)* temporary behavior unavailability as well as *(b)* unexpected behavior/environment evolution in a totally *reactive* and *on-the-fly* manner—that is, without any extra effort or "re-planning" required to continue the realization of the target behavior—if at all possible, by the very nature of the composition generator. Second, the composition solutions can be *parsimoniously refined* when a module *(c)* becomes permanently unavailable, or *(d)* unexpectedly resumes operation.

We complement the proposed technique by showing how it can be implemented by making use of *model checking* technology applied to some special *game structures* developed in the context of Synthesis in Computer Science [3, 45, 38, 67, 26]. To that end, we show how to polynomially encode behavior compositions into *safety games* of a specific form, in which each strategy for winning the game corresponds to a composition (Section 5). With that reduction at hand, one is then able to use available tools such as TLV [69] in order to actually compute the controller generator by symbolic model checking (Section 6).

The rest of the paper is organized as follows. In Section 2 we spell out our framework for behavior composition. In Section 3, we provide our technique based on simulation for synthesizing compositions, and we detail the notion of controller generator. In Section 4, we show how the approach can deal with behavior failures. Then, in Section 5, we turn to synthesis by model checking, and show how one can compute the

controller generator through safety games. Based on the results of the previous sections, we show in Section 6 how to implement behavior composition in practice using existing platforms for synthesis by model checking such as TLV [69]. (The full TLV code for our running example is reported in an appendix.) We discuss related work in various areas of Artificial Intelligence and Computer Science in Section 7, and draw conclusions in Section 8.

2. The Framework

In this section, we formally define the problem of concern, by developing an abstract framework based on (sort of) finite state transition systems.

Environment. We assume to have a shared fully observable *environment* which provides an abstract account of action preconditions and effects, and can be regarded as a mean of communication among behaviors (defined below). As, in general, we have *incomplete information* about preconditions and effects (akin to an action theory), the environment can, in general, be *non-deterministic*.

Formally, an *environment* is a tuple $\mathcal{E} = \langle \mathcal{A}, E, e_0, \rho \rangle$, where:

- $\mathcal{A}$ is a finite set of *shared actions*;
- E is the finite set of *environment states*;
- $e_0 \in E$ is the *environment initial state*;
- $\rho \subseteq E \times \mathcal{A} \times E$ is the *environment transition relation* among states.

When referring to environment transitions, we equivalently use notations $\langle e, a, e' \rangle \in \rho$ or $e \xrightarrow{a} e'$ (in $\mathcal{E}$), both denoting that performing action a in state e may lead the environment to successor state e'.

Observe that this notion of environment shares a lot of similarities with so-called "transition systems" in action languages [32]; indeed, that formalism might well be used to compactly represent the environment, in our setting.

Behaviors. A *behavior* abstracts the program of some agent (or, more in general, the logic of a device/module), in terms of (internal) states, actions and transitions. Behaviors are not intended to execute on their own but, rather, to operate within an environment (and, through this, possibly interact with other behaviors). Hence, they are equipped with the ability to test, when needed, conditions (or *guards*) on environment states.

Formally, a *behavior* over an environment $\mathcal{E}$ is a tuple $\mathcal{B} = \langle B, b_0, G, F, \varrho \rangle$, where:

- B is the finite set of *behavior states*;
- $b_0 \in B$ is the *behavior initial state*;
- G is a set of *guards over* $\mathcal{E}$, that is, boolean functions $g : E \mapsto \{\top, \bot\}$;
- $F \subseteq B$ is the set of *behavior final states*;

- $\varrho \subseteq B \times G \times \mathcal{A} \times B$ is the *behavior transition relation*.

We freely interchange notations $\langle b, g, a, b' \rangle \in \varrho$, and $b \xrightarrow{g,a} b'$ in $\mathcal{B}$. A "guarded" transition $\langle b, g, a, b' \rangle \in \varrho$ denotes that: *(i)* action a can be executed by $\mathcal{B}$ in state b when the environment is in a state e such that $g(e) = \top$; and *(ii)* the execution may lead the behavior to successor state b'. Notice that a behavior's evolution depends on the environment it is defined over, as action executability depends on guard satisfaction.

Intuitively, behavior states model agent's decision points: when the behavior is in a given state, the agent selects the action to be executed next among those executable [1] at that state. Executing the selected action, besides other effects, leads the behavior to a successor state, where a new set of actions become executable, and a new iteration starts. Final states are those where the behavior can be safely stopped (e.g., final states of a mechanic arm might correspond to safe configurations).

We say that a behavior $\mathcal{B}$ over environment $\mathcal{E}$ is *deterministic* if no behavior and environment states exist, say $b \in B$ and $e \in E$, respectively, for which two transitions $b \xrightarrow{g_1,a} b'$ and $b \xrightarrow{g_2,a} b''$ exist such that $b' \neq b''$ and $g_1(e) = g_2(e) = \top$.

Clearly, given a *deterministic* behavior's and an environment's states, and an executable action, *the* next behavior state is always predictable. In other words, deterministic behaviors are *fully controllable* by appropriate action selections. In general, however, behaviors are *non-deterministic*, that is, the state resulting from an action execution is unpredictable, and, thus, so are the actions that will be available in such a state. In other words, non-deterministic behaviors are only *partially controllable*.

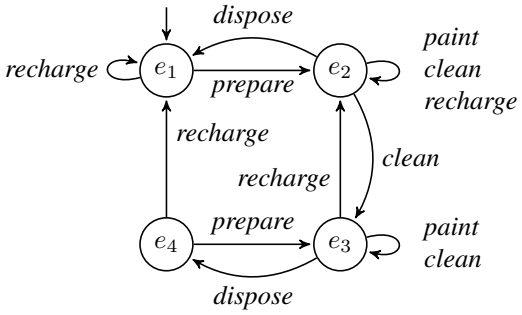
System and Target Behavior. As said above, behaviors operate within an environment (the one they are defined over) and can, through this, interact with each other. The notion of *system* introduced below allows for identifying a set of interacting behaviors over the same environment.

The *system* is a tuple $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$, where $\mathcal{E}$ is an environment and $\mathcal{B}_1, \ldots, \mathcal{B}_n$ are predefined, possibly non-deterministic, *available behaviors* over $\mathcal{E}$. We stress that available behaviors are given and cannot be modified, though they can, of course, be (partially) controlled through action execution. The behaviors of a system model the only available implementations one can actually use to execute actions. Importantly, a behavior cannot be instructed to execute actions regardless of its (and environment's) current state, but needs to be in a state where the desired action is actually executable; external controllers must, of course, take these constraints into account when coordinating a set of behaviors.

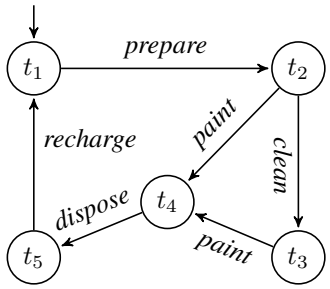
Finally, we define the so-called *target behavior* $\mathcal{B}_T$ as a *deterministic* behavior over $\mathcal{E}$, which represents the fully controllable desired behavior to be obtained. Roughly speaking, the challenge we deal with here is to bring about the "virtual" (i.e., non-readily available) target behavior by properly "composing" the execution of available behaviors.

Example 1. In the *painting arms scenario* depicted in Figure 1, the overall aim of the system is to process blocks. Only one block at a time can be processed: it can

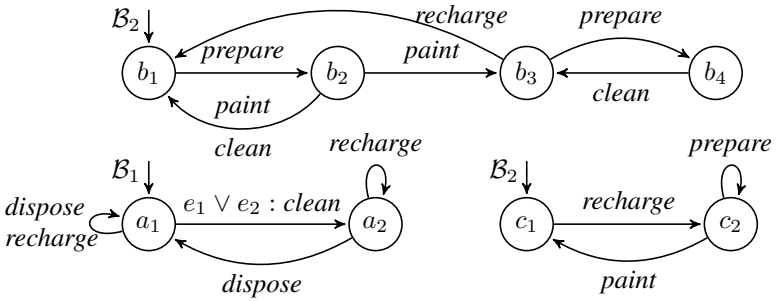
[1] Subject to environment's current state.



(a) Environment $\mathcal{E}$.



(b) Target arm $\mathcal{B}_T$.



(c) Available arms $\mathcal{B}_1$, $\mathcal{B}_2$, and $\mathcal{B}_3$.

Figure 1: The painting arms system $\mathcal{S} = \langle \mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3, \mathcal{E} \rangle$ and the target arm $\mathcal{B}_T$.

be cleaned or painted, but needs first to be prepared. After preparation, cleaning and painting can be performed when water and paint, stored in two different tanks, are (respectively) available. Both tanks can be charged simultaneously by pushing a button. Blocks can also be cleaned, but only in particular circumstances (i.e., environment in state e_3, see below).

The non-deterministic environment $\mathcal{E}$ provides general domain's rules. Nodes and edges represent states and transitions, respectively; each edge label represents the action that triggers the transition; and the initial state has an incoming edge without source. For instance, as said, blocks can be painted or cleaned *only* after they have been prepared: so, from e_1, a state where either action *paint* or *clean* is enabled (either e_2 or e_3) can only be reached by first executing *prepare*. Though not graphically represented, the environment accounts for tank states, e.g.: in e_1 and e_2 the water tank is not empty, while it is in e_3 and e_4. Action *clean* can also be performed in e_3, even though the water tank is empty, as in this state a cleaning tool not relying on water becomes available.

$\mathcal{B}_T$ describes the (deterministic) behavior of a desired (target) arm-agent module. Observe that state t_2 captures a decision point: cleaning a block is optional, as the selection of the transition is demanded to the executor, which makes its decisions ac-

cording to internal policies –e.g., ensuring first that the block is dirty. Also, notice that $\mathcal{B}_T$ is "conservative," in that it always recharges the tanks after processing a block, so as to guarantee that *clean* will be executable, if needed.

The desired arm $\mathcal{B}_T$ *does not* exist in reality. Nonetheless, there are three different actual arms available: $\mathcal{B}_1$ (states a_1, a_2), a cleaning-disposing arm able to clean and dispose blocks; $\mathcal{B}_2$ (states $b_1, \ldots, b_4$), capable of preparing, cleaning, and painting blocks; and $\mathcal{B}_3$ (states c_1, c_2), a paint arm that can also prepare blocks for processing. All three arms are able to press the charge button (to refill the tanks). Notice that arm $\mathcal{B}_2$ behaves non-deterministically when it comes to painting a block. This non-determinism captures modeler's incomplete information about $\mathcal{B}_2$'s internal logic. Observe also that arm $\mathcal{B}_1$ requires the environment to be in e_1 or e_2, in order to perform *clean*, as it needs water to actually execute the action.

In this example, all behavior states are assumed final, thus imposing no restrictions on when the execution can be stopped. ■

Next, we derive the notions of behavior and system *enactment*, which are abstract structures needed to formally state the composition problem and characterize its solutions.

Enacted behaviors. Behaviors and the environment mutually affect their executions. Such a "combined" evolution is formally described by *enacted behaviors*. Given a behavior $\mathcal{B} = \langle B, b_0, G, F, \varrho \rangle$ over an environment $\mathcal{E} = \langle \mathcal{A}, E, e_0, \rho \rangle$, the *enacted behavior* of $\mathcal{B}$ on $\mathcal{E}$ is a tuple $\mathcal{T}_\mathcal{B} = \langle S, \mathcal{A}, s_0, Q, \delta \rangle$, where:

- $S = B \times E$ is the (finite) set of $\mathcal{T}_\mathcal{B}$'s states, where for each state $s = \langle b, e \rangle \in S$, we denote b as *beh*(s) and e as *env*(s);
- $\mathcal{A}$ is the same set of actions as in $\mathcal{E}$;
- $s_0 \in S$ is the initial state of $\mathcal{T}_\mathcal{B}$, such that *beh*$(s_0) = b_0$ and *env*$(s_0) = e_0$;
- $\delta \subseteq S \times \mathcal{A} \times S$ is the enacted transition relation, where $\langle s, a, s' \rangle \in \delta$ or, equivalently, $s \xrightarrow{a} s'$ in $\mathcal{T}_\mathcal{B}$, if and only if:
  - *env*$(s) \xrightarrow{a}$ *env*(s') in $\mathcal{E}$, that is, action a is actually executable in $\mathcal{E}$;
  - *beh*$(s) \xrightarrow{g,a}$ *beh*(s') in $\mathcal{B}$, with $g($*env*$(s)) = \top$ for some $g \in G$, that is, action a can be performed by $\mathcal{B}$ from its state *beh*(s) when the environment state *env*(s) satisfies the guard which labels the respective transition.
- $Q = \{s \in S \mid$ *beh*$(s) \in F\}$ is the set of the enacted behavior's final states.

Technically, $\mathcal{T}_\mathcal{B}$ is the synchronous product of the behavior and the environment, and represents all possible executions obtained from running behavior $\mathcal{B}$ once guards are evaluated and actions are performed in $\mathcal{E}$. Observe that the enacted behavior non-determinism stems from both environment's and behavior's. Moreover, notice that action executability for a behavior is subject to: *(i)* its own state; *(ii)* guard evaluation in current environment state; and *(iii)* the environment state itself. In particular, even though a transition labeled with action a and outgoing from current behavior ($\mathcal{B}$) state

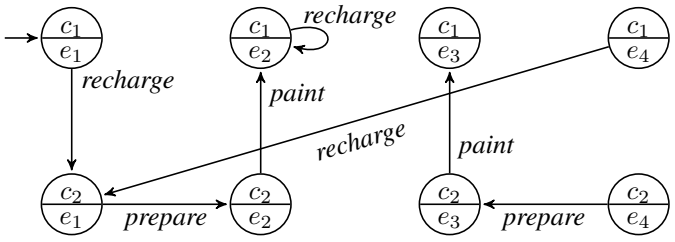


Figure 2: Enacted Arm $\mathcal{T}_3$.

exists, if, given current environment state e, no transition outgoing from e is labelled with a, then $\mathcal{B}$ cannot execute a –as if its precondition were not satisfied. In the following, when no ambiguity arises, we simplify the notation by denoting the enacted counterpart of a behavior $\mathcal{B}_i$ simply as $\mathcal{T}_i$, instead of $\mathcal{T}_{\mathcal{B}_i}$.

Example 2. The enacted behavior $\mathcal{T}_3$ depicted in Figure 2 describes the evolution of arm $\mathcal{B}_3$ if it were to act alone in the environment. Observe that there exist some joint states that cannot be reached by $\mathcal{B}_3$ alone. For instance, $\langle c_1, e_4 \rangle$ can be reached only by executing action *clean* which, however, is not available in $\mathcal{B}_3$. ■

Enacted system behavior. The *enacted system behavior* formally captures the concurrent, interleaved, execution of *all* available behaviors on the environment of a system. Let $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$ be a system, where $\mathcal{E} = \langle \mathcal{A}, E, e_0, \rho \rangle$ and $\mathcal{B}_i = \langle B_i, b_{i0}, G_i, F_i, \varrho_i \rangle$ $(i = 1, \ldots, n)$. The *enacted system behavior* of $\mathcal{S}$ is a tuple $\mathcal{T}_\mathcal{S} = \langle S_\mathcal{S}, \mathcal{A}, \{1, \ldots, n\}, s_{\mathcal{S}0}, Q_\mathcal{S}, \delta_\mathcal{S} \rangle$, where:

- $S_\mathcal{S} = B_1 \times \cdots \times B_n \times E$ is the finite set of $\mathcal{T}_\mathcal{S}$ states; given $s_\mathcal{S} = \langle b_1, \ldots, b_n, e \rangle$, we denote b_i as $beh_i(s_\mathcal{S})$ $(i = 1, \ldots, n)$ and e as $env(s_\mathcal{S})$;
- $s_{\mathcal{S}0} \in S_\mathcal{S}$ is the initial state of $\mathcal{T}_\mathcal{S}$, such that $beh_i(s_{\mathcal{S}0}) = b_{i0}$ $(i = 1, \ldots, n)$ and $env(s_{\mathcal{S}0}) = e_0$;
- $Q_\mathcal{S} = \{s_\mathcal{S} \in S_\mathcal{S} \mid \forall i \in \{1, \ldots, n\}\ beh_i(s_\mathcal{S}) \in F_i\}$ is the set of $\mathcal{T}_\mathcal{S}$ final states;
- $\delta_\mathcal{S} \subseteq S_\mathcal{S} \times \mathcal{A} \times \{1, \ldots, n\} \times S_\mathcal{S}$ is $\mathcal{T}_\mathcal{S}$'s transition relation, where $\langle s_\mathcal{S}, a, k, s'_\mathcal{S} \rangle \in \delta_\mathcal{S}$ or, equivalently, $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$, if and only if:
  - $env(s_\mathcal{S}) \xrightarrow{a} env(s'_\mathcal{S})$ in $\mathcal{E}$;
  - $beh_k(s_\mathcal{S}) \xrightarrow{g,a} beh_k(s'_\mathcal{S})$ in $\mathcal{B}_k$, with $g(env(s_\mathcal{S})) = \top$, for some $g \in G_k$;
  - $beh_i(s_\mathcal{S}) = beh_i(s'_\mathcal{S})$, for $i \in \{1, \ldots, n\} \setminus \{k\}$.

The enacted system behavior $\mathcal{T}_\mathcal{S}$ is technically the synchronous product of: *(i)* the environment, and *(ii)* the asynchronous product of the available behaviors. Except for the presence of index k in transitions, which identifies the behavior that performs the labeling action, it is formally analogous to an enacted behavior.

Controller. We are now ready to introduce the main component of our framework: the *controller*, which models an entity able to instruct available behaviors to execute actions, as well as to activate, stop, and resume their execution. We assume the controller has *full observability* on both available behaviors and the environment, that is, it can keep track, at runtime, of their current states. Although other choices are possible, full observability is quite natural in this context, since available behaviors and environment are already suitable abstractions of *actual* modules: if details have to be hidden, this can be done directly within the exposed abstract behaviors, by resorting to non-determinism.

In order to formally define controllers, we start with the notions of *traces* and *histories*. Let $\mathcal{T}_\mathcal{B} = \langle S, \mathcal{A}, s_0, Q, \delta \rangle$ be an enacted behavior of some (available or target) behavior $\mathcal{B}$ over environment $\mathcal{E}$. A *trace* for $\mathcal{T}_\mathcal{B}$ is a possibly infinite sequence $\tau = s^0 \xrightarrow{a^1} s^1 \xrightarrow{a^2} \cdots$, such that *(i)* $s^0 = s_0$; and *(ii)* $s^j \xrightarrow{a^{j+1}} s^{j+1}$ in $\mathcal{T}_\mathcal{B}$, for all $j \geq 0$. A *history* is just a *finite* prefix (ending with a state) $h = s^0 \xrightarrow{a^1} \cdots \xrightarrow{a^\ell} s^\ell$ of a trace. We denote h's last state s^ℓ by *last*(h), and its length ℓ by $|h|$. As finite traces are also histories, function $|\cdot|$ is also defined over them; if τ is an infinite trace, we let $|\tau| = \infty$.

Traces and histories extend immediately to enacted system behaviors, by adding index k. System traces have the form $s^0 \xrightarrow{a^1,k^1} s^1 \xrightarrow{a^2,k^2} \cdots$, and system histories have the form $s^0 \xrightarrow{a^1,k^1} \cdots \xrightarrow{a^\ell,k^\ell} s^\ell$. Functions $|\cdot|$ and *last* are extended in the obvious way.

Now, consider a system $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$ and its enacted behavior $\mathcal{T}_\mathcal{S}$. Let $\mathcal{H}$ be the set of all $\mathcal{T}_\mathcal{S}$ histories. A *controller* for $\mathcal{S}$ is a possibly partial function $P : \mathcal{H} \times \mathcal{A} \mapsto \{1, \ldots, n\}$. Intuitively, $P(h, a)$ identifies the available behavior, i.e., $\mathcal{B}_{P(h,a)}$, to delegate action a to, after $\mathcal{S}$ has evolved as described by enacted system behavior history h.

The Behavior Composition Problem. Roughly speaking, the problem we deal with is that of synthesizing, for a given system $\mathcal{S}$, a controller that *realizes* a desired target behavior, that is, able to coordinate the available behaviors so that the resulting behavior is, in fact, analogous to the target. In order to formalize this notion, we first need to define *trace realizations*. Let $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$ be a system, $\mathcal{B}_T$ a target behavior, and P a controller for $\mathcal{S}$. Furthermore, let $\mathcal{H}$ be the set of all $\mathcal{T}_\mathcal{S}$ (enacted system behavior) histories, and consider a $\mathcal{T}_T$ (enacted target behavior) trace $\tau = s^0 \xrightarrow{a^1} s^1 \xrightarrow{a^2} \ldots$. We say that *$P$ realizes τ* if:

- for all $\mathcal{T}_\mathcal{S}$ histories $h \in \mathcal{H}_{\tau,P} \subseteq \mathcal{H}$ ($\mathcal{H}_{\tau,P}$ defined below): if $|h| < |\tau|$, then $P(h, a^{|h|+1}) = k$ and *last*$(h) \xrightarrow{a^{|h|+1},k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ for some $s'_\mathcal{S}$, where $\mathcal{H}_{\tau,P} = \bigcup_{\ell \geq 0} \mathcal{H}^\ell_{\tau,P}$ is the set of *$\mathcal{T}_\mathcal{S}$ histories induced by P and τ*, inductively defined as follows:
  - $\mathcal{H}^0_{\tau,P} = \{s_{\mathcal{S}0}\}$;
  - $\mathcal{H}^{j+1}_{\tau,P}$ is the set of all $(j+1)$-length histories $h \xrightarrow{a^{j+1},k^{j+1}} s^{j+1}_\mathcal{S}$ such that:

  - $h \in \mathcal{H}^{j}_{\tau,P}$;
  - $env(s^{j+1}_{\mathcal{S}}) = env(s^{j+1})$;
  - $k^{j+1} = P(h, a^{j+1})$, that is, at history h, action a^{j+1} is delegated to available behavior $\mathcal{B}_{k^{j+1}}$;
  - $last(h) \xrightarrow{a^{j+1},k^{j+1}} s^{j+1}$ in $\mathcal{T}_{\mathcal{S}}$, that is, behavior $\mathcal{B}_{k^{j+1}}$ can actually execute action a^{j+1};

- if τ is finite and $s^{|\tau|} \in Q_T$ (i.e., $beh(s^{|\tau|})$ is final for $\mathcal{B}_T$), then all $|\tau|$-length histories $h \in \mathcal{H}^{|\tau|}_{\tau,P}$ are such that $last(h) \in Q_{\mathcal{S}}$.

Informally, saying that a controller realizes a target behavior trace means that: given a (possibly infinite) sequence of actions compliant with the target behavior, and a possible environment evolution resulting from the execution of such action sequence, the controller selects at each step of execution a behavior able to actually execute the action requested at that step, no matter how behaviors –which are non-deterministic– selected earlier evolved. In addition, if the target trace finishes at a final state (for the enacted target behavior), then the whole system is brought to a legal terminating state, too. In other words, the controller is *always* able to delegate the actions so as to mimicking the target behavior.

Because a deterministic behavior itself can be seen as a specification of a set of traces, we say that *a controller P realizes a target behavior $\mathcal{B}_T$* if and only if it realizes all traces of $\mathcal{T}_T$. This can be informally rephrased as the ability to delegate, step by step, all target behavior's action sequences, no matter how the environment and the available behaviors evolve.

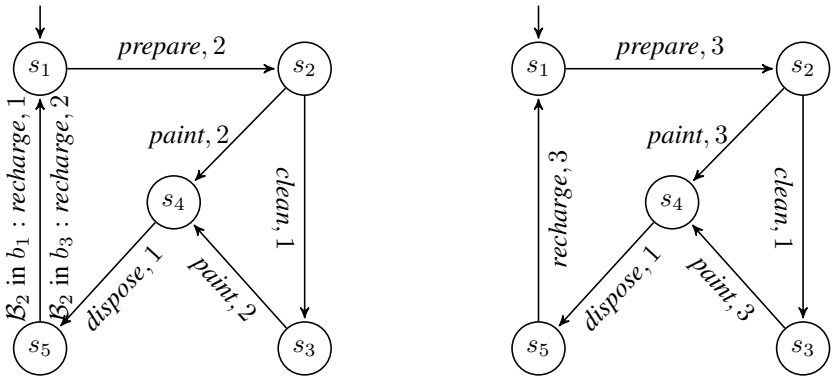
Observe that the controller can observe the current states of the available behaviors as well as that of the environment (in fact, it can observe the whole system history up to the current state), in order to decide which behavior to select next. This makes these controllers akin to an advanced form of conditional plans and, in fact, the problem itself is related to planning [37], being both synthesis tasks. Here, though, we are not planning for choosing the next action, but for *who shall execute the next action*, whatever such action happens to be at runtime. Formally, the problem that we deal with is as follows:

> Given a system $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$ and a deterministic *target* behavior $\mathcal{B}_T$ over $\mathcal{E}$, synthesize a controller P that realizes $\mathcal{B}_T$.

All controllers that are a solution to this problem are called *compositions* (of $\mathcal{B}_T$ on $\mathcal{E}$).

Example 3. Even though compositions are, in general functions of system histories (and actions), there are cases where they depend on history last k (≥ 0) states only. In such cases, they can be represented as finite-state machines. In Figure 3, for instance, two *finite-state* controllers, P_1 and P_2, are depicted. An edge outgoing from a state s and labeled with a pair $c : \langle a, k \rangle$ means that when the controller is in state s and action a is requested, it is delegated to behavior $\mathcal{B}_k$, provided condition c holds (omitted conditions are assumed true).

The main difference between P_1 and P_2 is in the arm used for painting: P_1 uses $\mathcal{B}_2$, while P_2 uses $\mathcal{B}_3$. In addition, P_1 charges the tanks using either $\mathcal{B}_1$ or $\mathcal{B}_2$, in



(a) Controller P_1 (b) Controller P_2

Figure 3: Two finite-state controllers.

particular: if $\mathcal{B}_2$ is in state b_1 then $\mathcal{B}_1$ is used, while if $\mathcal{B}_2$ is in b_3 then $\mathcal{B}_2$ is used. As for controller P_2, instead, it uses always $\mathcal{B}_3$. It can be easily seen that P_1 realizes all of $\mathcal{T}_T$ traces, and, therefore that it is a composition of $\mathcal{B}_T$ on $\mathcal{E}$. On the contrary, this is not the case for P_2, as it does not even realize the simple $\mathcal{T}_T$ one-action trace $\langle t_1, e_1\rangle \stackrel{prepare}{\longrightarrow} \langle t_2, e_2\rangle$. Finally, to see the usefulness of conditions, take a controller P_1' analogous to P_1, excepting for the edge from s_5 to s_1, which is re-labeled with just "*recharge*, 1" (i.e., action *recharge* is *always* delegated to $\mathcal{B}_1$). In such a case, P_1' would realize all those traces where $\mathcal{B}_2$ *always* evolves to b_1, after executing *paint*. However, as, in general, $\mathcal{B}_2$ may well evolve to b_3,too , there exist also $\mathcal{B}_T$ traces that cannot be realized by P_1'. Clearly, as P_1 shows, this can be easily avoided by a properly conditioning action delegation, based on behavior states. Notice also that this possibility is guaranteed by the assumption of full observability on state of the (whole) system. ■

This concludes the formal statement of the behavior composition problem. The framework just presented stands for what can be considered the *"core"* framework, i.e., a basic setting that incorporates all distinguishing features of the problem. However, we stress that extensions and generalization can be defined so as to obtain non-trivial variants, which can be adopted to model and solve similar problems from domains that satisfy different assumptions (see Section 8 for a discussion on this).

3. Composition via Simulation

Next, we present our approach to composition synthesis. This is originally inspired by [15], where a restricted version of the composition problem was addressed, in the context of services, by taking the standard notion of *simulation relation* [58, 39] as a formal tool for solution characterization. Here, the shared environment and the (devilish) non-determinism of both the available behaviors and the environment significantly sophisticate that framework, calling for a new formal setting, the one presented here,

where the usual notion of simulation relation is no longer enough to fully characterize the set of solutions and, hence, to guide the solution process.

Intuitively, we say that a transition system S_1 *simulates* another transition system S_2, if S_1 is able to "match", step by step, all of S_2 moves during execution. More precisely, imagine to execute S_2 starting from its initial state. At each step of execution, S_2 performs a transition among those allowed in its (current) state. If, for all possible ways of executing S_2, S_1 can, at each step, choose a transition that "matches" (according to some criteria, e.g., label equivalence) the one executed by S_2 then S_1 simulates S_2. We stress that S_1 decisions are required to be made in an "online" fashion, as S_2 evolves. In other words, it is not the case that S_1 knows in advance which transitions S_2 will execute in the future.

Such an intuition is formalized in the following definition, where both non-determinism and the shared environment are taken into account.

Let $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$ be a system, $\mathcal{B}_T$ a target behavior over $\mathcal{E}$, and $\mathcal{T}_\mathcal{S} = \langle S_\mathcal{S}, \mathcal{A}, \{1, \ldots, n\}, s_{\mathcal{S}0}, Q_\mathcal{S}, \delta_\mathcal{S} \rangle$ and $\mathcal{T}_T = \langle S_T, \mathcal{A}, s_{T0}, Q_T, \delta_T \rangle$ the enacted system and enacted target behaviors corresponding to $\mathcal{S}$ and $\mathcal{B}_T$ on $\mathcal{E}$, respectively. An *ND-simulation relation* of $\mathcal{T}_T$ by $\mathcal{T}_\mathcal{S}$ is a relation $R \subseteq S_T \times S_\mathcal{S}$, such that $\langle s_T, s_\mathcal{S} \rangle \in R$ implies:

1. $\mathit{env}(s_T) = \mathit{env}(s_\mathcal{S})$;
2. if $s_T \in Q_T$, then $s_\mathcal{S} \in Q_\mathcal{S}$;
3. for all $a \in \mathcal{A}$, there exists a $k \in \{1, \ldots, n\}$—also referred to as a *witness* of $\langle s_T, s_\mathcal{S} \rangle \in R$ for action a—such that for all transitions $s_T \xrightarrow{a} s'_T$ in $\mathcal{T}_T$:
   (a) there exists a transition $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ with $\mathit{env}(s'_\mathcal{S}) = \mathit{env}(s'_T)$;
   (b) for all transitions $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ with $\mathit{env}(s'_\mathcal{S}) = \mathit{env}(s'_T)$, it is the case that $\langle s'_T, s'_\mathcal{S} \rangle \in R$.

In words, if a pair of enacted states is in the ND-simulation (relation), then: *(i)* its states share the same environment component; *(ii)* if the target behavior is in a final state, so does the system; and *(iii)* for all actions the (enacted) target behavior can execute, there exists a *witness* behavior $\mathcal{B}_k$ that can execute the same action while guaranteeing, regardless of non-determinism, preservation of the ND-simulation relation for successor target and system states.

We say that a state $s_T \in S_T$ *is ND-simulated by* a state $s_\mathcal{S} \in S_\mathcal{S}$ (or $s_\mathcal{S}$ *ND-simulates* s_T), denoted $s_T \preceq s_\mathcal{S}$, if there exists an ND-simulation relation R of $\mathcal{T}_T$ by $\mathcal{T}_\mathcal{S}$ such that $\langle s_T, s_\mathcal{S} \rangle \in R$. Observe that this is a co-inductive definition. As a result, the relation $\preceq$ is itself an ND-simulation relation, in fact the *largest* one, in the sense that all ND-simulations are contained in $\preceq$.

Given $\mathcal{T}_T$ and $\mathcal{T}_\mathcal{S}$, relation $\preceq$ can be computed by Algorithm 1 (*NDS*). Roughly speaking, the algorithm works by iteratively removing those tuples for which the requirements of the ND-simulation definition do not apply, until a fixpoint is reached. It is straightforward to prove that the algorithm reaches a fixpoint in a finite number of steps and computes the largest ND-simulation, by comparing the algorithm with the definition of ND-simulation relation and observing that no tuple is ever added to the candidate set $\mathcal{R}$, and that $\mathcal{C} \subseteq \mathcal{R}$.

Algorithm 1: $NDS(\mathcal{T}_T, \mathcal{T}_\mathcal{S})$ – Largest ND-Simulation

$\mathcal{R} := S_T \times S_\mathcal{S} \setminus \{\langle s_T, s_\mathcal{S}\rangle \mid env(s_T) \neq env(s_\mathcal{S}) \vee (s_T \in Q_T \wedge s_\mathcal{S} \notin Q_\mathcal{S})\}$;
repeat
  $\mathcal{R} := (\mathcal{R} \setminus \mathcal{C})$, where $\mathcal{C}$ is the set of $\langle s_T, s_\mathcal{S}\rangle \in \mathcal{R}$ such that there exists an action $a \in \mathcal{A}$ and for each k there exists a transition $s_T \xrightarrow{a} s'_T$ in $\mathcal{T}_T$ such that either:
    (a) there is no transition $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ such that $env(s'_T) = env(s'_\mathcal{S})$; or
    (b) there exists a transition $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ such that $env(s'_T) = env(s'_\mathcal{S})$ but $\langle s'_T, s'_\mathcal{S}\rangle \notin \mathcal{R}$.
until $(\mathcal{C} = \emptyset)$;
return $\mathcal{R}$;

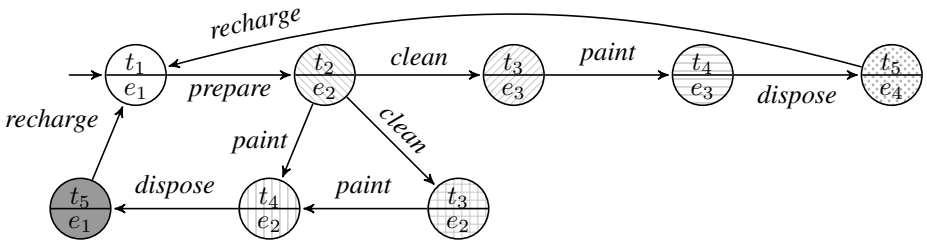
Example 4. Figure 4 shows a fragment of the largest ND-simulation relation for our painting blocks world example. In particular, Figure 4(a) shows the enacted target behavior of $\mathcal{B}_T$ and Figure 4(b) depicts a fragment of the system enacted behavior. States in Figure 4(b) contain, in the bottom half, the environment component, and, in the top half, a compact representation of available service (current) states: the first component of the integer string represents the subscript of the state that $\mathcal{B}_1$ is in, the second refers to $\mathcal{B}_2$, and so on. For instance, the node labeled with $\langle 211, e_4\rangle$ represents the system state $\langle\langle a_2, b_1, c_1\rangle, e_4\rangle$.

Matching patterns between $\mathcal{T}_T$ and $\mathcal{T}_\mathcal{S}$ states mean that such states are in ND-simulation. For example, $\langle\langle a_1, b_3, c_2\rangle, e_2\rangle$ of $\mathcal{T}_\mathcal{S}$ *ND-simulates* $\langle t_2, e_2\rangle$ of $\mathcal{T}_T$; this implies that *(i)* every conceivable action taken in $\langle t_2, e_2\rangle$ can be replicated by some behavior (possibly a different one for each action) when the system is in $\langle\langle a_1, b_3, c_2\rangle, e_2\rangle$ and, moreover, that *(ii)* this property propagates to the resulting successor states.

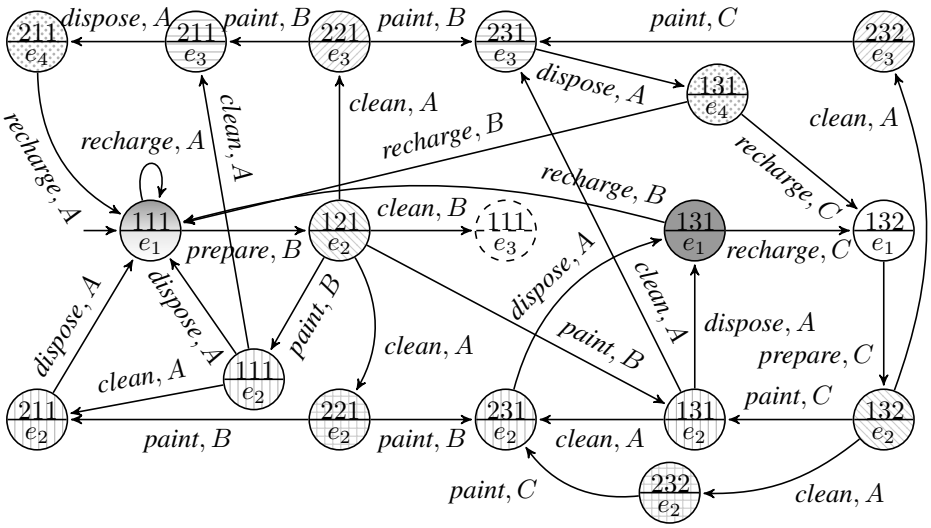
Observe that, clearly, a $\mathcal{T}_T$ state can be simulated by several $\mathcal{T}_\mathcal{S}$'s, as is the case for, e.g., $\langle t_4, e_2\rangle$, which is simulated by both $\langle\langle a_1, b_1, c_1\rangle, e_2\rangle$ and $\langle\langle a_1, b_3, c_1\rangle, e_2\rangle$. Also the converse may happen: $\langle\langle a_1, b_1, c_1\rangle, e_1\rangle$ in $\mathcal{T}_\mathcal{S}$ *ND-simulates* $\mathcal{T}_T$ state $\langle t_1, e_1\rangle$ as well as $\langle t_5, e_1\rangle$. ■

The relevance of the ND-simulation relation to the composition problem addressed here is twofold. Firstly, as will be shown next, computing the largest ND-simulation relation between a target enacted behavior and a system enacted behavior is essentially equivalent to checking whether there exists a composition for the target behavior that "uses" the behaviors available in the system. Secondly, this "simulation-based" approach overcomes the main obstacles that previous solution techniques (e.g., [13]) encountered, as enabling the construction of *flexible* solutions that can take runtime information into account, at no additional (worst-case) cost.

Our first main result states that checking the existence of a composition can be reduced to checking whether the enacted target behavior's initial state is ND-simulated by the enacted system behavior's initial state, which corresponds to checking whether there exists an ND-simulation relation that includes the initial states of both.



(a) Enacted Target Arm $\mathcal{T}_T$



(b) Enacted System Behavior $\mathcal{T}_\mathcal{S}$

Figure 4: The largest ND-simulation relation $\preceq$ between the enacted target behavior $\mathcal{T}_T$ and (a fragment of) the enacted system behavior $\mathcal{T}_\mathcal{S}$. A state in $\mathcal{T}_\mathcal{S}$ ND-simulates those in $\mathcal{T}_T$ that share its pattern, e.g., $\langle\langle a_1, b_3, c_1\rangle, e_2\rangle \preceq \langle t_4, e_2\rangle$. Observe that state $\langle\langle a_1, b_1, c_1\rangle, e_1\rangle$ has two patters—black and white–and hence ND-simulates both $\langle t_1, e_1\rangle$ and $\langle t_5, e_1\rangle$, and that state $\langle\langle a_1, b_1, c_1\rangle, e_3\rangle$ ND-simulates no state.

Theorem 1. *Let $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E}\rangle$ be a system and $\mathcal{B}_T$ a target behavior over $\mathcal{E}$. Moreover, let $\mathcal{T}_T = \langle S_T, \mathcal{A}, s_{T0}, Q_T, \delta_T\rangle$ and $\mathcal{T}_\mathcal{S} = \langle S_\mathcal{S}, \mathcal{A}, \{1, \ldots, n\}, s_{\mathcal{S}0}, Q_\mathcal{S}, \delta_\mathcal{S}\rangle$ be the enacted target behavior and the enacted system behavior for $\mathcal{B}_T$ and $\mathcal{S}$, respectively. Then, a composition controller P of target $\mathcal{B}_T$ on system $\mathcal{S}$ exists if and only if $s_{T0} \preceq s_{\mathcal{S}0}$.*

Proof. (IF PART). First, we define P. To this end, let $h = s^0_\mathcal{S} \xrightarrow{a^1,k^1} \cdots \xrightarrow{a^\ell,k^\ell} s^\ell_\mathcal{S} \in \mathcal{H}$ be a $\mathcal{T}_\mathcal{S}$ history and $a \in \mathcal{A}$ an action. If there exists a $\mathcal{T}_T$ history $h_T = s^0_T \xrightarrow{a^1} \cdots \xrightarrow{a^\ell} s^\ell_T$ (i.e., a history matching the actions of h) such that $s^\ell_T \preceq s^\ell_\mathcal{S}$, and a transition $s^\ell_T \xrightarrow{a} s^{\ell+1}_T$ in $\mathcal{T}_T$ (i.e., action a is $\mathcal{B}_T$-*executable* in s^ℓ_T), then we define $P(h, a) \in \omega_a$, where ω_a is the set of indexes such that for all transitions $s^\ell_T \xrightarrow{a} s^{\ell+1}_T$ and any $k \in \omega_a$:

- there exists a transition $s_{\mathcal{S}}^{\ell} \xrightarrow{a,k_a} s_{\mathcal{S}}^{\ell+1}$ in $\mathcal{T}_{\mathcal{S}}$ with $env(s_{\mathcal{S}}^{\ell+1}) = env(s_T^{\ell+1})$;
- for all transitions $s_{\mathcal{S}}^{\ell} \xrightarrow{a,k_a} s_{\mathcal{S}}^{\ell+1}$ in $\mathcal{T}_{\mathcal{S}}$ with $env(s_{\mathcal{S}}^{\ell+1}) = env(s_T^{\ell+1})$, $s_T^{\ell+1} \preceq s_{\mathcal{S}}^{\ell+1}$.

Because $s_T^{\ell} \preceq s_{\mathcal{S}}^{\ell}$, we know that $\omega_a \neq \emptyset$. In all other cases, namely, when h_T does not exist or a is not $\mathcal{B}_T$-executable, we take $P(h, a) = u$.

Next, we prove that P is indeed a composition, that is, we show that every $\mathcal{T}_T$ trace is realized by P. To this end, we consider any $\mathcal{T}_T$ trace $\tau = s_T^0 \xrightarrow{a^1} s_T^1 \xrightarrow{a^2} \cdots$ ($s_T^0 = s_{T0}$) and prove the following claim first:

> (†) for every $\mathcal{T}_{\mathcal{S}}$ history $h = s_{\mathcal{S}}^0 \xrightarrow{a^1,k^1} \cdots \xrightarrow{a^{\ell},k^{\ell}} s_{\mathcal{S}}^{\ell} \in \mathcal{H}_{\tau,P}$, with $0 \leq \ell < |\tau|$, it is the case that $s_T^{\ell} \preceq s_{\mathcal{S}}^{\ell}$.

Since $\mathcal{H}_{\tau,P} = \bigcup_{\ell \geq 0} \mathcal{H}_{\tau,P}^{\ell}$, we prove (†) by induction on ℓ as follows:

- Let $\mathcal{H}_{\tau,P}^0 = \{s_{\mathcal{S}0}\}$. Clearly, $s_{T0} \preceq s_{\mathcal{S}0}$ and (†) holds trivially.
- Take $h^{\ell+1} = s_{\mathcal{S}}^0 \xrightarrow{a^1,k^1} \cdots \xrightarrow{a^{\ell},k^{\ell}} s_{\mathcal{S}}^{\ell} \xrightarrow{a^{\ell+1},k^{\ell+1}} s_{\mathcal{S}}^{\ell+1} \in \mathcal{H}_{\tau,P}^{\ell+1}$, where $s_{\mathcal{S}}^0 = s_{\mathcal{S}0}$. By the definition of $\mathcal{H}_{\tau,P}^{\ell+1}$, $s_{\mathcal{S}}^{\ell} \xrightarrow{a^{\ell+1},k^{\ell+1}} s_{\mathcal{S}}^{\ell+1}$ in $\mathcal{T}_{\mathcal{S}}$, $env(s_T^{\ell+1}) = env(s_{\mathcal{S}}^{\ell+1})$ and $P(h^{\ell}, a^{\ell+1}) = k^{\ell+1}$. By the induction hypothesis, we know that $s_T^{\ell} \preceq s_{\mathcal{S}}^{\ell}$. Then, by the way P was defined above, it follows that $s_T^{\ell+1} \preceq s_{\mathcal{S}}^{\ell+1}$.

Next, take any $h \in \mathcal{H}_{\tau,P}$ such that $|h| < |\tau|$. Because of the way each $\mathcal{H}_{\tau,P}^{i}$ is constructed, h ought to be of the form $h = s_{\mathcal{S}}^0 \xrightarrow{a^1,k^1} \cdots \xrightarrow{a^{\ell},k^{\ell}} s_{\mathcal{S}}^{\ell}$, that is, h has to match all actions in τ. From (†) above, we have that $s_T^{\ell} \preceq s_{\mathcal{S}}^{\ell}$. Then, by definition of ND-simulation, the fact that $a^{\ell+1}$ is $\mathcal{B}_T$-executable in s_T^{ℓ}, and the way P is defined above, there exists a transition $s_S^{\ell} \xrightarrow{a^{\ell+1},k} s_{\mathcal{S}}^{\ell+1}$, where $k = P(h, a^{\ell+1})$ and $k \in \{1, \ldots, n\}$. In addition, if τ is finite, then for every $h \in \mathcal{H}_{\tau,P}^{|\tau|}$ we have, due to (†) above, that $s_T^{|\tau|} \preceq last(h)$, which in turns implies that if $s_T^{|\tau|} \in Q_T$ (i.e., $\mathcal{T}_T$ is final in enacted state $s_T^{|\tau|}$), then $last(h) \in Q_{\mathcal{S}}$. Then, P realizes τ and P is a composition.

(ONLY-IF PART). Let P be a controller for $\mathcal{S}$ that is a composition of $\mathcal{B}_T$ on $\mathcal{E}$. From P, we build a relation $R \subseteq S_T \times S_{\mathcal{S}}$ that is an ND-simulation such that $\langle s_{T0}, s_{\mathcal{S}0} \rangle \in R$. The definition of R is as follows: $\langle s_T, s_{\mathcal{S}} \rangle \in R$ if and only if there exists a $\mathcal{T}_T$ trace $\tau = s_T^0 \xrightarrow{a^1} s_T^1 \xrightarrow{a^2} \cdots$ and an (induced) $\mathcal{T}_{\mathcal{S}}$ history $h \in \mathcal{H}_{\tau,P}$ such that $s_T = s_T^{|h|}$ and $s_{\mathcal{S}} = last(h)$.

Next, we show that R is an ND-simulation relation (page 12). Consider then a pair $\langle s_T, s_{\mathcal{S}} \rangle \in R$. By R's definition, there exists a $\mathcal{T}_T$ trace of the form $\tau = s_T^0 \xrightarrow{a^1} \cdots \xrightarrow{a^{\ell}} s_T \cdots$ and an ℓ-length $\mathcal{T}_{\mathcal{S}}$ history (induced by τ and P) $h \in \mathcal{H}_{\hat{\tau},P}^{\ell}$ such that $h = s_{\mathcal{S}}^0 \xrightarrow{a^1,k^1} \cdots \xrightarrow{a^{\ell},k^{\ell}} s_{\mathcal{S}}$.

First, due to the way set $\mathcal{H}_{\tau,P}^{\ell}$ is constructed, $env(s_T) = env(s_{\mathcal{S}})$ holds, as only system histories matching the evolution of the environment as in trace τ are considered.

Second, because P is a composition, P realizes τ as well as its $|h|$-length trace prefix $\tau|_{|h|}$. It follows then that if $\mathit{last}(\tau|_{|h|}) = s_T \in Q_T$, then $s_{\mathcal{S}} \in Q_{\mathcal{S}}$.

It remains to prove that the third requirement of ND-simulation holds. To that end, consider an action $a \in \mathcal{A}$ that is $\mathcal{B}_T$-executable in s_T, that is, there exists $s_T \xrightarrow{a} s_T^*$ in $\mathcal{B}_T$. Take now trace $\tau^* = \tau|_\ell \xrightarrow{a} s_T^*$. Clearly, $h \in \mathcal{H}^\ell_{\tau^*,P}$, that is, h can be induced by P when realizing trace τ^*. Since P is a composition, it realizes trace τ^* and hence there exits $k_a \in \{1, \ldots, n\}$ such that $P(h, a) = k_a$ and $s_{\mathcal{S}} \xrightarrow{a,k_a} s_{\mathcal{S}}^*$ in $\mathcal{T}_{\mathcal{S}}$. Next consider any transition $s_T \xrightarrow{a} s'_T$ in $\mathcal{T}_T$. Because the evolution of the environment is independent of that of the behaviors, there must exist $s_{\mathcal{S}} \xrightarrow{a,k_a} s'_{\mathcal{S}}$ with $\mathit{env}(s'_T) = \mathit{env}(s'_{\mathcal{S}})$ and hence condition (3.a) of ND-simulation definition applies. Moreover, $\tau' = \tau|_\ell \xrightarrow{a} s'_T$ is a legal trace of $\mathcal{T}_T$ and history $h \xrightarrow{a,k_a} s'_{\mathcal{S}} \in \mathcal{H}^{\ell+1}_{\tau',P}$. Hence, by definition of R above, $R(s'_T, s'_{\mathcal{S}})$ holds, that is, requirement (3.b) of the ND-simulation definition is satisfied, with $k_a = P(h, a)$ being indeed a witness of $s_T \preceq s_{\mathcal{S}}$ for action a. □

Theorem 1 provides a straightforward method for checking the existence of a composition, namely: *(i)* compute the largest ND-simulation relation between $\mathcal{T}_T$ and $\mathcal{T}_{\mathcal{S}}$, and *(ii)* check whether $\langle s_{T0}, s_{\mathcal{S}0}\rangle$ is in such a relation.

As for computational complexity considerations, observe that algorithm *NDS* described above computes the largest ND-simulation relation $\preceq$ between $\mathcal{T}_T$ and $\mathcal{T}_{\mathcal{S}}$ in polynomial time, with respect to the size of $\mathcal{T}_T$ and $\mathcal{T}_{\mathcal{S}}$. Since the number of states in $\mathcal{T}_{\mathcal{S}}$ is exponential in the number of available behaviors $\mathcal{B}_1, \ldots, \mathcal{B}_n$, we get that the largest ND-simulation relation $\preceq$ can be computed in exponential time in the *number of available behaviors*. Hence, as formally stated in the next theorem, this technique is a notable improvement with respect to the ones based on reduction to PDL [29, 77], which are exponential also in the *number of states* of both the behaviors and the environment.[2] Considering that the composition problem is EXPTIME-hard [59], the upper bound we get is indeed tight, that is, roughly speaking, this is the best we can hope for.

Theorem 2. *Checking for the existence of compositions by computing the largest ND-simulation relation $\preceq$ can be done in polynomial time in the number of states of the available behaviors, of the environment, and of the target behavior, and in exponential time in the number of available behaviors.*

Once computed the ND-simulation relation, the problem of "synthesizing" a controller that is a composition arises. Next, we show how, from the largest ND-simulation relation, we can build a finite state program, i.e., a *controller generator*, that returns, at each step, the set of all available behaviors capable of performing the requested action, while guaranteeing the possibility of delegating to available services all (target-compliant) requests that can be issued in the future.

[2] Though in light of the result in here, a better complexity analysis involving the specific PDL satisfiability procedures could be carried out.

Formally, let $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$ be a system, $\mathcal{B}_T$ a target behavior over $\mathcal{E}$, and $\mathcal{T}_\mathcal{S} = \langle S_\mathcal{S}, \mathcal{A}, \{1, \ldots, n\}, s_{\mathcal{S}0}, Q_\mathcal{S}, \delta_\mathcal{S} \rangle$ and $\mathcal{T}_T = \langle S_T, \mathcal{A}, s_{T0}, Q_T, \delta_T \rangle$ the enacted system behavior and the enacted target behavior corresponding, respectively, to $\mathcal{S}$ and $\mathcal{B}_T$. The *controller generator (CG)* of $\mathcal{S}$ for $\mathcal{B}_T$ is a tuple $CG = \langle \Sigma, \mathcal{A}, \{1, \ldots, n\}, \partial, \omega \rangle$, where:

1. $\Sigma = \{\langle s_T, s_\mathcal{S} \rangle \in S_T \times S_\mathcal{S} \mid s_T \preceq s_\mathcal{S}\}$ is the set of *CG* states, formed by all pairs of $\mathcal{T}_T$ and $\mathcal{T}_\mathcal{S}$ state belonging to the largest ND-simulation relation; given $\sigma = \langle s_T, s_\mathcal{S} \rangle$ we denote s_T by $com_T(\sigma)$ and $s_\mathcal{S}$ by $com_\mathcal{S}(\sigma)$.
2. $\mathcal{A}$ is the finite set of shared actions.
3. $\{1, \ldots, n\}$ is the finite set of available behavior indexes.
4. $\partial \subseteq \Sigma \times \mathcal{A} \times \{1, \ldots, n\} \times \Sigma$ is the *transition relation*, where $\langle \sigma, a, k, \sigma' \rangle \in \partial$, or $\sigma \xrightarrow{a,k} \sigma'$ in *CG*, if and only if:
   - there exists a transition $com_T(\sigma) \xrightarrow{a} com_T(\sigma')$ in $\mathcal{T}_T$;
   - there exists a transition $com_\mathcal{S}(\sigma) \xrightarrow{a,k} com_\mathcal{S}(\sigma')$ in $\mathcal{T}_\mathcal{S}$;
   - for all transitions $com_\mathcal{S}(\sigma) \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ such that $env(s'_\mathcal{S}) = env(com_T(\sigma'))$, $\langle com_T(\sigma'), s'_\mathcal{S} \rangle \in \Sigma$ holds (i.e., k is a witness of $com_T(\sigma) \preceq com_\mathcal{S}(\sigma)$ for action a).
5. $\omega : \Sigma \times A \mapsto 2^{\{1,\ldots,n\}}$ is the *output function* defined as

$$\omega(\sigma, a) = \{k \mid \exists\, \sigma' \in \Sigma \text{ s.t. } \sigma \xrightarrow{a,k} \sigma' \text{ is in } CG\}.$$

Roughly speaking, the *CG* is a finite state transducer that, given an action a (compliant with the target behavior), outputs, through function ω, the set of *all* available behaviors that can perform a next, according to the largest ND-simulation relation $\preceq$. Observe that computing the *CG* from relation $\preceq$ is easy, as it involves checking *local* conditions only. In fact, one could directly compute the *CG* by *enriching* relation $\preceq$, during computation, with information about actions, indices, and transitions.

By Theorem 1, if there exists a composition of $\mathcal{B}_T$, then $s_{T0} \preceq s_{\mathcal{S}0}$ and *CG* does include state $\sigma_0 = \langle s_{T0}, s_{\mathcal{S}0} \rangle$. In such a case, we can build actual controllers, called *generated controllers*, that are compositions of $\mathcal{B}_T$, by picking up, at each step, one available behavior among those returned by output function ω. Notice that full-observability of available behavior states is a crucial assumption here, as both ω and ∂ depend on the current states of both the environment and all system behaviors, which, due to non-determinism, cannot be known with certainty, i.e., can be *reconstructed*, by just looking at the action history. As a result, after each action execution, in order to obtain ω's output, the new states of the system and of the environment need to be known. Of course, more complex scenarios where available behavior states are only partially observable can be considered, though this is out of the scope of this paper.

Formally, controllers that are compositions can be *generated* from the *CG* as follows. [3] Firstly, in analogy with behavior and system traces, we define *CG traces* and

[3] We stress that as a composition exists if and only if $\sigma_0 = \langle s_{T0}, s_{\mathcal{S}0} \rangle \in \Sigma$ (Theorem 1), constructing a composition makes sense only if this condition holds.

histories: a *trace for CG* is a possibly infinite sequence $\sigma^0 \xrightarrow{a^1,k^1} \sigma^1 \xrightarrow{a^2,k^2} \cdots$, such that each transition $\sigma^j \xrightarrow{a^{j+1},k^{j+1}} \sigma^{j+1}$ is in *CG*, for all $j \geq 0$ [4]; consequently, a *history for CG* is a finite prefix of a trace. Functions *last* and $|\cdot|$ over *CG* histories are defined as usual. For technical convenience, given a *CG* trace $\tau_{CG} = \sigma^0 \xrightarrow{a^1,k^1} \sigma^1 \xrightarrow{a^2,k^2} \cdots$, we define the corresponding *projected system trace* as the sequence $proj_{\mathcal{S}}(\tau_{CG}) = com_{\mathcal{S}}(\sigma^0) \xrightarrow{a^1,k^1} com_{\mathcal{S}}(\sigma^1) \xrightarrow{a^2,k^2} \cdots$, intuitively obtained from τ_{CG} by taking only the system component of each state. Clearly, by definition of *CG*, if $\sigma^0 = \langle s_{T0}, s_{\mathcal{S}0}\rangle$ then $proj_{\mathcal{S}}(\tau_{CG})$ is a legal $\mathcal{T}_{\mathcal{S}}$ trace. Also, from τ_{CG} we define the corresponding *projected target trace* $proj_T(\tau_{CG}) = com_T(\sigma^0) \xrightarrow{a^1} com_T(\sigma^1) \xrightarrow{a^2} \cdots$ that can be easily proven to be a legal $\mathcal{T}_T$ trace if $\sigma^0 = \langle s_{T0}, s_{\mathcal{S}0}\rangle$. Similarly, we can derive from each *CG* history h_{CG} a *projected system history* and a *projected target history*, which are, respectively, a $\mathcal{T}_T$ history and a $\mathcal{T}_{\mathcal{S}}$ history, if $\sigma^0 = \langle s_{T0}, s_{\mathcal{S}0}\rangle$.

Next, let $\mathcal{H}_{CG}$ be the set of all *CG* histories, and consider a selection function

$$CGP_{\text{CHOOSE}} : \mathcal{H}_{CG} \times \mathcal{A} \mapsto \{1, \ldots, n\}$$

such that for all $h_{CG} \in \mathcal{H}_{CG}$, $CGP_{\text{CHOOSE}}(h_{CG}, a) = \text{CHOOSE}(\omega(last(h_{CG}), a))$, where CHOOSE stands for a *choice* function that picks a random (index) element among those returned by $\omega(last(h_{CG}), a)$, if non empty, while is left unconstrained, if $\omega(last(h_{CG}), a)$ is empty or h_{CG} is not a legal history for *CG*. Observe that, being a choice function, different applications of CHOOSE to the same parameters yield different results, each corresponding to a different way of resolving the non-determinism in the element choice.

Finally, assuming that *CG* includes $\sigma_0 = \langle s_{T0}, s_{\mathcal{S}0}\rangle$, for each *CG* history $h_{CG} = \sigma^0 \xrightarrow{a^1,k^1} \cdots \xrightarrow{a^\ell,k^\ell} \sigma^\ell$ such that $\sigma^0 = \sigma_0$, consider the corresponding projected system history:

$$proj_{\mathcal{S}}(h_{CG}) = com_{\mathcal{S}}(\sigma^0) \xrightarrow{a^1,k^1} \cdots \xrightarrow{a^\ell,k^\ell} com_{\mathcal{S}}(\sigma^\ell).$$

For a given selection function CGP_{CHOOSE}, a *generated controller* is any function $P_{\text{CHOOSE}} : \mathcal{H} \times \mathcal{A} \mapsto \{1, \ldots, n\}$ such that for every $\mathcal{T}_{\mathcal{S}}$ history $h \in \mathcal{H}$ and action $a \in \mathcal{A}$, if $h = proj_{\mathcal{S}}(h_{CG})$ for some *CG* history h_{CG}, then $P_{\text{CHOOSE}}(h, a) = CGP_{\text{CHOOSE}}(h_{CG}, a)$.

The following results relate *CG*s to compositions and show that, given a *CG* containing σ_0, one gets *all and only* controllers that are compositions by considering all possible resolutions of the non-determinism of function CHOOSE. Notably, while each specific composition may be an infinite state program, the controller generator, which in fact includes them all, is always finite.

Theorem 3. *Let $\mathcal{S}$, $\mathcal{B}_T$, $\mathcal{T}_{\mathcal{S}}$ and $\mathcal{T}_T$ be as above, and let $CG = \langle \Sigma, \mathcal{A}, \{1, \ldots, n\}, \partial, \omega\rangle$ be the controller generator of $\mathcal{S}$ for $\mathcal{B}_T$. If $\sigma_0 = \langle s_{T0}, s_{\mathcal{S}0}\rangle \in \Sigma$, then:*

1. *every generated controller obtained from CG as shown above is a composition of $\mathcal{B}_T$ on $\mathcal{E}$;*

[4] Observe that we do not require $\sigma^0 = \langle s_{T0}, s_{\mathcal{S}0}\rangle \doteq \sigma_0$ as, in general, the *CG* does not include σ_0.

2. *every controller that is a composition of $\mathcal{B}_T$ on $\mathcal{E}$ can be obtained from CG as shown above.*

Proof. To prove 1., we show that for every target trace $\tau \in \mathcal{H}_T$ and controller P_{CHOOSE} defined as above, there exists a controller P, defined as in the (IF PART) of Theorem 1, such that $\mathcal{H}_{\tau,P} = \mathcal{H}_{\tau,P_{\text{CHOOSE}}}$. Since P is proven to realize τ, by looking at the definition of *trace realization*, this is enough to prove that P_{CHOOSE} realizes τ as well.

Let $\mathcal{H}^{\ell}_{\tau,P_{\text{CHOOSE}}} \subseteq \mathcal{H}_{\tau,P_{\text{CHOOSE}}}$ be the set of system histories $h = s^0_{\mathcal{S}} \xrightarrow{a^1,k^1} \cdots \xrightarrow{a^\ell,k^\ell} s^\ell_{\mathcal{S}}$ induced by τ and P_{CHOOSE}. Also, let $\mathcal{H}^{\ell}_{\tau,P} \subseteq \mathcal{H}_{\tau,P}$ be the analogous set for τ and P. We prove, by induction, the existence of P (defined as in the (IF PART) of Theorem 1) such that $\mathcal{H}^{\ell}_{\tau,P_{\text{CHOOSE}}} = \mathcal{H}^{\ell}_{\tau,P}$, for every $\ell \geq 0$. Since, for every controller C, $\mathcal{H}_{\tau,C} = \bigcup_{\ell \geq 0} \mathcal{H}^{\ell}_{\tau,C}$, we get that $\mathcal{H}_{\tau,P_{\text{CHOOSE}}} = \mathcal{H}_{\tau,P}$.
For the base case ($\ell = 0$), no matter how P is defined, $\mathcal{H}^0_{\tau,P_{\text{CHOOSE}}} = \mathcal{H}^0_{\tau,P} = \{s_{\mathcal{S}0}\}$.
By induction hypothesis, assume $\mathcal{H}^{\ell}_{\tau,P_{\text{CHOOSE}}} = \mathcal{H}^{\ell}_{\tau,P}$, and consider a system history $h = s^0_{\mathcal{S}} \xrightarrow{a^1,k^1} \cdots \xrightarrow{a^\ell,k^\ell} s^\ell_{\mathcal{S}} \in \mathcal{H}^{\ell}_{\tau,P_{\text{CHOOSE}}}$. Because h is an induced history, for $i = 1, \ldots, \ell$, we have $k^i = P_{\text{CHOOSE}}(h|_{i-1}, a^i)$, where $h|_j = s^0_{\mathcal{S}} \xrightarrow{a^1,k^1} \cdots \xrightarrow{a^j,k^j} s^j_{\mathcal{S}}$. In particular $k^\ell = P_{\text{CHOOSE}}(h|_{\ell-1}, a^\ell)$ is defined, and therefore, by P_{CHOOSE} definition, there exists a *CG* history $\tilde{h}_{CG} = \langle \tilde{s}^0_T, s^0_{\mathcal{S}} \rangle \xrightarrow{a^1,k^1} \cdots \xrightarrow{a^{\ell-1},k^{\ell-1}} \langle \tilde{s}^{\ell-1}_T, s^{\ell-1}_{\mathcal{S}} \rangle$ such that $proj_{\mathcal{S}}(\tilde{h}_{CG}) = h|_{\ell-1}$. In principle, $\tilde{h}_{CG}$ can be any, as long as $proj_{\mathcal{S}}(\tilde{h}_{CG}) = h|_{\ell-1}$. In particular, it could be unrelated to τ. But because $\tilde{h}_{CG}$ is a history for *CG*, it is such that $\langle \tilde{s}^i_T, s^i_{\mathcal{S}} \rangle \in \Sigma$ $(i = 0, \ldots, \ell-1)$, hence $\tilde{s}^i_T \preceq s^i_{\mathcal{S}}$ and therefore $env(\tilde{s}^i_T) = env(s^i_{\mathcal{S}})$. In turn, h being induced by τ, $env(s^i_{\mathcal{S}}) = env(s^i_T)$, and hence $env(\tilde{s}^i_T) = env(s^i_{\mathcal{S}}) = env(s^i_T)$ $(i = 0, \ldots, \ell-1)$. Finally, $\mathcal{B}_T$ being deterministic and having $\tilde{s}^0_T = s^0_T = s_{T0}$, we get $beh(\tilde{s}^i_T) = beh(s^i_T)$. So, we conclude that $proj_T(\tilde{h}_{CG}) = \tau|_{\ell-1}$. Based on this and the fact that $proj_S(\tilde{h}_{CG}) = h|_{\ell-1}$, we also have that $\tilde{h}_{CG}$ is unique, for fixed h.

By definition of induced history, given h, $k^\ell = P_{\text{CHOOSE}}(h|_{\ell-1}, a^\ell) = CGP_{\text{CHOOSE}}(\tilde{h}_{CG}, a^\ell) \in \omega(\langle s^{\ell-1}_T, s^{\ell-1}_{\mathcal{S}} \rangle, a^\ell)$. So, observing the definition of *CG* and ω, it is easily seen that the sequence $h_{CG} = \langle s^0_T, s^0_{\mathcal{S}} \rangle \xrightarrow{a^1,k^1} \cdots \xrightarrow{a^\ell,k^\ell} \langle s^\ell_T, s^\ell_{\mathcal{S}} \rangle$ is a CG history, in particular, such that $proj_{\mathcal{S}}(h_{CG}) = h$ and $proj_T(h_{CG}) = \tau|_\ell$.

Next, we prove that all possible extensions of h, obtained by *realizing* action $a^{\ell+1}$ in τ (if any), according to P_{CHOOSE}, are also possible under P, and viceversa. In other words, we prove that $\mathcal{H}^{\ell+1}_{\tau,P_{\text{CHOOSE}}} = \mathcal{H}^{\ell+1}_{\tau,P}$. Two cases are possible: either *(i)* $\tau = \tau|_\ell$ (i.e., τ is finite); or *(ii)* not. In case *(i)*, we trivially obtain $\mathcal{H}^{\ell+1}_{\tau,P_{\text{CHOOSE}}} = \mathcal{H}^{\ell+1}_{\tau,P} = \emptyset$. For case *(ii)*, observe that $s^\ell_T \preceq s^\ell_{\mathcal{S}}$ –as $\langle s^\ell_T, s^\ell_{\mathcal{S}} \rangle \in \Sigma$– and that $a^{\ell+1}$ is $\mathcal{B}_T$-executable in s^ℓ_T (this trivially comes from $a^{\ell+1}$ position in τ). In addition, h_{CG} is proved, above, a *CG* history such that $proj_{\mathcal{S}}(h_{CG}) = h$. Therefore, by definition of generated controller, $P_{\text{CHOOSE}}(h, a^{\ell+1}) = k^{\ell+1} \in \omega(\langle s^\ell_T, s^\ell_{\mathcal{S}} \rangle, a^{\ell+1}) \neq \emptyset$.

On the other hand, consider the construction of P in the (IF PART) of Theorem 1. Given h, $\tau|_\ell$ matches (by construction) all actions in h, and is such that $s^\ell_T = last(\tau|_\ell) \preceq last(h) = s^\ell_{\mathcal{S}}$ (as proven above). So, $P(h, a^{\ell+1}) \in \omega_{a^{\ell+1}} \neq \emptyset$. But then, observing that $\omega(\langle s^\ell_T, s^\ell_{\mathcal{S}} \rangle, a^{\ell+1}) = \omega_{a^{\ell+1}}$, no matter which index P_{CHOOSE} returns, P can choose the same index, say $k^{\ell+1}$, from $\omega_{a^{\ell+1}}$ so that $k^{\ell+1} = P_{\text{CHOOSE}}(h, a^{\ell+1}) = P(h, a^{\ell+1})$. Clearly, given h, $a^{\ell+1}$ and $k^{\ell+1}$, every possible system history of the form

$\hat{h} = h \xrightarrow{a^{\ell+1},k^{\ell+1}} s^{\ell+1}$ is such that $\hat{h} \in \mathcal{H}^{\ell+1}_{\tau,P}$ if and only if $\hat{h} \in \mathcal{H}^{\ell+1}_{\tau,P_{\text{CHOOSE}}}$. Since h is arbitrarily chosen, we ultimately get $\mathcal{H}^{\ell+1}_{\tau,P} = \mathcal{H}^{\ell+1}_{\tau,P_{\text{CHOOSE}}}$.

We prove 2. by showing that for all $\mathcal{T}_T$ traces, all decisions made by P along an arbitrary history induced by τ and P are compliant with the definition of generated controller.

Let $\tau = s^0_T \xrightarrow{a^1} s^1_T \xrightarrow{a^2} \cdots$ be a $\mathcal{T}_T$ trace, and $h = s^0_{\mathcal{S}} \xrightarrow{a^1,k^1} \cdots \xrightarrow{a^\ell,k^\ell} s^\ell_{\mathcal{S}} \in \mathcal{H}_{\tau,P}$ a generic history induced by τ and P. Since P is a composition, by (ONLY-IF PART) of Theorem 1, we get that $k^i = P(h|_i, a^{i+1})$ is a witness of $s^i_T \preceq s^i_{\mathcal{S}}$ for a^{i+1}, for $i = 0, \ldots, \ell-1$. Then, $h_{CG} = \langle s^0_T, s^0_{\mathcal{S}}\rangle \xrightarrow{a^1,k^1} \cdots \xrightarrow{a^\ell,k^\ell} \langle s^\ell_T, s^\ell_{\mathcal{S}}\rangle$ is a *CG*-history. Indeed, by definition of ω, for every a that is $\mathcal{B}_T$-executable in s_T, $\omega(\langle s_T, s_{\mathcal{S}}\rangle, a)$ contains all (and only) the witnesses of $s_T \preceq s_{\mathcal{S}}$ for a. So, this will be true, in particular, for $\omega(\langle s^i_T, s^i_{\mathcal{S}}\rangle, a^{i+1})$.

Since every prefix of a history is a history itself, and ℓ being arbitrary, the argument above proves that for every prefix of h, say $h|_j$ ($j = 0, \ldots, \ell-1$), there exists an h_{CG} prefix $h_{CG}|_j$, which is a *CG*-history, such that $proj_{\mathcal{S}}(h_{CG}|_j) = h|_j$. But then, $P_{\text{CHOOSE}}(h|_j, a^{j+1}) = \textit{CGP}_{\text{CHOOSE}}(h_{CG}|_j) \in \omega(\langle s^j_T, s^j_{\mathcal{S}}\rangle, a^{j+1})$. As $k^j \in \omega(\langle s^j_T, s^j_{\mathcal{S}}\rangle, a^{j+1})$, we get that P_{CHOOSE} can behave in the same way as P, along h, by properly picking, at every step, an element from the set returned by ω. Since ℓ was arbitrarily chosen, this result extends to all histories $h \in \mathcal{H}_{\tau,P}$. □

We close this Section by observing that compositions can be generated *just-in-time*, based on both the *CG* and observability of behavior and environment states. Intuitively, the *CG* is analogous to a sort of "meta-plan" or a stateful non-deterministic "complete universal plan", which keeps all the existing plans at its disposal and selects the one to follow for next action, possibly with contingent decisions.

Example 5. The *CG* can decide how to delegate actions, as requests from target arm $\mathcal{B}_T$ come in. For instance, if a *clean* action is requested after a block has been prepared, the *CG* knows it ought to delegate such a request to arm $\mathcal{B}_A$, so as to stay within the ND-simulation relation. While physically possible, delegating *clean* to arm $\mathcal{B}_B$ would bring the enacted system into state $\langle\langle a_1, b_1, c_1\rangle, e_3\rangle$ which is known not to be in ND-simulation with the (enacted) target. ■

4. On Behavior Failures

In discussing the behavior composition problem, we have, so far, assumed implicitly that all (available) component modules are fully reliable, i.e., that they are always available, and behave "correctly", relative to their specification. However, there are many situations and domains in which full reliability of components might be not an adequate assumption. For example, in multi-agent complex and highly dynamic domains, one cannot rely neither on total availability nor on reliability of the existing modules, which may stop being available due to a variety of reasons, e.g.: devices may break down, agents may decide to stop cooperating, communication with agents may

drop, exogenous events may change the state of the environment, and many other; also, behaviors may possibly re-appear into the system at some later stage, thus creating new "composition opportunities" for the controller.

Generally speaking, behavior and environment specifications can be seen as contracts, and failures, such as those described above, can be interpreted as "breaches" of such contracts. In this Section, we identify some classes of failures and propose respective procedures to "repair" the controller under execution when the failure occurred. Specifically, we identify five core ways of breaking contracts:[5]

(a) A behavior *temporarily freezes*, that is, it stops responding and remains still, then eventually resumes in the same state it was in. As a result, while frozen, the controller cannot delegate actions to it.
(b) A behavior unexpectedly and arbitrarily (i.e., without respecting its transition relation) *changes its current state*. The controller can in principle keep delegating actions to it, but it must take into account the behavior's new state.
(c) The *environment* unexpectedly and arbitrarily (i.e., without respecting its transition relation) changes its current state. The controller has to take into account that this affects both the target and the available behaviors.
(d) A behavior dies, that is, it becomes *permanently unavailable*. The controller has to completely stop delegating actions to it.
(e) A behavior that was assumed dead unexpectedly *resumes operation* starting in a certain state. The controller can exploit this opportunity by delegating actions to the resumed behavior, again.

Previous composition techniques (e.g., [14, 29, 77]) do not address these cases, as they assume that controllers always deal with fully reliable modules. Consequently, upon any of the above failures, we are only left with the (default) option of "re-planning" from scratch for a whole new controller, if any. What we shall prove in this Section is that the simulation-based technique presented in Section 3 is *intrinsically robust*, in the sense of being able to deal with unexpected failures by suitably *refining* the solution at hand, either *on-the-fly* (for cases (a), (b), and (c)), or *parsimoniously* (for cases (d) and (e)), thus avoiding full re-planning.

4.1. Reactive Adaptability

We start by showing that Theorem 3 provides us with a sound and complete technique for dealing with failure cases (a), (b), and (c), without requiring any re-planning step. As a matter of fact, once we have the controller generator, actual compositions can be generated "*just-in-time*", as (target compliant) actions are requested. What is particularly interesting about generated controllers is that one can *delay the choice* performed by CHOOSE until run-time, when contingent information, such as actual behavior availability, can be taken into account. This ability provides controllers with great flexibility, as, in a sense, they can "switch" compositions online, as needed. A controller that delay delegation in this way is referred to as *just-in-time generated controllers*, and denoted as CGP_{JIT}. Below, we discuss the effectiveness of CGP_{JIT}s in cases (a), (b), and (c).

[5] Obviously, we assume an infrastructure that is able to distinguish between these failures.

Freezing of behaviors. A CGP_{JIT} fully addresses temporary behavior freezings, i.e., failure case (a). Indeed, if a behavior is temporarily frozen, the CGP_{JIT} simply stops delegating actions to it, and continues with any other possible choice. [6] Obviously, if no other choices are possible, the CGP_{JIT} is left with no other option than waiting for the frozen behavior to come back.

State change of behaviors and environment. CGP_{JIT}s also address unexpected changes in the internal state of behaviors and/or of the environment, that is, failure cases (b) and (c).[7] To understand this, let us denote by $\mathcal{T}_{\mathcal{S}}(z_{\mathcal{S}})$ the variant of the enacted system behavior whose initial state is $z_{\mathcal{S}}$ instead of $s_{\mathcal{S}0}$. Similarly, let us denote by $\mathcal{T}_T(z_T)$ the enacted target behavior whose initial state is z_T instead of s_{T0}. Next, suppose that the state of the enacted system behavior changes, unexpectedly, to state $\hat{s}_{\mathcal{S}}$, due to a change of the state of a behavior (or a set of behaviors) and/or of the environment. Then, if s_T is the state of the target when the failure happened, one should recompute the composition with the system starting in $\hat{s}_{\mathcal{S}}$ and the target starting from $\hat{s}_T$, where $\hat{s}_T$ is just s_T with its environment state replaced by the one in $\hat{s}_{\mathcal{S}}$ (note $\hat{s}_T = s_T$ for failures of type (b)). Observe, though, that ND-simulation relations are *independent* from the initial states of both the target and the system enacted behaviors. Therefore, the largest ND-simulation relation between $\mathcal{T}_T(\hat{s}_T)$ and $\mathcal{T}_{\mathcal{S}}(\hat{s}_{\mathcal{S}})$ is, in fact, relation $\preceq$, that we already computed. This implies that we can still use the very same controller generator *CG* (and the same just-in-time generated controller CGP_{JIT} as well), with the guarantee that all compositions of the system variant for the target variant, if any, are still captured by the *CG* (and CGP_{JIT} too). Put it all together, we only need to check whether $\hat{s}_T \preceq \hat{s}_{\mathcal{S}}$, and, if so, continue to use CGP_{JIT} (now from 0-length *CG* history $h^0_{CG} = \langle \hat{s}_T, \hat{s}_{\mathcal{S}} \rangle$).

Example 6. Upon an unexpected change in the system, in the environment or any available behavior, the *CG* can react/adapt to the change immediately. For instance, referring to Figure 4, suppose the target is in state t_3, the environment in state e_3, and the available behaviors $\mathcal{B}_1$, $\mathcal{B}_2$, and $\mathcal{B}_3$, are in their states a_2, b_2, and c_2, respectively. That is, $\mathcal{T}_T$ is in $\langle t_3, e_3 \rangle$, and $\mathcal{T}_{\mathcal{S}}$ is in $\langle \langle a_2, b_2, c_1 \rangle, e_3 \rangle$. Suppose that, unexpectedly, the environment happens to change to state e_2—someone has recharged the water tank. All that is needed in this case is to check whether the new states of $\mathcal{T}_T$ and $\mathcal{T}_{\mathcal{S}}$, namely $\langle t_3, e_2 \rangle$ and $\langle \langle a_2, b_2, c_1 \rangle, e_2 \rangle$, respectively, are still related according to relation $\preceq$. Since they are, the *CG* continues the realization of the target from such (new) enacted states. ■

Computing reactive compositions on-the-fly. Observe that CGP_{JIT}, that is CGP_{CHOOSE} where CHOOSE is resolved at run-time (and *CG* for the matter), can be computed *on-the-fly* by storing only the ND-simulation relation $\preceq$. In fact, at each point, the only information required for the next choice is $\omega(\sigma, a)$, where $\sigma \in \Sigma$ (recall $\Sigma = \preceq$) is formed by the current state of the enacted target behavior and that of the enacted system behavior. Now, in order to compute $\omega(\sigma, a)$ we only need to know $\preceq$.

[6] If more information is at hand, the CGP_{JIT} may use it to choose in an informed way, though this is out of the scope of this paper.

[7] Although hardly as meaningful as the ones above, unforeseen changes in the target state can be accounted for in a similar way.

Algorithm 2: *NDSP*$(\mathcal{T}_T, \mathcal{T}_\mathcal{S}, \mathcal{R}_{init}, \mathcal{R}_{sure})$

$\mathcal{R} := \mathcal{R}_{init} \setminus \mathcal{R}_{sure}$;
$\mathcal{R} := \mathcal{R} \setminus \{\langle s_T, s_\mathcal{S}\rangle \mid (env(s_T) \neq env(s_\mathcal{S})) \vee (s_T \in Q_T \wedge s_\mathcal{S} \notin Q_\mathcal{S})\}$;
repeat
  $\mathcal{R} := (\mathcal{R} \setminus \mathcal{C})$, where $\mathcal{C}$ is the set of $\langle s_T, s_\mathcal{S}\rangle \in \mathcal{R}$ such that there exists an action $a \in \mathcal{A}$ such that for each k there is a transition $s_T \xrightarrow{a} s'_T$ in $\mathcal{T}_T$ such that either:
    (a) there is no transition $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ such that $env(s'_T) = env(s'_\mathcal{S})$; or
    (b) there exists a transition $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ such that $env(s'_T) = env(s'_\mathcal{S})$ but $\langle s'_T, s'_\mathcal{S}\rangle \notin \mathcal{R} \cup \mathcal{R}_{sure}$.
until $(\mathcal{C} = \emptyset)$;
return $\mathcal{R} \cup \mathcal{R}_{sure}$;

4.2. Parsimonious Refinement

As seen above, failure cases (a), (b), and (c), do not need any particular effort to be dealt with. However, when considering cases (d) and (e), things change significantly: a simple reactive approach is no longer sufficient, and more complex refinement techniques are required. In this Section, we show how a composition can be refined, in an intelligent manner, so as to cope with the latter cases. We start by defining a new algorithm, namely Algorithm 2 (*NDSP*), and some related results that will be useful later on.

Algorithm 2 is a (generalized) parametric version of Algorithm 1 that computes the largest ND-simulation relation contained in a set $\mathcal{R}_{init}$, between $\mathcal{T}_T$ and $\mathcal{T}_\mathcal{S}$. To this end, it takes two extra input parameters, namely: $\mathcal{R}_{init} \subseteq S_T \times S_\mathcal{S}$, i.e., a starting relation from which the largest ND-simulation is to be computed; and $\mathcal{R}_{sure} \subseteq S_T \times S_\mathcal{S}$, i.e., a relation containing tuples already known to be in the ND-simulation relation. Intuitively, the algorithm works the same way as *NDS*, except that: *(i)* instead of starting from $S_T \times S_\mathcal{S}$, it takes the initial set $\mathcal{R}_{init}$ as input and, in addition, *(ii)* neglects all pairs contained in $\mathcal{R}_{sure}$, as they are assumed to be (*surely*) included in the ND-simulation relation. Clearly, for $\mathcal{R}_{init} = S_T \times S_\mathcal{S}$ and $\mathcal{R}_{sure} = \emptyset$, we expect *NDSP* to behave in the same way as *NDS*. Indeed, this is a special case of next result, which identifies sufficient conditions on the new parameters to guarantee that the outputs of *NDSP* and *NDS* match.

Lemma 4. *Consider a system $\mathcal{S} = \{\mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E}\}$ and a target behavior $\mathcal{B}_T$, and let $\mathcal{T}_\mathcal{S}$ and $\mathcal{T}_T$ be their respective enacted behaviors. If $\mathcal{R}_{sure} \subseteq NDS(\mathcal{T}_T, \mathcal{T}_\mathcal{S}) \subseteq \mathcal{R}_{init}$, then $NDSP(\mathcal{T}_T, \mathcal{T}_\mathcal{S}, \mathcal{R}_{init}, \mathcal{R}_{sure}) = NDS(\mathcal{T}_T, \mathcal{T}_\mathcal{S})$.*

Proof. Let $\mathcal{R}_1^i$ and $\mathcal{R}_2^i$ be the sets representing $\mathcal{R}$ in algorithms *NDS* and *NDSP*, respectively, after i repeat-loop iteration. Similarly, define $\mathcal{C}_1^i$ and $\mathcal{C}_2^i$. Moreover, assume that *NDSP* and *NDS* require n_2 and n_1, respectively, repeat-loop iterations (clearly, $n_2 \leq n_1$).

First, let us prove, by induction on i, that $\mathcal{R}_2^i \cup \mathcal{R}_{sure} \subseteq \mathcal{R}_1^i$. It is obvious that $\mathcal{R}_2^0 \cup \mathcal{R}_{sure} \subseteq \mathcal{R}_1^0$ (observe $\mathcal{R}_{init} \subseteq S_T \times S_\mathcal{S}$). Suppose now that $\mathcal{R}_2^r \cup \mathcal{R}_{sure} \subseteq \mathcal{R}_1^r$, with $r < n_1$. Let $\pi = \langle s_T, s_\mathcal{S} \rangle \in \mathcal{R}_2^{r+1} \cup \mathcal{R}_{sure}$, but $\pi \notin \mathcal{R}_1^{r+1}$. Since neither $\mathcal{R}_1^i$ nor $\mathcal{R}_2^i$ are ever expanded along iterations, it is the case that $\pi \notin \mathcal{R}_{sure}$ (otherwise $\pi \in \mathit{NDS}(\mathcal{T}_T, \mathcal{T}_\mathcal{S})$ and $\pi \in \mathcal{R}_1^{r+1}$, as $\mathit{NDS}(\mathcal{T}_T, \mathcal{T}_\mathcal{S}) \subseteq \mathcal{R}_1^{r+1}$), and thus $\pi \in \mathcal{R}_2^{r+1}$, $\pi \in \mathcal{R}_2^r$, and $\pi \in \mathcal{R}_1^r$. Because $\pi \notin \mathcal{R}_1^{r+1}$, π was deleted at the r-th loop iteration of *NDS*, that is, $\pi \in \mathcal{C}_1^r$. This means that there exists an action $\hat{a} \in \mathcal{A}$ such that for each k there is a transition $s_T \xrightarrow{\hat{a}} s'_T$ in $\mathcal{T}_T$ such that either *(a)* or *(b)* of step 3 of *NDS* holds. If case *(a)* holds, then also $\pi \in \mathcal{C}_2^r$ trivially holds. If case *(b)* applies for some k, then there exists a tuple $\pi'_k = \langle s'_T, s'_\mathcal{S} \rangle$ such that $s_\mathcal{S} \xrightarrow{\hat{a},k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$, but $\pi'_k \notin \mathcal{R}_1^r$. By induction hypothesis, $\pi'_k \notin \mathcal{R}_2^r \cup \mathcal{R}_{sure}$. Thus, action $\hat{a}$ and tuple π'_k do indeed satisfy the requirement of the step 4 of *NDSP*. Hence, $\pi \in \mathcal{C}_2^r$. We conclude that if either *(a)* or *(b)* of *NDS*' step 3 hold then $\pi \in \mathcal{C}_2^r$ and, consequently, $\pi \notin \mathcal{R}_2^{r+1}$. Contradiction. Therefore, $\mathcal{R}_2^{r+1} \cup \mathcal{R}_{sure} \subseteq \mathcal{R}_1^{r+1}$ and $\mathit{NDSP}(\mathcal{T}_t, \mathcal{T}_\mathcal{S}, \mathcal{R}_{init}, \mathcal{R}_{sure}) \subseteq \mathit{NDS}(\mathcal{T}_t, \mathcal{T}_\mathcal{S})$.

Next, we prove that $\mathit{NDS}(\mathcal{T}_T, \mathcal{T}_\mathcal{S}) \subseteq \mathit{NDSP}(\mathcal{T}_T, \mathcal{T}_\mathcal{S}, \mathcal{R}_{init}, \mathcal{R}_{sure})$. To that end, we shall prove, by induction on i, that $\mathit{NDS}(\mathcal{T}_T, \mathcal{T}_\mathcal{S}) \subseteq \mathcal{R}_2^i \cup \mathcal{R}_{sure}$. Since $\mathit{NDS}(\mathcal{T}_T, \mathcal{T}_\mathcal{S}) \subseteq \mathcal{R}_{init}$, $\mathit{NDS}(\mathcal{T}_T, \mathcal{T}_\mathcal{S}) \subseteq \mathcal{R}_2^0 \cup \mathcal{R}_{sure}$. Next, suppose that $\mathit{NDS}(\mathcal{T}_T, \mathcal{T}_\mathcal{S}) \subseteq \mathcal{R}_2^r \cup \mathcal{R}_{sure}$, for some $r < n_2$, and let $\pi = \langle s_T, s_\mathcal{S} \rangle \in \mathit{NDS}(\mathcal{T}_T, \mathcal{T}_\mathcal{S})$ but $\pi \notin \mathcal{R}_2^{r+1} \cup \mathcal{R}_{sure}$. By induction hypothesis, $\pi \in \mathcal{R}_2^r \cup \mathcal{R}_{sure}$, and π was therefore removed from $\mathcal{R}_2$ in the r-th iteration of the *NDSP* algorithm. This means that there exists an action $\hat{a} \in \mathcal{A}$ such that for each k there is a transition $s_T \xrightarrow{\hat{a}} s'_T$ in $\mathcal{T}_\mathcal{S}$ such that either *(a)* or *(b)* of the fourth step in *NDSP* holds. In particular, if case *(b)* applies for some k, then there exists a tuple $\pi'_k = \langle s'_T, s'_\mathcal{S} \rangle$ such that $s_\mathcal{S} \xrightarrow{\hat{a},k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$, but $\pi'_k \notin R_2^r \cup \mathcal{R}_{sure}$. By the induction hypothesis, $\pi'_k \notin \mathit{NDS}(\mathcal{T}_T, \mathcal{T}_\mathcal{S})$ and thus $\pi'_k \notin \mathcal{R}_1^{n_1}$. However, since $\pi \in \mathit{NDS}(\mathcal{T}_T, \mathcal{T}_\mathcal{S})$, $\pi \in \mathcal{R}_1^{n_1}$. But, by using the same action $\hat{a}$, together with the corresponding $\pi'_k \notin \mathcal{R}_1^{n_1}$ tuples, π is a candidate to be removed from set $\mathcal{R}_1^{n_1}$, i.e., $\pi \in \mathcal{C}_1^{n_1}$. Then, algorithm *NDS* requires more than n_1, a contradiction. Hence, $\pi \in \mathcal{R}_2^r \cup \mathcal{R}_{sure}$ and $\mathit{NDS}(\mathcal{T}_T, \mathcal{T}_\mathcal{S}) \subseteq \mathit{NDSP}(\mathcal{T}_t, \mathcal{T}_\mathcal{S}, \mathcal{R}_{init}, \mathcal{R}_{sure})$ follows. □

Next, we introduce convenient notations to shrink and expand systems and ND-simulation relations. Given a system $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$ and a set of behavior indexes $W \subseteq \{1, \ldots, n\}$, we denote by $\mathcal{S}(W)$ the system derived from $\mathcal{S}$ by considering only (i.e., projecting on) all behaviors $\mathcal{B}_i$ such that $i \in W$ (note $\mathcal{S} = \mathcal{S}(\{1, \ldots, n\})$). Also, for an enacted target behavior $\mathcal{T}_T$ over $\mathcal{E}$, we denote by $\preceq_W$ the *largest* ND-simulation relation of $\mathcal{T}_T$ by $\mathcal{T}_{\mathcal{S}(W)}$. Finally, given a further set of indexes $U \subseteq \{1, \ldots, n\}$ such that $W \cap U = \emptyset$, we denote by $\preceq_W \otimes U$ the relation obtained from $\preceq_W$, by (trivially) putting back into the system all $\mathcal{B}_i$ such that $i \in U$. Formally, this latter operation can be define as follows (without loss of generality, assume $W = \{1, \ldots, \ell\}$ and $U = \{\ell + 1, \ldots, m\}$):

$$
\begin{array}{l}
\preceq_W \otimes U = \\
\quad \{ \langle s_T, s' \rangle \mid s' = \langle b_1, \ldots, b_\ell, b_{\ell+1}, \ldots, b_m, e \rangle \\
\qquad\qquad \text{such that } \langle s_T, \langle b_1, \ldots, b_\ell, e \rangle \rangle \in \preceq_W \text{ and} \\
\qquad\qquad b_i \text{ is a state of } \mathcal{B}_i, \text{ for } i \in \{\ell + 1, \ldots, m\} \ \}.
\end{array}
$$

Intuitively, adding a set of behaviors to a system can only extend, and never reduce, the capabilities of the system. Indeed, the additional behaviors do not constrain in any way those already present, while, in general, make the system able to execute more actions. In particular, if a system can simulate a target behavior (on some environment), we expect it to have the same ability, and possibly more, after introducing additional behaviors. The next result proves this intuition, i.e., that when "putting back" a set of behaviors U into system $\mathcal{S}(W)$, by extending $\preceq_W$ as shown above, we are guaranteed to obtain an ND-simulation relation for the (expanded) system $\mathcal{S}(W \cup U)$, *though not necessarily the largest one*.

Lemma 5. *Given a system $\mathcal{S} = \{\mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E}\}$, a target behavior $\mathcal{B}_T$ and its respective enacted behavior $\mathcal{T}_T$ over $\mathcal{E}$, let $W, U \subseteq \{1, \ldots, n\}$ be such that $W \cap U = \emptyset$. The following hold:*

- $\preceq_W \otimes U \subseteq \preceq_{W \cup U}$*;*
- $\preceq_W \otimes U$ *is an ND-simulation relation of $\mathcal{T}_T$ by $\mathcal{T}_{\mathcal{S}(W \cup U)}$.*

Proof. Without loss of generality, consider $W = \{1, \ldots, \ell\}$, and $U = \{\ell + 1, \ldots, m\}$. Suppose that $\langle \langle t, e \rangle, \langle b_1, \ldots, b_\ell, b_{\ell+1}, \ldots, b_m, e' \rangle \rangle \in \preceq_W \otimes U$. Due to the definition of operation $\otimes$, it is the case that $\langle t, e \rangle \preceq_W \langle b_1, \ldots, b_\ell, e' \rangle$. This means that $e = e'$ and that for each $a \in \mathcal{A}$, there exists index $k_a \in W$ satisfying the requirements of the ND-simulation relation definition for system $\mathcal{S}(W)$. Then, $\langle t, e \rangle \preceq_{W \cup U} \langle b_1, \ldots, b_\ell, b_{\ell+1}, \ldots, b_m, e' \rangle$. Indeed, $e = e'$, and for every $a \in \mathcal{A}$, the same index k_a would also satisfy the requirements of the ND-simulation definition for system $\mathcal{S}(W \cup U)$—the new behaviors are not used and they cannot either remove or inhibit other behaviors capabilities. This shows that $\preceq_W \otimes U$ is an ND-simulation relation of $\mathcal{T}_T$ by $\mathcal{T}_{\mathcal{S}(W \cup U)}$ and, hence, $\preceq_W \otimes U \subseteq \preceq_{W \cup U}$, as $\preceq_{W \cup U}$ is the largest ND-simulation relation of $\mathcal{T}_T$ by $\mathcal{T}_{\mathcal{S}(W \cup U)}$. □

As it turns out, adding new behaviors has a minimal impact on the ND-simulation relation, that can be recomputed through simple projection operations. Unfortunately, this is not the case when behaviors become unavailable. As discussed below, this has, in general, a disruptive impact on the ND-simulation relation, which, in order to be recomputed, requires more than just *local* changes. To see this, let $F \subseteq W$ be the set of indexes of those behaviors that become permanently unavailable, and denote by $\preceq_W|_F$ the relation obtained from $\preceq_W$ by *projecting out* all (failed) behaviors $\mathcal{B}_i$ such that $i \in F$. In general, the so-obtained relation just contains (possibly properly) the new largest ND-simulation after failure. Specifically, we have:

Lemma 6. *For $\mathcal{S}$ and $\mathcal{T}_T$ as above, let $W, F \subseteq \{1, \ldots, n\}$ be such that $F \subseteq W$. The following holds:*

- $\preceq_{(W \setminus F)} \subseteq \preceq_W|_F$*;*
- $\preceq_W|_F$ *may not be an ND-simulation relation of $\mathcal{T}_T$ by $\mathcal{T}_{\mathcal{S}(W \setminus F)}$.*

Proof. By Lemma 5, $\preceq_{(W\setminus F)} \otimes F \subseteq \preceq_{(W\setminus F)\cup F}$, that is, $\preceq_{(W\setminus F)} \otimes F \subseteq \preceq_W$. By projecting out F on both relations, we get $\preceq_{(W\setminus F)} \otimes F|_F \subseteq \preceq_W|_F$. Then, since $\preceq \otimes X|_X = \preceq$ for any $\preceq$ and X, $\preceq_{(W\setminus F)} \subseteq \preceq_W|_F$ follows.

It is immediate to find cases where the containment is proper, and hence the second part follows. □

Notice that even though $\preceq_W$ is the largest ND-simulation relation when all behaviors in W are active, the projected relation $\preceq_W|_F$ is *not* necessarily even an ND-simulation relation for the (contracted) system $\mathcal{S}(W \setminus F)$.

In light of these results, we next show how to deal with failure cases (d) and (e).

Permanent unavailability. When a behavior becomes permanently unavailable (cf. case (d)), one cannot wait for it to resume. Instead, one can either continue to executing the composition (controller) and just "hope for the best", i.e., that the failed behavior will not be actually required (because, e.g., some actions occurring in the target behavior are not executed at runtime), or one can "refine" the current composition so as to continue guaranteeing the full realization of the target behavior.

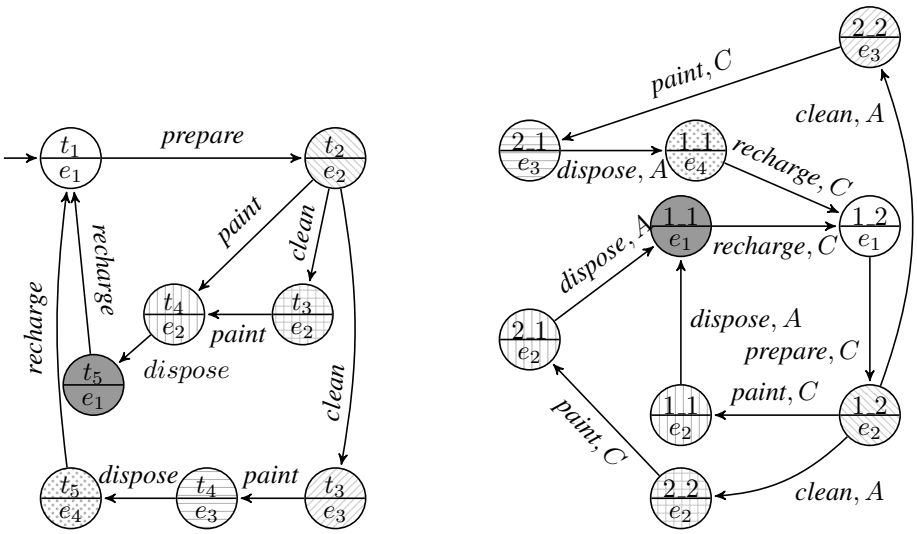
Assume that, at some point, while a composition built from an ND-simulation relation is executing, a set of available behaviors become unavailable. Clearly, the current composition is no longer sound (as some required behaviors might be unavailable), the ND-simulation relation is no longer useful, and one is required to recompute a new one (and a corresponding composition) in order to keep executing the target behavior. Of course, the new ND-simulation relation can always be computed "from scratch," by considering only the set of currently available behaviors. However, from the computational point of view this might not be the best solution, as it does not take advantage of what has been previously computed. In the following, we propose a different approach, based on the above results, that aims at minimizing the required computational effort by *refining*, rather than recomputing, the ND-simulation relation at hand.

Lemma 6 essentially says that, when some behaviors become unavailable, in order to compute the new ND-simulation relation, it is enough executing the *NDSP* algorithm by instantiating $\mathcal{R}_{init}$ with the relation obtained by projecting out the failed components from the current ND-simulation relation. This yields, in general, substantially less algorithm iterations than *NDS*. Indeed, as behaviors become unavailable, the effort to obtain the new largest ND-simulation relation is *systematic* and *incremental*, in that no tuples that were previously discarded are considered again. This, along with Lemma 4 leads to the following

Theorem 7. *Consider $\mathcal{S}$, $\mathcal{B}_T$ and $\mathcal{T}_T$ as above. Let $W \subseteq \{1,\ldots,n\}$ contain the indexes of the behaviors currently working in $\mathcal{S}$ and let $F \subseteq W$ contain the indexes of the behaviors that, at a given point, become permanently unavailable. Then, for every relation β such that $\beta \subseteq \preceq_{(W\setminus F)}$, the following holds:*

$$\preceq_{(W\setminus F)} = \mathit{NDSP}(\mathcal{T}_T, \mathcal{T}_{\mathcal{S}(W\setminus F)}, \preceq_W|_F, \beta).$$

Proof. Direct consequence of Lemmas 4 and 6. □



(a) Enacted Target Arm $\mathcal{T}_T$ (b) Enacted System Behavior

Figure 5: An ND-simulation relation between enacted target behavior $\mathcal{T}_T$ and enacted system $\mathcal{T}_{\mathcal{S}(\{1,3\})}$.

Example 7. Suppose that arm $\mathcal{B}_T$ (Figure 1) is being successfully realized by means of controller P_1 (Figure 3). At some point, assume that arm $\mathcal{B}_2$ breaks down in state b_3, just after painting a block. With $\mathcal{B}_2$ out, controller P_1 cannot guarantee $\mathcal{B}_T$ realization anymore —yet, interestingly, this can be now done by controller P_2 on the new (unexpected) sub-system. To handle such a failure case, first, behavior $\mathcal{B}_2$ is projected out from the ND-simulation relation $\preceq_{\{1,2,3\}}$, thus getting $\preceq_{\{1,2,3\}}|_{\{2\}}$; then, the new largest ND-simulation relation is computed with *NDSP* starting from relation $\preceq_{\{1,2,3\}}|_{\{2\}}$, thus obtaining $\preceq_{\{1,3\}}$ —from which, a new *CG* and a corresponding composition can be derived. The result is shown in Figure 5, where the enacted target behavior is the same as in Figure 4(a), reported here for convenience. Like in Example 4, matching filling patterns individuate pairs in the ND-simulation relation. Observe that tuple $\langle\langle t_3, e_3\rangle, \langle\langle a_2, c_1\rangle, e_3\rangle\rangle$ belongs to relation $\preceq_{\{1,2,3\}}|_{\{2\}}$, but is filtered out by the *NDSP* algorithm (the original tuple $\langle\langle t_3, e_3\rangle, \langle\langle a_2, b_2, c_1\rangle, e_3\rangle\rangle \in \preceq_{\{1,2,3\}}$ relied on $\mathcal{B}_2$ for maintaining the ND-simulation). ■

Resumed behaviors. Consider now the situation where the operating behaviors are those with indexes in W, and others, supposed to be permanently unavailable, become unexpectedly available (cf. case (e)). Let U be the set of indexes of such behaviors, with $U \cap W = \emptyset$. As already observed, this never reduces the capabilities of the whole system but could enhance it with more choices, or, differently said, after behaviors in U become available again, the system can still realize *at least* the same executions as before. However, if one wants to exploit the further capabilities brought by the resumed behaviors, the new largest ND-simulation relation $\preceq_{(W \cup U)}$ must be computed.

In doing so, one can leverage on the fact that $\preceq_{(W\cup U)}$ *contains* relation $\preceq_W \otimes U$ (cf. Lemma 5) and completely neglect, for potential filtering, tuples in $\preceq_W \otimes U$. That is, such tuples can be provided in input to the *NDSP* algorithm as the "sure set."

Theorem 8. *Consider $\mathcal{S}$, $\mathcal{B}_T$ and $\mathcal{T}_T$ as above. Let $W \subseteq \{1,\ldots,n\}$ contain the indexes of the behaviors currently working in $\mathcal{S}$, and $U \subseteq \{1,\ldots,n\}$, with $W \cap U = \emptyset$, the indexes of those that were assumed permanently unavailable but have unexpectedly resumed. Then, for every set α such that $\preceq_{(W\cup U)} \subseteq \alpha$, the following holds:*

$$\preceq_{(W\cup U)} \;=\; \mathit{NDSP}(\mathcal{T}_T, \mathcal{T}_{\mathcal{S}(W\cup U)}, \alpha, \preceq_W \otimes U).$$

Proof. Consequence of Lemmas 4 and 5. □

As it turns out, this requires, in general, less iterations than those required for computing the ND-simulation relation from scratch, as tuples considered by *NDS* are not processed by *NDSP*. Observe that even if new behaviors not appearing in $\{1,\ldots,n\}$ are included in U, the thesis of Lemma 5 still holds. Therefore, if the system is enriched with new behaviors, one can use the largest ND-simulation relation previously computed, in order to save computational efforts, when computing the new ND-simulation relation.

Reusing previous computed ND-simulations. Theorems 7 and 8 essentially show that, by using algorithm *NDSP*, when a behavior resumes or becomes unavailable and a new ND-simulation relation needs to be re-computed, one can take advantage of the ND-simulation relation previously computed. In fact, such theorems can be combined so as to reuse not only the last ND-simulation relation computed, but all those computed in the past (assuming they have been stored).

To see this, let $\mathcal{W} \subseteq 2^{\{1,\ldots,n\}}$, such that $\{1,\ldots,n\} \in \mathcal{W}$, be a set of sets of behavior indices, and assume that the largest ND-simulation relation for each set in $\mathcal{W}$ has been already computed and stored. For $W \notin \mathcal{W}$, in order to compute the (largest) ND-simulation relation $\preceq_W$, one can first define the following sets:

$$\bar{\alpha} = \bigcap\nolimits_{\{W' \in \ni_W^{\mathcal{W}}\}} \preceq_{W'}|_{(W'\setminus W)};$$
$$\bar{\beta} = \bigcup\nolimits_{\{W' \in \Subset_W^{\mathcal{W}}\}} \preceq_{W'} \otimes (W \setminus W');$$

where $\ni_W^{\mathcal{W}}$ and $\Subset_W^{\mathcal{W}}$ stand for the set of *tightest* supersets and subsets, respectively, of W in $\mathcal{W}$, namely:

$$\ni_W^{\mathcal{W}} = \{W' \in \mathcal{W} \mid W \subseteq W' \wedge \forall V \in \mathcal{W}.W \subseteq V \rightarrow V \not\subset W'\};$$
$$\Subset_W^{\mathcal{W}} = \{W' \in \mathcal{W} \mid W' \subseteq W \wedge \forall V \in \mathcal{W}.V \subseteq W \rightarrow W' \not\subset V'\}.$$

Then, by applying Theorems 7 and 8, $\preceq_W$ can simply be computed as follows:

$$\preceq_W \;=\; \mathit{NDSP}(\mathcal{T}_T, \mathcal{T}_{\mathcal{S}(W)}, \bar{\alpha}, \bar{\beta}).$$

Clearly, by using $\mathit{NDSP}(\mathcal{T}_T, \mathcal{T}_{\mathcal{S}}(W), \bar{\alpha}, \bar{\beta})$ to compute $\preceq_W$, the computations already carried out are *maximally reused* to devise other ND-simulation relations, as $\bar{\alpha}$ and $\bar{\beta}$

are the tightest sets one can obtain starting from the ND-simulation relations for sets in $\mathcal{W}$. Of course, once computed $\preceq_W$, CGP_{JIT} can be immediately computed on-the-fly, as before.

We close this section by noting that the kind of failures considered can be seen as *core* classes of breach-of-contract, with respect to the specification. Other forms of failures are clearly conceivable [86, 62, 53], which assume additional information at hand —e.g., a module may announce unavailability duration and/or the state (or possible states) it will join back—, and that can be exploited for failure reaction, thus opening interesting research directions. However, covering a wider range of failure cases is out of the scope of the present paper, and we limit our attention only to the classes presented above.

5. Simulation and Safety Games

In previous Sections, we have shown that the behavior composition problem can be reduced to the problem of finding an ND-simulation relation between two transition systems that, together, describe the original problem instance. Moreover, we have discussed optimization approaches to obtain computational benefits, when computing a new ND-simulation relation in response to different type of failures. In the rest of the paper, we adopt a more pragmatic perspective, and focus on finding effective ways for actually computing an ND-simulation relation. Concretely, we will demonstrate how controller generators can be synthesized by applying model checking techniques.

We begin by laying down the theoretical bases for actually solving the behavior composition problem, and show that an ND-simulation relation can be constructed by resorting to infinite games. In particular, we argue that constructing an ND-simulation relation is equivalent to building a *winning strategy in a safety-game* (cf. [5, 6, 67]).[8] The main motivation behind the use of game structures is the availability of software tools, such as TLV [69], LILY [42], ANZU [43], and MOCHA [4], which provide *(i)* effective procedures for strategy computation; and *(ii)* convenient languages for representing the problem instance in a modular and high-level manner. In fact, next section explains in detail how to solve behavior composition problem instances using the TLV system. We note that, even though not all tools above offer efficient, or more appropriately *optimized*, solution techniques, there are currently promising efforts in this direction (cf., e.g., [42]), so we may likely expect formal synthesis technology to become available as an effective alternative, in the future –similarly to model checking [23].

5.1. Safety-Game structures

We specialize the notion of *game structure* proposed in [67], to deal with synthesis problems for invariant properties. Roughly speaking, a safety-game structure represents a game played by two players, *system* and *controller*,[9] where, at each turn, the

[8] *Safety* games are those where some condition—the invariant property—needs to be *always* maintained, in our case: $\mathcal{T}_S$ is *always* able to "locally" (i.e., state-by-state) mimicking $\mathcal{T}_T$.

[9] To avoid confusion with our previous notation, we adopt a notation different from that of [67], in which the players are the *environment* (our system) and the *system* (our controller)

former moves and the latter replies. Moves are subject to constraints (i.e., only some moves/replies are allowed in a given game state). Intuitively, the controller's objective is to be always able to reply to system's moves so as to satisfy a given (goal) property, while the system tries to avoid this.

Throughout the rest of the paper, we assume to deal with infinite-play (though finite-state) games, possibly obtained by introducing fake loops, as customary, e.g., in LTL verification. Infinite plays are assumed for technical convenience only, so as to handle all plays—finite or infinite—in a uniform way. This assumption, however, does not limit the power of game structures (for technical details about plays, see below).

A *safety-game structure* ($\Box$-GS, for short) is a tuple $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_s, \rho_c, \Box\varphi \rangle$, where:

- $\mathcal{V} = \{v_1, \ldots, v_n\}$ is the finite set of *state variables*, which range over finite domains $V_1, \ldots, V_n$, respectively. Set $\mathcal{V}$ is partitioned into sets $\mathcal{X} = \{v_1, \ldots, v_m\}$ (the *system variables*) and $\mathcal{Y} = \{v_{m+1}, \ldots, v_n\}$ (the *controller variables*). A *valuation* of variables in $\mathcal{V}$ is a total function $val : \mathcal{V} \longrightarrow \bigcup_{i=1}^{n} V_i$ such that $val(v_i) \in V_i$, for each $i \in \{1, \ldots, n\}$. For convenience, we represent valuations as vectors $\vec{s} = \langle s_1, \ldots, s_n \rangle \in V$, where $V = V_1 \times \ldots \times V_n$ and $s_i = val(v_i)$, for each $i \in \{1, \ldots, n\}$. Consequently, (sub)valuations of variables in $\mathcal{X}$ (resp. $\mathcal{Y}$) are represented by vectors $\vec{x} \in X$ ($\vec{y} \in Y$), with $X = V_1 \times \cdots \times V_m$ ($Y = V_{m+1} \times \cdots \times V_n$). A *game state* is a valuation $\vec{s} = \langle s_1, \ldots, s_n \rangle \in V$, and its sub-vectors $\vec{x} = \langle s_1, \ldots, s_m \rangle \in X$ and $\vec{y} = \langle s_{m+1}, \ldots, s_n \rangle \in Y$ are the corresponding *system* and *controller* states, respectively. By a slight abuse of notation, we shall also write $\vec{s} = \langle \vec{x}, \vec{y} \rangle$.

- Θ is a formula representing the initial states of the game. Technically, it is a boolean combination of expressions of the form $(v_i = s_i)$, where $v_i \in \mathcal{V}$, for some $i \in \{1, \ldots, n\}$, and $s_i \in V_i$. Each of such expressions is an assignment constraint, satisfied by state $\vec{s} = \langle s_1, \ldots, s_n \rangle$ if $val(v_i) = s_i$. In general, not all variables in $\mathcal{V}$ are required to occur in Θ. Given a game state $\langle \vec{x}, \vec{y} \rangle \in V$, we write $\langle \vec{x}, \vec{y} \rangle \models \Theta$ if $\langle \vec{x}, \vec{y} \rangle$ satisfies, in the obvious way, the boolean combination of assignment constraints specified by Θ.

- $\rho_s \subseteq X \times Y \times X$ is the *system transition relation*, which relates each game state to its possible successor system states.

- $\rho_c \subseteq X \times Y \times X \times Y$ is the *controller transition relation*, relating each game state together with one if its successor system states (i.e., move), to possible successor controller states.

- $\Box\varphi$ is the *goal* formula, representing the invariant property to be guaranteed, where φ has the same form as Θ above.

The above definition is completed by enforcing the infinite-play game assumption, informally stated above, by requiring that for each game state $\langle \vec{x}, \vec{y} \rangle \in V$:

- there exists an $\vec{x}' \in X$ such that $\rho_s(\vec{x}, \vec{y}, \vec{x}')$; and

- for all $\vec{x}'$ such that $\rho_s(\vec{x}, \vec{y}, \vec{x}')$, there exists a $\vec{y}' \in Y$ such that $\rho_c(\vec{x}, \vec{y}, \vec{x}', \vec{y}')$.

In the rest of the paper, when no ambiguity arises, we will use "game structure" or simply "game" to refer to a safety-game structure. The idea behind game structures is that, with the game in some state $\vec{s} = \langle \vec{x}, \vec{y} \rangle$, the system *moves*, by choosing $\vec{x}'$ such that $\rho_s(\vec{x}, \vec{y}, \vec{x}')$, and the controller then *replies*, by choosing $\vec{y}'$ such that $\rho_c(\vec{x}, \vec{y}, \vec{x}', \vec{y}')$. Each pair of system move and subsequent controller reply defines a game transition from $\vec{s} = \langle \vec{x}, \vec{y} \rangle$ to state $\vec{s}' = \langle \vec{x}', \vec{y}' \rangle$. Note that the controller is allowed to *observe* the system move before replying, as witnessed by the presence of $\vec{x}'$ in $\rho_c(\vec{x}, \vec{y}, \vec{x}', \vec{y}')$.

With the formal notion of games at hand, let us next define the corresponding dynamics and the notion of *winning* in a game. A game state $\langle \vec{x}', \vec{y}' \rangle$ is a *successor* of a state $\langle \vec{x}, \vec{y} \rangle$ iff $\rho_s(\vec{x}, \vec{y}, \vec{x}')$ and $\rho_c(\vec{x}, \vec{y}, \vec{x}', \vec{y}')$. A *game play* starting from state $\langle \vec{x}_0, \vec{y}_0 \rangle \in V$ is an infinite sequence of states $\eta = \langle \vec{x}_0, \vec{y}_0 \rangle \langle \vec{x}_1, \vec{y}_1 \rangle \cdots$ such that for each $j \geq 0$, $\langle \vec{x}_{j+1}, \vec{y}_{j+1} \rangle$ is a successor of $\langle \vec{x}_j, \vec{y}_j \rangle$. Clearly, by the infinite-play assumption, every game always admits at least a play. Intuitively, plays capture (infinite) sequences of game states obtained by alternating system moves and controller replies. A play is said *winning* (for the controller) if it satisfies the winning condition $\Box\varphi$, that is, $\langle \vec{x}_i, \vec{y}_i \rangle \models \varphi$, for all $i \geq 0$. The intuition is that the play remains within a set of *safe* states, i.e., which satisfy the invariant property.

A (controller) *strategy* is a partial function $f : (X \times Y) \times X^+ \mapsto Y$ such that for every (finite) sequence of game states $\lambda : \langle \vec{x}_0, \vec{y}_0 \rangle \cdots \langle \vec{x}_n, \vec{y}_n \rangle$ and for every system state $\vec{x}' \in X$ such that $\rho_s(\vec{x}_n, \vec{y}_n, \vec{x}')$, it is the case that $\rho_c(\vec{x}_n, \vec{y}_n, \vec{x}', f(\langle \vec{x}_0, \vec{y}_0 \rangle, \vec{x}_1 \cdots \vec{x}_n \vec{x}'))$ holds. A play $\eta = \langle \vec{x}_0, \vec{y}_0 \rangle \langle \vec{x}_1, \vec{y}_1 \rangle \cdots$ is *compliant with a strategy* f if $\vec{y}_\ell = f(\langle \vec{x}_0, \vec{y}_0 \rangle, \vec{x}_1 \cdots \vec{x}_\ell)$, for all $\ell > 0$, that is, intuitively, all controller replies in the play match those the strategy prescribes. A strategy f is *winning from a state* $\vec{s}$ if all plays starting from $\vec{s}$ and compliant with f are winning. A strategy f is *winning* for a game G if f is winning from all of G's initial states. We say that a game is *winning* (for the controller) if there exists a winning strategy for it, and that a game state is *winning* if there exists a winning strategy from that state. The *winning set* of a game G is the set of all winning states of that game.

Intuitively, a game is winning if the controller can *control* the game evolution, through a winning strategy that affects only $\mathcal{Y}$ variables, so as to guarantee that the winning condition φ holds along all game plays, no matter how the system moves happen to be. In order to prove that a game is winning, one thus needs to prove the existence of a winning strategy, which is clearly equivalent to show that the set of game's initial states is a subset of the winning set.

Next, we show how one can compute the winning set of a given safety-game structure $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_s, \rho_c, \Box\varphi \rangle$. The key ingredient is the following operator $\pi : 2^V \longrightarrow 2^V$ (see [5, 67]):

$$\pi(P) \doteq \{\langle \vec{x}, \vec{y} \rangle \in V \mid \forall\, \vec{x}'.\rho_s(\vec{x}, \vec{y}, \vec{x}') \rightarrow \exists\, \vec{y}'.\rho_c(\vec{x}, \vec{y}, \vec{x}', \vec{y}') \wedge \langle \vec{x}', \vec{y}' \rangle \in P\}.$$

Intuitively, given a set of game states $P \subseteq V$, $\pi(P)$ denotes the set of P*'s controllable predecessors*, that is, the set of all game states from which the controller can force the play to reach a state in P, *no matter how* the system happens to move. Using this operator, Algorithm 3 can be applied to compute the set of all G's winning states, as proven by Theorem 9 below.

The algorithm essentially computes a fixpoint, starting from the set of all game states that satisfy the goal formula φ. After the first iteration, W' (the next "candidate"

Algorithm 3: *WIN* – Computes a safety-game structure's winning set

```
1 W' := {⟨x⃗, y⃗⟩ ∈ V | ⟨x⃗, y⃗⟩ ⊨ φ};
2 repeat
3 |   W := W' ;                          // current candidate set
4 |   W' := W ∩ π(W) ;                  // compute next candidate set
5 until (W = W') ;
6 return W;
```

set) contains all those game states that satisfy φ, and from which the controller has a strategy to force, in *one* step, the game to a state that satisfies φ. The process is then iterated, by refining the current candidate set W, ruling out all those states that are not controllable predecessors of W. At the end of the n-th iteration, set W contains all those game states from which the controller has a strategy to make the game traverse n states satisfying φ, independently of system moves. When a fixpoint is reached, n can be replaced by ∞. Termination of the algorithm is evident, as no new states are ever added to W. The following theorem proves that the obtained set is indeed the winning set.

Theorem 9. *Let $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_s, \rho_c, \Box\varphi \rangle$ be a safety-game structure as above, and let W be obtained by running Algorithm 3 on G. Given a game state $\langle \vec{x}_0, \vec{y}_0 \rangle \in V$, there exists a winning strategy from $\langle \vec{x}_0, \vec{y}_0 \rangle$ if and only if $\langle \vec{x}_0, \vec{y}_0 \rangle \in W$.*

Proof. (IF PART) When the algorithm returns, it is the case that $W' = W$. Being $W' = W \cap \pi(W)$, we have that $W = W \cap \pi(W)$ and therefore $W \subseteq \pi(W)$. Hence, by definition of $\pi(W)$, the following holds:

$$\forall \langle \vec{x}, \vec{y} \rangle \in W, \forall \vec{x}' \in X. \rho_s(\vec{x}, \vec{y}, \vec{x}') \rightarrow \Phi(\vec{x}, \vec{y}, \vec{x}') \neq \emptyset, \tag{1}$$

where $\Phi(\vec{x}, \vec{y}, \vec{x}') = \{\vec{y}' \mid \rho_c(\vec{x}, \vec{y}, \vec{x}', \vec{y}') \wedge \langle \vec{x}', \vec{y}' \rangle \in W\}$ represents, informally, the set of all "good" moves when the system has just played $\vec{x}'$ from game state $\langle \vec{x}, \vec{y} \rangle$.

Using set Φ, we consider next any strategy $f(\langle \vec{x}, \vec{y} \rangle, \lambda)$ satisfying the following constraint (here $\ell \geq 1$):

$$f(\langle \vec{x}_0, \vec{y}_0 \rangle, \vec{x}_1 \cdots \vec{x}_\ell) \in \Phi(\vec{x}_{\ell-1}, \vec{y}_{\ell-1}, \vec{x}_\ell), \qquad \text{whenever } \Phi(\vec{x}_{\ell-1}, \vec{y}_{\ell-1}, \vec{x}_\ell) \neq \emptyset,$$

where $\vec{y}_{\ell-1} = f(\langle \vec{x}_0, \vec{y}_0 \rangle, \vec{x}_1 \cdots \vec{x}_{\ell-1})$, when $\ell > 1$ (when $\ell = 1$, $\vec{y}_{\ell-1} = \vec{y}_0$).

Next, let us prove that strategy f is indeed a winning strategy from the initial game state. To that end, all we have to do is to show that for any game play $\eta = \langle \vec{x}_0, \vec{y}_0 \rangle \langle \vec{x}_1, \vec{y}_1 \rangle \cdots$ from game state $\langle \vec{x}_0, \vec{y}_0 \rangle$ and *compliant* with strategy f, it is the case that $\langle \vec{x}_i, \vec{y}_i \rangle \in W$, for all $i \geq 0$. Observe that for any game state $\langle \vec{x}, \vec{y} \rangle \in W$, it is the case that $\langle \vec{x}, \vec{y} \rangle \models \varphi$. This is because the algorithm starts exactly with the game states that satisfy φ (line 1) and only *removes* states from the candidate set (line 4). So, let us prove that $\langle \vec{x}_i, \vec{y}_i \rangle \in W$, for all $i \geq 0$, by induction on the index i. The base case when $i = 0$ is trivial, as $\langle \vec{x}_0, \vec{y}_0 \rangle \in W$ holds by assumption.

Next, suppose that for $\langle \vec{x}_i, \vec{x}_i \rangle \in W$, for all $i \leq k$, for some $k \geq 0$. Because η is a game play, it is the case that $\rho_s(\vec{x}_k, \vec{y}_k, \vec{x}_{k+1})$. Also, by the induction hypothesis,

$\langle \vec{x}_k, \vec{y}_k \rangle \in W$. Therefore, by applying equation (1), we have that $\Phi(\vec{x}_k, \vec{y}_k, \vec{x}_{k+1}) \neq \emptyset$. From this—together with the fact that $\vec{y}_k = f(\langle \vec{x}_0, \vec{y}_0 \rangle, \vec{x}_1 \cdots \vec{x}_k)$ when $k > 0$, as play η is complaint with f—it follows that $f(\langle \vec{x}_0, \vec{y}_0 \rangle, \vec{x}_1 \cdots \vec{x}_{k+1}) = \vec{y}_{k+1} \in \Phi(\vec{x}_k, \vec{y}_k, \vec{x}_{k+1})$, and by definition of set Φ, $\langle \vec{x}_{k+1}, \vec{y}_{k+1} \rangle \in W$ follows.

(ONLY-IF PART) Let W_i be the version of W at i-th iteration (at line 5), where $1 \leq i \leq N$, assuming the algorithm terminates in N iterations and hence it returns W_N. We show, by induction on index i, that for any game state $\langle \vec{x}, \vec{y} \rangle$, if $\langle \vec{x}, \vec{y} \rangle \notin W_i$, and hence $\langle \vec{x}, \vec{y} \rangle \notin W_N$, then the system can always force from state $\langle \vec{x}, \vec{y} \rangle$ to reach, *in at most i steps*, a state $\langle \vec{x}', \vec{y}' \rangle$ such that $\langle \vec{x}', \vec{y}' \rangle \not\models \varphi$.

For the base case, suppose that $\langle \vec{x}, \vec{y} \rangle \notin W_1$. Due to lines 1 and 3 of Algorithm 3, set W_1 is exactly those and only those states that satisfy φ, that is, $\langle \vec{x}, \vec{y} \rangle \not\models \varphi$, and the claim follows trivially. Now, assume the claim holds for all $i \leq k$ and consider a game state $\langle \vec{x}, \vec{y} \rangle \notin W_{k+1}$. If $\langle \vec{x}, \vec{y} \rangle \notin W_k$, then the game state was removed at some previous iteration $j \leq k$, and by the induction hypothesis, the system can force all plays to violate the goal in at most k (and hence $k+1$) steps. So, suppose on the other hand that $\langle \vec{x}, \vec{y} \rangle \in W_k$, that is, the game state was removed at the $k+1$ iteration (in line 4). From line 4 in the algorithm, we know that $W_{k+1} = W_k \cap \pi(W_k)$. Since $\langle \vec{x}, \vec{y} \rangle \in W_k$ but $\langle \vec{x}, \vec{y} \rangle \notin W_{k+1}$, it follows that $\langle \vec{x}, \vec{y} \rangle \notin \pi(W_k)$. By definition of π, there the system has a move $\vec{x}' \in X$, with $\rho_s(\vec{x}, \vec{y}, \vec{x}')$, such that for all controller replies $\vec{y}' \in Y$ with $\rho_c(\vec{x}, \vec{y}, \vec{x}', \vec{y}')$, it is the case that $\langle \vec{x}', \vec{y}' \rangle \notin W_k$. By the induction hypothesis, the system can always force from game state $\langle \vec{x}', \vec{y}' \rangle$ to reach, *in at most k steps*, a state that violates φ. Thus, by playing $\vec{x}'$ from the initial state $\langle \vec{x}, \vec{y} \rangle$, the system is always able to force violating φ *in at most $k+1$ steps*.

Now, suppose that there exits a winning strategy f from state $\langle \vec{x}_0, \vec{y}_0 \rangle$, but on the contrary, that $\langle \vec{x}_0, \vec{y}_0 \rangle \notin W$, or what is the same, that $\langle \vec{x}_0, \vec{y}_0 \rangle \notin W_N$. By our reasoning above, the system can always force the game to violate φ in at most N steps. This implies that there exits a game play $\eta = \langle \vec{x}_0, \vec{y}_0 \rangle \langle \vec{x}_1, \vec{y}_1 \rangle \cdots$ (i.e., starting from $\langle \vec{x}_0, \vec{y}_0 \rangle$) and complaint with f such that for some $i < N$, $\langle \vec{x}_1, \vec{y}_1 \rangle \not\models \varphi$ applies. Hence, f would not be a winning strategy from $\langle \vec{x}_0, \vec{y}_0 \rangle$, a contradiction is reached, and it follows then that $\langle \vec{x}_0, \vec{y}_0 \rangle \in W$ must apply. □

Importantly, once the winning set is computed, it can be used to define a winning strategy. To see this, assume that $\eta = \langle \vec{x}_0, \vec{y}_0 \rangle \ldots \langle \vec{x}_n, \vec{y}_n \rangle$ is the prefix of a play executed up to some point. For each next system move $\vec{x}' \in X$ (such that $\rho_s(\vec{x}_n, \vec{y}_n, \vec{x}')$), one can define $f(\langle \vec{x}_0, \vec{y}_0 \rangle, \vec{x}_1 \ldots \vec{x}_n \vec{x}') = \vec{y}'$, by taking any reply $\vec{y}'$ such that $\langle \vec{x}', \vec{y}' \rangle \in W$ (and $\rho_c(\vec{x}_n, \vec{y}_n, \vec{x}', \vec{y}')$). Indeed, such a condition guarantees that the controller has a winning strategy from $\langle \vec{x}', \vec{y}' \rangle$, informally meaning that it can force the (future extension of the) play to maintain φ.

5.2. From Composition to Safety Games

Next, we show how the behavior composition problem can be reduced in practice to the problem of *synthesizing a winning strategy in a safety-game structure*. In order to do so, we need to identify which place each component of a composition problem –target behavior, available behaviors, environment, and composition controller– occupies in the game representation, that is, players controller and system need to be defined

for the particular setting. Generally speaking, when composing behaviors, a controller can be seen as a strategy, i.e., a function of system histories that returns decisions, so, from this perspective, it seems very natural representing the composition as the (synthesized strategy for) controller player, and all other components combined together as the system player.

Let $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$ be a system and $\mathcal{B}_T$ a target behavior over $\mathcal{E}$, where $\mathcal{B}_i = \langle B_i, b_{i0}, G_i, F_i, \varrho_i \rangle$, for $i = 1, \ldots, n, T$, and $\mathcal{E} = \langle \mathcal{A}, E, e_0, \rho \rangle$. We derive a safety-game structure $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_s, \rho_c, \Box\varphi \rangle$ that captures the relationship between the target behavior and the system, as follows:

1. $\mathcal{V} = \{b_1, \ldots, b_n, e, b_T, a, ind\}$, where:
   - b_i ranges over $\hat{B}_i = B_i \cup \{\sharp\}$, for each for $i \in \{1, \ldots, n, T\}$;
   - e ranges over $\hat{E} = E \cup \{\sharp\}$;
   - a ranges over $\hat{\mathcal{A}} = \mathcal{A} \cup \{\sharp\}$;
   - *ind* ranges over $\{1, \ldots, n\} \cup \{\sharp\}$.
2. $\mathcal{X} = \{b_1, \ldots, b_n, e, b_T, a\}$ is the set of player system variables, and $X = \hat{B}_1 \times \cdots \times \hat{B}_n \times \hat{E} \times \hat{B}_T \times \hat{\mathcal{A}}$ represents the set of all possible corresponding valuations.
3. $\mathcal{Y} = \{ind\}$ is the (singleton) set of player controller variables, and $Y = \{1, \ldots, n, \sharp\}$ represents the set of all possible corresponding valuations.
4. $\Theta = \bigwedge_{i \in \{1,\ldots,n,T\}} (b_i = \sharp) \wedge (a = \sharp) \wedge (ind = \sharp) \wedge (e = \sharp)$.
5. $\rho_s \subseteq X \times Y \times X$ is such that $\langle \langle b_1, \ldots, b_n, e, b_T, a \rangle, ind, \langle b'_1, \ldots, b'_n, e', b'_T, a' \rangle \rangle \in \rho_s$ iff one of the following cases applies:
   (a) $b_1, \ldots, b_n, e, b_T, a = \sharp$, $ind = \sharp$, and $b'_i = b_{i0}$, for each $i \in \{1, \ldots, n, T\}$, and $e' = e_0$;
   (b) $ind \neq \sharp$ and
      i. there exists a transition $b_T \xrightarrow{g_T, a} b'_T$ in $\mathcal{B}_T$ such that $g_T(e) = \top$;
      ii. there exists a transition $b_{ind} \xrightarrow{g, a} b'_{ind}$ in $\mathcal{B}_{ind}$ such that $g(e) = \top$;
      iii. $b_i = b'_i$, for all $i \in \{1, \ldots, n\} \setminus \{ind\}$;
      iv. there exists a transition $e \xrightarrow{a} e'$ in $\mathcal{E}$; or
   (c) $e' = e$, and $b'_i = b_i$, for each $i \in \{1, \ldots, n, T\}$, and either $ind = \sharp$ but $b_1, \ldots, b_n, e, b_T, a \neq \sharp$, or $ind \neq \sharp$ and one of the following holds:
      i. there is no transition $b_T \xrightarrow{g_T, a} b''_T$ in $\mathcal{B}_T$ such that $g_T(e) = \top$;
      ii. there is no transition $b_{ind} \xrightarrow{g, a} b'_{ind}$ in $\mathcal{B}_{ind}$ such that $g(e) = \top$;
      iii. $b_i \neq b'_i$, for some $i \in \{1, \ldots, n\} \setminus \{ind\}$;
      iv. there is no transition $e \xrightarrow{a} e''$ in $\mathcal{E}$;

   and either $a' = \sharp$ or there exists a transition $b'_T \xrightarrow{g', a'} b''_T$ in $\mathcal{B}_T$, for some $b''_T \in B_T$, such that $g'(e') = \top$ and a transition $e' \xrightarrow{a'} e''$ in $\mathcal{E}$, for some $e'' \in E$.
6. $\langle \langle b_1, \ldots, b_n, e, b_T, a \rangle, ind, \langle b'_1, \ldots, b'_n, e', b'_T, a' \rangle, ind' \rangle \in \rho_c$ iff $ind' \neq \sharp$.
7. Formula φ is defined depending on current system, target, and environment state, current requested action and current behavior selection:

$$\varphi(b_1, \ldots, b_n, e, b_T, a, ind) \doteq \Theta \vee [(\bigwedge_{i=1}^{n} \neg fail_i) \wedge (final_T \rightarrow \bigwedge_{i=1}^{n} final_i)],$$

   where for each $i \in \{1, \ldots, n, T\}$:

- $\mathit{fail}_i \doteq \mathit{ind} = i \wedge \bigwedge_{\langle b_i, g, a, s' \rangle \in \varrho_i} g(e) = \bot$;[10]
- $\mathit{final}_i \doteq \bigvee_{b \in F_i} (b_i = b)$.

Intuitively, the system player represents all possible evolutions of $\mathcal{S}$ generated by legal executions of $\mathcal{B}_T$, which are indeed the only evolutions relevant to our problem. Each complete valuation of variables in $\mathcal{V}$ captures the current state of both the system (variables $b_1, \ldots, b_n$, and e) and the target behavior (variable b_T), the action to be performed next (variable a), and the available behavior selected to perform the action (variable *ind*). For technical convenience, a special value $\sharp$ is used, both for actions and states, to represent a single, domain independent, initial game state $\langle \sharp, \ldots, \sharp \rangle$ as well as requests for no action (when $a = \sharp$).

As for the evolution of the game, the player system's transition relation ρ_s accounts for the synchronous evolution of the system and the target behavior. Condition (5a) states that from the initial distinguished state there exists one and only one transition, leading the system player to the state representing $\mathcal{S}$'s initial state. Condition (5b) encodes the evolution of system $\mathcal{S}$ when the controller has instructed some behavior to act, i.e., $\mathit{ind} \in \{1, \ldots, n\}$. Basically, provided the delegation is feasible, the new system player's state encodes the correct evolution of the target (condition 5(b)i), the selected available behavior (condition 5(b)ii), the non-selected behaviors (condition 5(b)iii), and the environment (condition 5(b)iv). When the behavior delegation is not feasible with respect to the composition problem (e.g., the selected behavior cannot perform the requested action or the action is not legal in current environment or target state), behaviors and environment are assumed to stay still (condition 5c). Finally, the last constraint in the definition of ρ_s states that the next requested action can either be $\sharp$ (i.e., no request) or one that conforms with the target behavior logic. Observe that in a certain game state, transition function ρ_s may allow several different system player's moves, thus reflecting the non-determinism coming from the available behaviors, the environment, as well as from target action requests.

The rules for controller player's moves are simpler, as such player is allowed to arbitrarily assign any available behavior index in any of its moves (condition 6).

To fully comply with our definition of safety-game structures given in Section 5.1, we need to show that G satisfies the infinite-play assumption. For legibility, from now on, when $\vec{x}_i = \langle b_{1i}, \ldots, b_{ni}, e_i, b_{Ti}, a_i \rangle$ is a system player state in G, we will use $\mathit{com}_T(\vec{x}_i) = \langle b_{Ti}, e_i \rangle$ and $\mathit{com}_{\mathcal{S}}(\vec{x}_i) = \langle b_{1i}, \ldots, b_{ni}, e_i \rangle$ to project the enacted target and the enacted system states encoded in $\vec{x}_i$, respectively, and $a(\vec{x}_i) = a_i$ to project the action request encoded in $\vec{x}_i$.

Lemma 10. *Let G be the safety-game structure derived for a behavior composition problem, as above. Then, for each game state $\langle \vec{x}, \vec{y} \rangle$, there exists $\vec{x}'$ such that $\rho_s(\vec{x}, \vec{y}, \vec{x}')$, and for each such $\vec{x}'$ there exists $\vec{y}'$ such that $\rho_c(\vec{x}, \vec{y}, \vec{x}', \vec{y}')$.*

Proof. Straightforward from the facts that: *(i)* if cases (5a) and (5b) do not account for *any* system player's move, then case (5c) will do and $\rho_s(\vec{x}, \vec{y}, \vec{x}')$ will hold with $\vec{x}'$

[10] We assume an empty set of conjuncts is equal to $\bot$.

matching $\vec{x}$ except, possibly, for $a(\vec{x}')$; and *(ii)* for every $\vec{x}, \vec{y}$ and $\vec{x}'$, $\rho_c(\vec{x}, \vec{y}, \vec{x}', \vec{y}')$ holds, for $\vec{y}' \in \{1, \ldots, n\}$. □

Once proven that G is a legal safety-game structure, we prove a useful property of (certain) successor game states. In words, Lemma 11 below says that, under particular assumptions, a game successor captures a legal evolution of the enacted target behavior $\mathcal{T}_T$ and the enacted system $\mathcal{T}_\mathcal{S}$. In addition, provided the successor game state encodes an actual action request, such request conforms with the enacted target behavior.

Lemma 11. *Let G be the safety-game structure derived for a behavior composition problem, as above. Let $\langle \vec{x}, \vec{y} \rangle$ be a (non-initial) game state of G such that $\langle \vec{x}, \vec{y} \rangle \not\models \Theta$, $\vec{y} \neq \sharp$, and there exist transitions $com_\mathcal{S}(\vec{x}) \xrightarrow{a(\vec{x}),\vec{y}} s_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ and $com_T(\vec{x}) \xrightarrow{a(\vec{x})} s_T$ in $\mathcal{T}_T$, for some $s_\mathcal{S} \in S_\mathcal{S}$ and $s_T \in S_T$. Then, $\langle \vec{x}', \vec{y}' \rangle$ is a successor state of $\langle \vec{x}, \vec{y} \rangle$ iff*

- *$com_T(\vec{x}) \xrightarrow{a(\vec{x})} com_T(\vec{x}')$ in $\mathcal{T}_T$;*
- *$com_\mathcal{S}(\vec{x}) \xrightarrow{a(\vec{x}),ind} com_\mathcal{S}(\vec{x}')$ in $\mathcal{T}_\mathcal{S}$; and*
- *if $a(\vec{x}') \neq \sharp$, then there exists $s'_T \in S_T$ such that $com_T(\vec{x}') \xrightarrow{a(\vec{x}')} s'_T$ in $\mathcal{T}_T$.*

Proof. All three claims follow directly from G's ρ_c definition (see condition 5). The first claim follows from conditions 5(b)i and 5(b)iv. The second one is a consequence of conditions 5(b)ii, 5(b)iii, and 5(b)iv. Finally, the third claim follows from the constraint on $a(\vec{x}')$. □

Finally, consider the goal formula φ. As for the first disjunct, it is trivially satisfied by the initial state. Concerning the second one, it is better understood by looking at subformulae $fail_i$ and $final_i$. The former holds if behavior $\mathcal{B}_i$ is selected (i.e., $ind = i$), but cannot execute the requested action a, that is, each transition outgoing from its current state b_i for action a has its guard not satisfied by the current environment state e. The latter holds if the target behavior is in a final state, but not all available behaviors do. Essentially, φ requires that the controller player makes an *adequate* decision, in the sense of never selecting a behavior that may not be able to execute the current requested action.

Once the game structure is build, the problem we deal with is that of synthesizing a (winning) strategy for the controller player that guarantees φ to hold along all possible plays starting from the initial state $\langle \langle \sharp, \ldots, \sharp \rangle, \sharp \rangle$. We shall demonstrate next that this corresponds to synthesizing a composition. More specifically, we show that by computing G's winning set, one is able to derive the controller generator CG. Recall that, in order to define the *CG*, one needs first to build the largest ND-simulation relation (see Section 3). We start by showing that this is in fact equivalent to computing G's maximal winning set.

Theorem 12. *Let $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$ be a system and $\mathcal{B}_T$ a target behavior over $\mathcal{E}$. Let $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_s, \rho_c, \Box\varphi \rangle$ be a $\Box$-GS derived as above from $\mathcal{S}$ and $\mathcal{B}_T$, and let $W \subseteq V$ be the maximal set of controller winning states for G. Then, $\langle \vec{x}_0, \vec{y}_0 \rangle = \langle \langle b_1, \ldots, b_n, e, b_T, a \rangle, ind \rangle \in W$ if and only if $\langle b_T, e \rangle \preceq \langle b_1, \ldots, b_n, e \rangle$.*

Proof. (IF PART) We show that a winning strategy f from the (unique) game's initial state does exist. This, along with Theorem 9, is enough to show that $\langle\langle b_1, \ldots, b_n, e, b_T, a\rangle, \mathit{ind}\rangle \in W$. So, consider a strategy $f(\langle \vec{x}, \vec{y}\rangle, \lambda)$ such that ($k \geq 1$)

$$f(\langle \vec{x}_0, \vec{y}_0\rangle, \vec{x}_1 \cdots \vec{x}_k) \in \omega(\langle \mathit{com}_T(\vec{x}_k), \mathit{com}_{\mathcal{S}}(\vec{x}_k)\rangle, a(\vec{x}_k)),$$

whenever $a(\vec{x}_k) \neq \sharp$ and $\mathit{com}_T(\vec{x}_k) \preceq \mathit{com}_{\mathcal{S}}(\vec{x}_k)$ hold, where $\omega(\cdot,\cdot)$ is the output function of the controller generator of $\mathcal{S}$ for $\mathcal{B}_T$ (see page 17).

Let us prove that for any f-complaint game play $\eta = \langle \vec{x}_0, \vec{y}_0\rangle\langle \vec{x}_1, \vec{y}_1\rangle \cdots$, it is the case that $\mathit{com}_T(\vec{x}_i) \preceq \mathit{com}_{\mathcal{S}}(\vec{x}_i)$ (i.e., $\langle b_{Ti}, e_i\rangle \preceq \langle b_{1i}, \ldots, b_{ni}, e_i\rangle$), for all $i \geq 1$.

So, the base case (i.e., when $i = 1$) is trivial: by condition 5a in G's definition, $\vec{x}_1 = \langle b_{10}, \ldots, b_{n0}, e_0, b_{T0}, a_0\rangle$ (for some $a_0 \in \hat{\mathcal{A}}$), and $\langle b_{T0}, e_0\rangle \preceq \langle\langle b_{10}, \ldots, b_{n0}\rangle, e_0\rangle$ is true by assumption.

Consider now game state $\langle \vec{x}_{k+1}, \vec{y}_{k+1}\rangle$. By induction hypothesis, we know that previous game state $\langle \vec{x}_k, \vec{y}_k\rangle$ is such that $\mathit{com}_T(\vec{x}_k) \preceq \mathit{com}_{\mathcal{S}}(\vec{x}_k)$. If $a(\vec{x}_k) = \sharp$, case 5c ought to apply for the step between the k and $k+1$ game step, which implies that $\mathit{com}_T(\vec{x}_{k+1}) = \mathit{com}_T(\vec{x}_k)$ and $\mathit{com}_{\mathcal{S}}(\vec{x}_{k+1}) = \mathit{com}_{\mathcal{S}}(\vec{x}_k)$, and therefore $\mathit{com}_T(\vec{x}_{k+1}) \preceq \mathit{com}_{\mathcal{S}}(\vec{x}_{k+1})$. Observe that cases 5a and 5b may not apply, because the game initial state may never repeat itself in any game play (see no game variable b_i can ever take value $\sharp$ again) and no behavior can ever make a transition wrt "action" $\sharp$ (condition 5(b)ii will always be false in this case).

Assume next that $a(\vec{x}_k) \neq \sharp$. Since $\mathit{com}_T(\vec{x}_k) \preceq \mathit{com}_{\mathcal{S}}(\vec{x}_k)$, $\langle \mathit{com}_T(\vec{x}_k), \mathit{com}_{\mathcal{S}}(\vec{x}_k)\rangle \in \Sigma$ is a state in the controller generator CG (see page 3). Also, by construction of f, we have that $\vec{y}_k \in \omega(\langle \mathit{com}_T(\vec{x}_k), \mathit{com}_{\mathcal{S}}(\vec{x}_k)\rangle, a(\vec{x}_k))$. By definition of CG's output function ω, we have that $\langle \mathit{com}_T(\vec{x}_k), \mathit{com}_{\mathcal{S}}(\vec{x}_k)\rangle \xrightarrow{a(\vec{x}_k),\vec{y}_k} \langle s'_T, s'_{\mathcal{S}}\rangle$ in CG, for some $s'_T \in S_T$ and $s_{\mathcal{S}} \in S'_{\mathcal{S}}$. By CG's transition relation ϑ, this means that $\mathit{com}_T(\vec{x}_k) \xrightarrow{a(\vec{x}_k)} s'_T$ and $\mathit{com}_{\mathcal{S}}(\vec{x}_k) \xrightarrow{a(\vec{x}_k),\vec{y}_k} s'_{\mathcal{S}}$. Then, by applying Lemma 11, we have:

$$\mathit{com}_{\mathcal{S}}(\vec{x}_k) \xrightarrow{a(\vec{x}_k),\vec{y}_k} \mathit{com}_{\mathcal{S}}(\vec{x}_{k+1}); \tag{2}$$

$$\mathit{com}_T(\vec{x}) \xrightarrow{a(\vec{x})} \mathit{com}_T(\vec{x}_{k+1}). \tag{3}$$

From (2) and the third condition in CG's transition relation, it follows that $\langle s'_T, \mathit{com}_{\mathcal{S}}(\vec{x}_{k+1})\rangle \in \Sigma$ is a state in CG, and thus $s'_T \preceq \mathit{com}_{\mathcal{S}}(\vec{x}_{k+1})$. Due to the first requirement of ND-simulations, $\mathit{env}(s'_T) = \mathit{env}(\mathit{com}_{\mathcal{S}}(\vec{x}_{k+1}))$ applies. Beacuse $\mathit{env}(\mathit{com}_{\mathcal{S}}(\vec{x}_{k+1})) = \mathit{env}(\mathit{com}_T(\vec{x}_{k+1}))$, we then get that $\mathit{env}(s'_T) = \mathit{env}(\mathit{com}_T(\vec{x}_{k+1}))$. Putting this together with (3) and the fact that the target behavior $\mathcal{B}_T$ is deterministic, we conclude that $s'_T = \mathit{com}_T(\vec{x}_{k+1})$, and as a result, $\mathit{com}_T(\vec{x}_{k+1}) \preceq \mathit{com}_{\mathcal{S}}(\vec{x}_{k+1})$ follows.

So, we have proven that for any f-complaint game play $\eta = \langle \vec{x}_0, \vec{y}_0\rangle\langle \vec{x}_1, \vec{y}_1\rangle \cdots$, it is the case that $\mathit{com}_T(\vec{x}_i) \preceq \mathit{com}_{\mathcal{S}}(\vec{x}_i)$, for all $i \geq 1$. Let us show, by induction on index ℓ, that $\langle \vec{x}_\ell, \vec{y}_\ell\rangle \models \varphi$, for all $\ell \geq 0$. First, $\langle \vec{x}_0, \vec{y}_0\rangle \models \Theta$ holds trivially. Consider game state $\langle \vec{x}_{\ell+1}, \vec{y}_{\ell+1}\rangle$. Because of the above proven fact that $\mathit{com}_T(\vec{x}_{\ell+1}) \preceq \mathit{com}_{\mathcal{S}}(\vec{x}_{\ell+1})$ and requirement 2 of ND-simulations, we get that $\langle \vec{x}_{\ell+1}, \vec{y}_{\ell+1}\rangle \models \mathit{final}_T \rightarrow \mathit{final}_i$, for each $i \in \{1, \ldots, n\}$. Now, if $a(\vec{x}_{\ell+1}) = \sharp$, then $\langle \vec{x}_{\ell+1}, \vec{y}_{\ell+1}\rangle \models \neg \mathit{fail}_i$, for each $i \in$

$\{1,\ldots,n\}$, since $\sharp \neq i$. If, on the other hand, $a(\vec{x}_{\ell+1}) \in \mathcal{A}$, that is, an actual action has been requested in $\vec{x}_{\ell+1}$, then due to the f's definition and the fact that $com_T(\vec{x}_{\ell+1}) \preceq com_{\mathcal{S}}(\vec{x}_{\ell+1})$, we know that $\vec{y}_{\ell+1} \in \omega(\langle com_T(\vec{x}_{\ell+1}), com_{\mathcal{S}}(\vec{x}_{\ell+1})\rangle, a(\vec{x}_{\ell+1}))$. This means that there exists a transition $\langle com_T(\vec{x}_{\ell+1}), com_{\mathcal{S}}(\vec{x}_{\ell+1})\rangle \xrightarrow{a(\vec{x}_{\ell+1}),\vec{y}_{\ell+1}} \sigma'$ in controller generator CG, for some $\sigma' \in \Sigma$. By CG's transition relation definition, there exists a transition $com_{\mathcal{S}}(\vec{x}_{\ell+1}) \xrightarrow{a(\vec{x}_{\ell+1}),\vec{y}_{\ell+1}} s'_{\mathcal{S}}$ in $\mathcal{T}_{\mathcal{S}}$, which—by the notion of enacted system—implies that behavior $\mathcal{B}_{\vec{y}_{\ell+1}}$ can make a transition on action $a(\vec{x}_{\ell+1})$ and $\langle \vec{x}_{\ell+1}, \vec{y}_{\ell+1}\rangle \models \neg fail_{\vec{y}_{\ell+1}}$ holds. (See $\langle \vec{x}_{\ell+1}, \vec{y}_{\ell+1}\rangle \models \neg fail_i$ trivially when $i \neq \vec{y}_{\ell+1}$).

(ONLY-IF PART) Due to Theorem 9 and the fact that $\langle\langle \sharp,\ldots,\sharp\rangle,\sharp\rangle \in W$, there exists a winning strategy f. Using such strategy, we define relation $R \subseteq S_T \times S_{\mathcal{S}}$ as follows: $\langle s_T, s_{\mathcal{S}}\rangle \in R$ iff there exists a game play $\eta = \langle \vec{x}_0, \vec{y}_0\rangle\langle \vec{x}_1, \vec{y}_1\rangle \cdots$ compliant with f and such that $com_T(\vec{x}_\ell) = s_T$ and $com_{\mathcal{S}}(\vec{x}_\ell) = s_{\mathcal{S}}$, for some $\ell \geq 1$.

Let us prove that R is indeed an ND-simulation relation and that $\langle\langle b_{T0}, e_0\rangle, \langle b_{10},\ldots,b_{n0},e_0\rangle\rangle \in R$. The latter claim follows from the fact that $\eta = \langle \vec{x}_0,\vec{y}_0\rangle\langle \vec{x}, f(\langle \vec{x}_0,\vec{y}_0\rangle,\vec{x})\rangle\langle \vec{x}, f(\langle \vec{x}_0,\vec{y}_0\rangle,\vec{x}\vec{x})\rangle\cdots$ where $\vec{x} = \langle b_{10},\ldots,b_{n0},e_0,b_{T0},\sharp\rangle$ is an f-complaint play and $com_T(\vec{x}) = \langle b_{T0},e_0\rangle$ and $com_{\mathcal{S}} = \langle b_{10},\ldots,b_{n0},e_0\rangle$.

To prove that R is an ND-simulation of $\mathcal{T}_T$ by $\mathcal{T}_S$ we are to prove the three requirements of ND-simulations (see page 12). To that end, assume $\langle s_T, s_{\mathcal{S}}\rangle \in R$. By construction of R, requirement 1 is satisfied trivially and there exists a game play $\eta = \langle \vec{x}_0,\vec{y}_0\rangle\langle \vec{x}_1,\vec{y}_1\rangle\cdots$ complaint with f such that $com_T(\vec{x}_k) = s_T$ and $com_{\mathcal{S}}(\vec{x}_k) = s_{\mathcal{S}}$, for some $k \geq 1$ and $a \in \hat{\mathcal{A}}$. Since f is a winning strategy, $\langle \vec{x}_k,\vec{y}_k\rangle \models \varphi$. Also, since $\langle \vec{x}_k,\vec{y}_k\rangle \neq \langle \vec{x}_0,\vec{y}_0\rangle$, $\langle \vec{x}_k,\vec{y}_k\rangle \not\models \Theta$ and hence $\langle \vec{x}_k,\vec{y}_k\rangle \models (final_T \rightarrow \bigwedge_{i=1}^{n} final_i)$, which yields requirement 2. Finally, for the third requirement of ND-simulations, consider a transitions $s_T \xrightarrow{a'} s'_T$ in $\mathcal{T}_T$. First, from conditions 5(b)i and 5(b)iv in G's definition, it follows that there exists an f-complaint game play $\eta' = \langle \vec{x}'_0,\vec{y}'_0\rangle\langle \vec{x}'_1,\vec{y}'_1\rangle\cdots$ such that $\vec{x}_i = \vec{x}'_i$ and $\vec{y}_i = \vec{y}'_i$, for all $i \in \{0,\ldots,k-1\}$, and $com_T(\vec{x}_k) = com_T(\vec{x}'_k) = s_T$ and $com_{\mathcal{S}}(\vec{x}_k) = com_{\mathcal{S}}(\vec{x}'_k) = s_{\mathcal{S}}$—play η' is exactly like η up to game state $\langle \vec{x}_k,\vec{y}_k\rangle$, except that $\vec{x}'_k$ may (possibly) encode a different requested action. In addition, due to conditions 5(b)iv and 5(b)i in G's definition, we can assume that η' is such that $com_T(\vec{x}_{k+1}) = s'_T$. Because $\langle \vec{x}'_k,\vec{y}'_k\rangle \models \neg fail_{\vec{y}'_k}$, f is winning, and η' is complaint with f, it follows—from conditions 7, 5(b)ii, 5(b)iii, and 5(b)iv in G's definition—that $com_{\mathcal{S}}(\vec{x}'_k) \xrightarrow{a(\vec{x}'_k),\vec{y}'_k} com_{\mathcal{S}}(\vec{x}'_{k+1})$, and therefore condition 3a of ND-simulations holds true. Finally, consider any $s_{\mathcal{S}} \xrightarrow{a'} s'_{\mathcal{S}}$ in $\mathcal{T}_{\mathcal{S}}$ with $env(s'_{\mathcal{S}}) = env(s'_T)$. Again, since every possible evolution of the enacted system is accounted by some successor game states (Lemma 11), we can assume that η' is such that $com_{\mathcal{S}}(\vec{x}'_{k+1}) = s'_{\mathcal{S}}$. Thus, by R's definition (see η' is complaint with f too), it follows that $R(s'_T, s'_{\mathcal{S}})$ and condition 3b of ND-simulation applies. □

As a straightforward consequence of this result and Theorem 1, we have that □-GS G is winning (i.e., $\langle\langle \sharp,\ldots,\sharp\rangle,\sharp\rangle \in W$) if and only if there exists a composition of the target in the system (i.e., $\langle b_{T0},e_0\rangle \preceq \langle b_{10},\ldots,b_{n0},e_0\rangle$).

In addition, addition to this, the following result holds, which gives us an actual procedure to build a controller generator and, hence, all possible compositions.

Theorem 13. *Let $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$ be a system and $\mathcal{B}_T$ a target behavior over $\mathcal{E}$. Let $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_s, \rho_c, \Box\varphi \rangle$ be the corresponding $\Box$-GS derived as above, for $\mathcal{B}_T$ and $\mathcal{S}$, and let W be the maximal set of winning states such that $\langle\langle \sharp, \ldots, \sharp \rangle, \sharp \rangle \in W$. Then, $CG = \langle \Sigma, \mathcal{A}, \{1, \ldots, n\}, \partial, \omega \rangle$ is the controller generator of $\mathcal{S}$ for $\mathcal{B}_T$, where:*

- $\Sigma \subseteq S_T \times S_{\mathcal{S}}$ *is such that* $\langle \langle b_T, e_T \rangle, \langle b_1, \ldots, b_n, e_{\mathcal{S}} \rangle \rangle \in \Sigma$ *iff* $e_T = e_{\mathcal{S}}$ *and there exists* $\langle \langle b_1, \ldots, b_n, e_T, b_T, a \rangle, ind \rangle \in W$, *for some* $a \in \mathcal{A}$ *and* $ind \in \{1, \ldots, n\}$.
- $\partial \subseteq \Sigma \times \mathcal{A} \times \{1, \ldots, n\} \times \Sigma$ *is such that* $\langle \sigma, a, k, \sigma' \rangle \in \partial$, *where* $\sigma = \langle \langle b_T, e \rangle, \langle b_1, \ldots, b_n, e \rangle \rangle$ *and* $\sigma' = \langle \langle b'_T, e' \rangle, \langle b'_1, \ldots, b'_n \rangle, e' \rangle$, *if and only if*
  - $\langle \langle b_1, \ldots, b_n, e, b_T, a \rangle, k \rangle \in W$; *and*
  - $\rho_s(\langle b_1, \ldots, b_n, e, b_T, a \rangle, k, \langle b'_1, \ldots, b'_n, e', b'_T, a' \rangle)$, *for some* $a' \in \mathcal{A} \cup \{\sharp\}$.
- $\omega(\sigma, a) = \{k \mid \exists\, \sigma' \in \Sigma$ *s.t.* $\sigma \xrightarrow{a,k} \sigma'$ *is in CG*$\}$.

Proof. We show that CG satisfies all requirements of a controller generator (see Section 3; page 17). First, by Theorem 12, $\langle \langle b_T, e_T \rangle, \langle b_1, \ldots, b_n, e_{\mathcal{S}} \rangle \rangle \in \Sigma$ iff $\langle b_T, e_T \rangle \preceq \langle b_1, \ldots, b_n, e_{\mathcal{S}} \rangle$. Second, suppose that $\langle \sigma, a, k, \sigma' \rangle \in \partial$, as defined above. From $\langle \langle b_1, \ldots, b_n, e, b_T, a \rangle, k, \langle b'_1, \ldots, b'_n, e', b'_T, a' \rangle \rangle \in \rho_c$, together with the fact that $a \neq \sharp$ and $ind \neq \sharp$, it follows that $com_T(\sigma) \xrightarrow{a} com_T(\sigma')$ in $\mathcal{T}_T$. Also, since $\langle \langle b_1, \ldots, b_n, e, b_T, a \rangle, k \rangle \in W$, we know that $\langle \langle b_1, \ldots, b_n, e, b_T, a \rangle, k \rangle \models \neg fail_k$, and therefore, $com_{\mathcal{S}}(\sigma) \xrightarrow{a,k} com_{\mathcal{S}}(\sigma')$ in $\mathcal{T}_{\mathcal{S}}$. Now consider any transition $com_{\mathcal{S}}(\sigma) \xrightarrow{a,k} s''_{\mathcal{S}} = \langle b''_1, \ldots, b''_n, e' \rangle$ in $\mathcal{T}_{\mathcal{S}}$. Due to Lemma 11, game state $\langle \langle b_1, \ldots, b_n, e, b_T, a \rangle, k \rangle$ ought to have a successor state of the form $\langle \langle b''_1, \ldots, b''_n, e', b'_T, a'' \rangle, k'' \rangle$. Moreover, since the latter state is in the winning set, there is at least one k'' such that $\langle \langle b''_1, \ldots, b''_n, e', b'_T, a'' \rangle, k'' \rangle \in W$, and thus $\langle \langle b'_T, e' \rangle, \langle b''_1, \ldots, b''_n, e' \rangle \rangle \in \Sigma$ follows. By following the same reasoning, but backwards, it is straightforward to prove that if the three transition requirements in a controller generator are satisfied, then such transition will be accounted by the above ∂ relation. So, the above transition relation ∂ coincides with that of the controller generator. Finally, function ω above coincides with the controller generator's output function by definition. □

The above theorems show how one can exploit tools from reactive system synthesis for computing all compositions of a given target behavior. In details, starting from $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$ and $\mathcal{B}_T$, one can build the corresponding game structure G, then compute the winning set W, and, if it contains G's initial state, use W to generate the controller generator. In fact, this last step is not really needed. It is not hard to see that given a system state $\langle b_1, \ldots, b_n, e, b_T, a \rangle$ (including action $a \in \mathcal{A}$ to be executed next), a behavior selection *ind* is "good" (i.e, the selected behavior can actually execute the action and the whole system can still ND-simulate the target behavior) if and only if W contains a tuple $\langle \langle b_1, \ldots, b_n, e, b_T, a \rangle, ind \rangle$. Consequently, at each step, based on (current) target behavior state b_T, available behaviors' states $b_1, \ldots, b_n$, environment state e, and requested action a, one can select a tuple from W, extract its *ind* component, and use it to select the next behavior.

Finally, note that the time complexity of Algorithm 3 is polynomial in $|V|$, the size of the input $\Box$-GS state space. Since, in our encoding, $|V|$ is polynomial in $|B_1|, \ldots, |B_n|, |B_T|, |E|$, and $|\mathcal{A}|$, and exponential in n, we get the following result:

Theorem 14. *Let $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$ be a system and $\mathcal{B}_T$ a target behavior over $\mathcal{E}$. Checking the existence of compositions by reduction to safety games can be done in polynomial time wrt $|B_1|, \ldots, |B_n|, |B_T|, |E|$, and $|\mathcal{A}|$, and exponential time in n.*

Such a result says that computing a composition using safety games has the same computational complexity as computing the ND-simulation relation for solving behavior composition problems (cf. Theorem 2). Since the composition problem is EXPTIME-hard [59], the technique based on safety games is actually optimal with respect to worst-case time complexity.

6. Implementing Behavior Composition in TLV

With the behavior composition problem formally reduced to that of synthesizing a winning strategy in a special safety-game, one can appeal to existing implemented systems that are capable of searching for winning strategies in game structures, such as TLV [69], ANZU [43], LILY [42], and MOCHA [4]. Although we shall focus on TLV, all basic concepts discussed here remain valid for all other tools.

TLV (Temporal Logic Verifier) is a (generic) software for verification and synthesis of LTL specifications, which exploits Binary Decision Diagrams (BDDs) for symbolic state manipulation, in order to contain state explosion. Generally speaking, TLV takes two inputs: *(i)* a synthesis procedure; and *(ii)* an LTL specification, encoded in SMV language [57], to be processed by the input procedure. In particular, for *(i)*, we consider a specific procedure for dealing with safety games and refer to the so-obtained system as TLV$_\Box$.[11] Essentially, TLV$_\Box$ takes as input an LTL specification encoding a $\Box$-GS and derives from the game's maximal winning set, if non empty, a structure representing the controller generator, as shown in Theorem 13. We refer to [69] for further details on TLV and the input language SMV, here introducing some essentials only.

Our approach consists in: *(i)* building, as described in Section 5.2, the $\Box$-GS corresponding to a given behavior composition problem; *(ii)* deriving the SMV encoding for the obtained $\Box$-GS; and *(iii)* executing the encoding in TLV$_\Box$, to both check whether the composition problem is solvable and, if so, compute the controller generator. Next, we detail *(ii)*.

In the SMV encoding, every aspect of a $\Box$-GS, e.g., the available behaviors or the controller, is modelled as a so-called "module." Figure 6 shows the basic blocks of the encoding for our painting world running example (see Figure 1; page 6). Modules, e.g., `ArmSys`, can be built from submodules, by declaring these in the `VAR` section, which is what we actually do in our construction. When doing so, according to the SMV semantics, the execution of the composite module corresponds to the synchronous execution of its submodules. In our encoding, however, asynchrony is often needed,

[11] This specific procedure for safety games was originally coded by Amir Pnueli.

```
MODULE System(a1op, a2op, a3op)
VAR
  asys : ArmSys(client.req,a1op,a2op,a3op);
  client : Client(asys.envstate);
DEFINE
  initial:= asys.initial & client.initial;
  failfinal :=  client.final & !asys.final;
  failure := asys.fail | failfinal;
  req := client.req;

MODULE ArmSys(req, a1op, a2op, a3op)
VAR
  env : Environment(req);
  a1 : ArmA(a1op, env.state);
  a2 : ArmB(a2op, env.state);
  a3 : ArmC(a3op, env.state);
DEFINE
  initial := env.initial &
     a1.initial & a2.initial & a3.initial;
  fail := a1.fail | a2.fail | a3.fail;
  final := s1.final & s2.final & s3.final;
  envstate := env.state;

MODULE Client(env)
VAR
  target : Target(env, req);
  req: {start,none,prepare,clean,...};
INIT
 req = start
TRANS
 case
  next(target.state) = t2 :
             next(req) in {paint,clean};
  ...
  TRUE : next(req) = none;
 esac
DEFINE
  ...
```

```
MODULE Controller(req)
VAR
 a1op: {start,none,prepare,
         clean,paint,dispose,recharge};
 a2op: {start,none,prepare,
         clean,paint,dispose,recharge};
 a3op: {start,none,prepare,
         clean,paint,dispose,recharge};
INIT
 a1op = start & a2op = start &
        a3op = start
TRANS
  !initial ->
 (
   -- start action only at initially
   (a1op != start) & (a2op != start) &
   (a3op != start)
 &
   -- Some behavior does the req. action
   (req = a1op | req = a2op | req = a3op)
 & -- Behaviors do actions requested
   (a1op != none -> a1op = req) &
   (a2op != none -> a2op = req) &
   (a3op != none -> a3op = req)
 & -- One behavior acts at a time
   (a1op != none ->
            a2op = none & a3op = none) &
   (a2op != none ->
            a1op = none & a3op = none) &
   (a3op != none ->
            a1op = none & a2op = none)
 )
DEFINE
 initial := a1op= start & a2op = start
           & a3op = start;
```

```
MODULE main
VAR
 sys: system System(contr.a1op,contr.a2op,contr.a3op);
 contr: system Controller(sys.req);
DEFINE
 good := (contr.initial & sys.initial) | !(sys.failure);
```

Figure 6: A TLV sample fragment encoding.

which we emulate by forcing each submodule that represents an available behavior to loop at each step, unless "selected."

Module `Main`, consisting of submodules `sys` and `contr`, wraps all the other modules, and represents the whole game structure. In particular, module `sys` captures the system player, by encoding the enacted system behavior (`asys`) together with the enacted target behavior (`client`), i.e., informally, the external *uncontrollable* system. Module `contr`, on the other hand, encodes the constraints on the controller player in the game structure, that is, the module to be synthesized. Finally, variable `good` encodes the goal invariant property to be respected, which states that a game state (including both player states) is "good" if and only if either both players are at their dummy initial states or the external system—the system player—has not been brought into a failure state. The external system may reach a failure state, for instance, if an available behavior is requested an action it cannot perform in its current state, or if the

target behavior is in a final state but some available behavior does not.

Modules `sys` and `contr` are meant to evolve *synchronously*, the former choosing the next requested action to be performed and the latter selecting the available behavior for its execution. Consequently, the requested action (`sys.req`) is passed as an input argument to the `contr` module, and the chosen available behavior is passed as an input to the `sys` module. Notice that instead of merely returning just the index of the available behavior meant to execute the currently requested action (as in the game structure previously defined), the `contr` module outputs one action per available behavior—e.g., `a2op` denotes the action assigned to behavior arm `a2`, using the distinguished action constant `none` to state that no action is requested. This approach enables the encoding of settings were more than one behavior may execute at the same time, like in [77]. We refer to this encoding as, while being clearly more general, it introduces no additional difficulty.

Next, we detail the submodules representing the two players of the game structure. As for `contr`, which is an instance of `Controller`, the transition relation defined by the constraints in the `INIT` and `TRANS` sections encodes an *unconstrained controller*, which assigns, at each step, one action to each available behavior, by assigning values to the state variables `a1op`, `a2op`, and `a3op`. *The synthesis goal is to restrict such a relation so as to obtain a winning strategy.* In particular, the constraints enforced on the controller player's state are as follows. According to the `INIT` section, in its initial state (where variable `initial` holds true) the controller must instruct every behavior to initialize itself by performing the dummy action `start` (all behaviors initialize simultaneously). As for non-initial states, the `TRANS` section defines the following constraints: *(i)* no initialization action can be assigned to any behavior; *(ii)* the current action request must match at least one of the behavior actions; *(iii)* a behavior can be instructed to execute an action only if that action is the one currently requested; and *(iv)* at most one behavior can be instructed to act at a time.

Concerning module `sys`, which is an instance of `System`, it essentially captures, as said above, all the aspects of the system player. Precisely, `sys` is the synchronous product of the enacted available system (submodule `asys`) and the client issuing the action requests (submodule `client`). On the one hand, submodule `asys` accounts for the available behaviors running in the environment, according to both the currently requested action (variable `req`) and the controller assignment to variables `a1op`, `a2op`, and `a3op`; on the other hand, submodule `client` provides, at every game state, the requested action (variable `req`), which is, of course, required to be compliant with the target behavior. Observe that `client` requests action `none` (last rule) only when no other legal action can be requested anymore. Since the execution of `none` yields no change in the current game state, it turns out that once executed, `none` remains the only action available to the target, from that point on.

Distinguished abbreviations are used to define, in the `DEFINE` section, initial, final, and failure states. In particular, the enacted system behavior (`ArmSys`) fails (`failure`) when any of the available behaviors does, an available behavior failing when instructed to perform an action it cannot execute, depending on its and the environment's current state. Avoiding such situations, by properly constraining `sys`'s transition relation, is exactly the synthesis procedure's aim. Clearly, the only way to achieve this is by suitably assigning `sys`'s *controllable* input variables `a1op`, `a2op`,

and a3op, that is, ultimately, by suitably "crafting" the contr module (while respecting its constraints). Finally, the whole enacted system does not respect the final-state condition (failfinal) when the client is in a state where it may legally terminate its execution but the available system does not.

We encoded our running example for TLV$_\Box$ and run it so as to compute the corresponding winning set, along with the controller generator. The result obtained was an automaton with 16 states and 21 transitions, from where controllers can be easily extracted. We report three sample states of the automaton:

```
State 3
sys.asys.env.state = e2,            sys.asys.a1.state = a1,
sys.asys.a2.state = b2,             sys.asys.a3.state = c1,
sys.client.target.state = t2,       sys.client.req = paint,
contr.a1op = none,  contr.a2op = paint, contr.a3op = none,

State 15
sys.asys.env.state = e2,            sys.asys.a1.state = a1,
sys.asys.a2.state = b3,             sys.asys.a3.state = c1,
sys.client.target.state = t4,       sys.client.req = dispose,
contr.a1op = dispose, contr.a2op = none,  contr.a3op = none,

State 16
sys.asys.env.state = e2,            sys.asys.a1.state = a1,
sys.asys.a2.state = b1,             sys.asys.a3.state = c1,
sys.client.target.state = t4,       sys.client.req = dispose,
contr.a1op = dispose, contr.a2op = none,  contr.a3op = none,
```

In state 3, for instance, the environment is in state e_2, the available arms are in states a_1, b_2, and c_1, the target behavior is in state t_2, the action requested next is *paint*, and the controller has selected arm $\mathcal{B}_2$ for carrying out the action. States 15 and 16 are the possible successor states that the game can be in, depending on how the non-deterministic transition in behavior $\mathcal{B}_2$ turns out.

The complete TLV specification for our example can be found in Appendix A.

7. Related Work

The framework developed in this paper can be seen as a *core* account for behavior composition, and can be extended in a number of directions. In [77], a *distributed* version of the problem is presented, where instead of a central entity that embodies the controller, a set of *local* controllers, one per available behavior, are meant to jointly realize the target behavior, by exploiting an underlying, shared communication channel. Another extension involves realizing not one but several target behaviors concurrently, using the same available system [75]. Composition under *partial* observability was also explored by De Giacomo et al. [24], whereas composition with *data exchange* was investigated by Berardi et al. [12] in the context of web-services. Finally, [76, 27] propose two frameworks (and corresponding techniques) for composing agent high-level programs. The techniques for all these extensions vary, from PDL satisfiability

([77, 12]) to LTL/ATL synthesis ([24, 75, 27]), to computation of specific fix-points ([76]). Also, a direct *search-based* technique for the core composition account was recently proposed by Stroeder and Pagnucco [83], which could turn out to be promising when it comes to applying heuristics.

The composition technique we proposed here is related to synthesis of reactive systems from LTL temporal specifications [68, 67, 45], which is proven 2EXPTIME-complete, in general [68]. In our particular case, however, we can restrict to a class of specifications, namely GR(1), for which the problem is EXPTIME-complete [67]. Though a subclass of full LTL, GR(1) type formulas are expressive enough to deal with many, if not most, realistic applications. They, for instance, have been used to support advanced forms of path planning in robots [47, 46, 11, 31]. Notably, a work that is inspired by our behavior composition is that of Lustig and Vardi [52], where the problem of synthesizing LTL specifications by coordinating given modules from an existing library is studied (and proven 2EXPTIME-complete). In turn, De Giacomo and Felli [25] showed how to solve the behavior composition problem by ATL model checking. ATL (Alternating-time Temporal Logic) [3] is a logic especially aimed for reasoning about multi-player games, where players can form coalitions to satisfy certain formulae. The result is important in that it gives access to some of the state-of-the-art model checking techniques and tools, such as MCMAS,[12] that have been recently developed within the agent community. Since the behavior composition task can be seen as winning a special kind of game (see Section 5), it would be interesting to explore whether the heuristic-based techniques developed in the context of General-Game Playing [33] can be applied for "playing" composition games that are either too difficult to solve at the outset or directly unsolvable.

Our work directly relates to several others (e.g., [36, 19, 13, 30, 12, 15, 35, 71]) on Service Oriented Computing (SOC) [2]. Indeed, available behaviors, ultimately transition systems, can be seen as the conceptual model for *conversational*, or stateful, (web) services. By taking this perspective, many results presented here become applicable, almost *off-the-shelf*, in the SOC area. One line of research that is quite related to our is that reported in [63, 64, 66, 16] which exploits techniques for conditional planning for temporally extended goals. Starting from a set of conversational available services, specified in BPEL4WS (Business Process Execution Language for Web Services), and a goal specified as a branching temporal formulae (in the language EAGLE, a suitable extension of CTL [22]), conditional planning techniques are exploited to find an interleaved execution of available services, so as to satisfy the desired goal. Roughly speaking, a goal represents a main, finite, desired path of states, plus some secondary paths to be followed when "exceptions" (i.e., deviation from the main path) arise. This technique, actually implemented in the system ASTRO[13] based on the Model Based Planner (MBP 3) [21], exploits Model Checking technology (ultimately, BDDs) to control the state space explosion. Two main features differentiate such work from ours. Firstly, our goals are actually new services (behaviors), rather than desired executions, which, once realized, can be executed as any other one. What is more, the

[12] http://www-lai.doc.ic.ac.uk/mcmas/
[13] http://astroproject.org

behaviors we synthesize are really intended to interact with some executor, instead of executing on their own, like plans do. So, from a high-level perspective, we aim at extending the set of services offered by a given system, whereas the work above focuses more on serving particular requests by taking advantage of the existing system. A research line on services that adopts the same approach as ours is that in [8, 9, 10]. Like ours, these works rely on techniques borrowed from controller synthesis, though the approach therein is more theoretical. In contrast, we fully take advantage of such results for practical reasons, by *(i)* exploiting controller synthesis techniques to build *flexible* solutions, and *(ii)* by showing how to use the actual existing technology, based on a symbolic approach, for effective solution construction.

In the series of works [55, 56, 81], the Situation Calculus logical framework is adopted as a theoretical framework for composing *semantic* Web services (specified in the OWL-S process ontology [54]). Available and goal services are modeled as (complex) GOLOG programs, and the objective is to find a terminating execution of the available services that corresponds to an execution of the goal service. Based on the same Situation Calculus semantics, Sirin et al. [80] exploits Hierarchical Task Networks (HTN) to model available (OWL-S) services, and then uses an HTN planner [60] to build a plan representing an actual, finite, execution of a desired target service. All such works share the idea of *achieving* a desired goal—being it a state or a situation—by executing a *terminating* plan or program. Our approach is different, and, in a sense, more general, essentially due to two major differences: first, we consider realization of *infinite* target behavior executions; second, a solution to our composition problem is required to realize *all* possible behavior service executions, rather than just one.

Behavior composition is also related to several forms of automated planning in AI, in particular, to planning for temporally extended goals (as mentioned above in the context of services), which investigate techniques for building finite or infinite plans that satisfy linear- or branching-time specifications [7, 65, 44]. Indeed, our problem requires an advanced conditional plan (with loops) that always guarantees all possible target requests to be served, which is, ultimately, a (temporal) invariant property. More specifically, the solutions obtained via the simulation technique developed in this work are akin to the so-called *universal plans* [79], i.e., plans representing every possible solution. A further recent work about planning, where temporal fairness constraints are explicitly stated so as to capture long-term effects of action executions is [28]. We conjecture that some of the concepts there can be exploited in our context to sophisticate the notion of behaviors to be composed.

Composing behaviors can also be linked to (multi-)agent systems in natural ways. For instance, a Belief-Desire-Intention agent operates on the coordinated execution of pre-defined non-deterministic plans—the available behaviors—in order to achieve its goals [73, 34]. One could then imagine composing such available plans so as to bring about another non-available plan—the target behavior—that represents all the goals of the agent. Similarly, composing behaviors can be seen as realizing a "team-oriented" behavior (e.g., a RoboCup sophisticated abstract "team" player), represented by the target behavior, from the behavior of single agents (e.g., a set of actual RoboCup robotic players with different capabilities), represented by the various available behaviors. Of course, the core composition framework as presented here lacks, so far, convenient features for programming team agent systems [70, 41], such as roles, holons, commu-

nication channels, etc.

Finally, behavior composition, as studied in this paper, is tightly related to the problem of integrating simple functionalities to implement advanced (intelligent) behaviors in the context of robot-ecologies [74, 18, 17]. The idea of leveraging on the capabilities of many simple robotic devices (e.g., vacuum cleaners, blinds, cameras, robot arms, etc.) in order to achieve complex tasks has attracted much attention lately given the marked tendency toward the embedding of intelligent, networked robotic devices in our homes and offices. While very close in "spirit," the work done in robot ecologies so far focuses on differs aspects. Most of the work in "composing" functionalities within an ecology of robots is devoted to the generation of adequate ways of *connecting* existing functionalities via so-called *configurations* in order to be able to carry a particular task, such as making the output of a video camera the input of a moving robot lacking visual capabilities. Instead of dealing explicitly with such connectivity issues (except for the interaction with the environment), our work focuses on how each component needs to be actually operated in order to achieve the target process. Also, the integration of functionalities is either done fully by hand (e.g., [74, 18]) or semi-automatically through hand-tailored planning techniques (e.g., [50, 51]) in the style of HTN planning. In the latter case, one is meant to define standard "recipes" to describe ways to combine functionalities for specific purposes. Our approach is more of a first-principle one, no domain information is available on how available behaviors can or should be combined. More importantly, while we took a high-level perspective on agents and shared devices, and focused on the synthesis problem only, the aforementioned work on robot ecologies deals better with many other practical aspects of concern when it comes to implementing the solution. For instance, how to design such devices so that they can easily interoperate among themselves, as we assume here, and how such interoperability is actually realized, via an appropriate middleware [17]. In fact, we expect a fruitful cross-fertilization between the theoretical studies on automated synthesis of agents, as the one in the present paper, and practical work on experimenting device integration in robot ecologies and ambient intelligence.

8. Conclusions

In this paper, we have carried out a deep investigation on the behavior composition problem, that is, the problem of realizing a desired, but non-available, target behavior by reusing and re-purposing accessible modules (devices, agents, plans, etc.), which are the only behaviors actually available. In particular, we have proposed a technique, based on the notion of *simulation*, for building a controller that coordinates the concurrent executions of the available behaviors so as to "mimic" the target behavior.

This work lays the basis for several further developments, some of which have already been mentioned in the related work section. We would like to close the paper by briefly discussing two of them that still require further study. The first one concerns the possibility of interchanging actions. More precisely, in this work we have implicitly assumed that two actions are equivalent if and only if they are named the same way, and hence, they are exactly the same action. Clearly, there are situations requiring a more flexible model, for instance when the domain includes actions with different names that execute, in fact, the same task; or where some actions specialize some

other, more abstract, ones. For example, actions *paint-red* and *paint-blue* may stand for specializations (or implementations) of the more abstract, and maybe not even directly available, action *paint*. Both concrete actions, when abstracting from other details, may be considered equivalent in terms of the effect of having an object painted. One natural way to generalize the composition framework developed in this paper is to assume the existence of an underlying *compatibility relation* $\ll \subseteq \mathcal{A} \times \mathcal{A}$ among actions: if $a \ll \hat{a}$ (i.e., action $\hat{a}$ is compatible with action a), then an execution of action a can be satisfied by the actual execution of action $\hat{a}$. With a domain compatibility relation at hand, one can then generalize the notion of ND-simulation from Section 3 to account for the fact that whenever an action a is requested by the target (e.g., *paint*), a compatible action $\hat{a}$, i.e., $a \ll \hat{a}$, can be carried out by some available behavior (e.g., *paint-red*). We expect all results presented here to still hold in such a generalized case, though further work is needed in order to formalize this intuition. While above we do not make any assumption on relation $\ll$, in practice it may be natural to assume that it satisfies certain properties. For instance, a reflexive compatibility relation captures the fact that every action can be always replaced by itself; a partial order captures a *hierarchy* of actions, where a general action a can be replaced by a more specific one, but not viceversa; and finally an equivalence relation can be used to assert that some actions carry out the very same task (relative to some features of interest). A further study of which properties of relation $\ll$ in specific applications is certainly of interest.

The second direction for further study stems from the observation that, when no compositions exist, it may be of interest to approximate "solutions." That is, if a composition does not exists, one may be interested in understanding which part of the target cannot be realized and which can. Some compelling contribution in this direction can be found in the area of supervisory control of deterministic discrete event systems [72]. In particular, there is a foundational result of great interest: given a specification of the allowed behavior in terms of a language, i.e., a possibly infinite set of runs that are deemed as "allowed," it is always possible to find a single maximal subset of such runs that can be obtained by controlling a given system, the so called "supremal controllable sublanguage" [88]. It would be quite interesting to understand if, at least in certain cases, an analogous property holds for behavior composition as well. The question then is what an "optimal" controller amounts to. Besides some domain-independent criteria (e.g., number of transitions realized), allowing the specification of additional domain information could help define what such best controllers are, such as quantifying all or some non-deterministic transitions and specifying preferences over target actions or available behaviors. A first step toward such direction is taken in the recent work of Yadav and Sardina [89], which proposes a decision-theoretic version of the composition problem studied in this article.

References

[1] Abadi, M., Lamport, L., Wolper, P., 1989. Realizable and unrealizable specifications of reactive systems. In: Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP). pp. 1–17. 2

[2] Alonso, G., Casati, F., Kuno, H., Machiraju, V., 2004. Web Services. Concepts, Architectures and Applications. Springer. 2, 44

[3] Alur, R., Henzinger, T. A., Kupferman, O., 2002. Alternating-time Temporal Logic. Journal of the ACM 49 (5), 672–713. 2, 3, 44

[4] Alur, R., Henzinger, T. A., Mang, F. Y. C., Qadeer, S., Rajamani, S. K., Tasiran, S., 1998. MOCHA: Modularity in model checking. In: Proceedings of the International Conference Computer Aided Verification (CAV). pp. 521–525. 29, 40

[5] Asarin, E., Maler, O., Pnueli, A., 1995. Symbolic controller synthesis for discrete and timed systems. In: Antsaklis, P., Kohn, W., Nerode, A., Sastry, S. (Eds.), Hybrid Systems II. Vol. 999 of LNCS. Springer, pp. 1–20. 29, 31

[6] Asarin, E., Maler, O., Pnueli, A., Sifakis, J., 1998. Controller Synthesis for Timed Automata. In: IFAC Symposium on System Structure and Control. Elsevier Science Publishers Ltd., pp. 469–474. 29

[7] Bacchus, F., Kabanza, F., 1998. Planning for temporally extended goals. Annals of Mathematics and Artificial Intelligence 22 (1-2), 5–27. 45

[8] Balbiani, P., Cheikh, F., Feuillade, G., 2008. Composition of interactive web services based on controller synthesis. In: Proceedings of the IEEE Congress on Services (SERVICES). pp. 521–528. 45

[9] Balbiani, P., Cheikh, F., Feuillade, G., 2009. Algorithms and complexity of automata synthesis by asynhcronous orchestration with applications to web services composition. Electronic Notes in Theoretical Computer Science (ENTCS) 229 (3), 3–18. 45

[10] Balbiani, P., Cheikh, F., Feuillade, G., 2010. Controller/orchestrator synthesis via filtration. Electronic Notes in Theoretical Computer Science (ENTCS) 262, 33–48. 45

[11] Belta, C., Bicchi, A., Egerstedt, M., Frazzoli, E., Klavins, E., Pappas, G. J., Mar. 2007. Symbolic planning and control of robot motion: State of the art and grand challenges. IEEE Robotics and Automation Magazine 14 (1), 61–70. 44

[12] Berardi, D., Calvanese, D., De Giacomo, G., Hull, R., Mecella, M., 2005. Automatic Composition of Transition-based Semantic Web Services with Messaging. In: Proceedings of the International Conference on Very Large Databases (VLDB). pp. 613–624. 43, 44

[13] Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M., 2003. Automatic composition of e-Services that export their behavior. In: Proceedings of the International Joint Conference on Service Oriented Computing (ICSOC). pp. 43–58. 2, 13, 44

[14] Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M., 2005. Automatic service composition based on behavioural descriptions. International Journal of Cooperative Information Systems 14 (4), 333–376. 2, 21

[15] Berardi, D., Cheikh, F., De Giacomo, G., Patrizi, F., 2008. Automatic service composition via simulation. International Journal of Foundations of Computer Science 19 (2), 429–451. 2, 11, 44

[16] Bertoli, P., Pistore, M., Traverso, P., 2010. Automated composition of web services via planning in asynchronous domains. Artificial Intelligence Journal 174 (3-4), 316–361. 2, 44

[17] Bordignon, M., Rashid, J., Broxvall, M., Saffiotti, A., 2007. Seamless integration of robots and tiny embedded devices in a PEIS-ecology. In: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 3101–3106. 2, 46

[18] Broxvall, M., Gritti, M., Saffiotti, A., Seo, B.-S., Cho, Y.-J., 2006. PEIS ecology: Integrating robots into smart environments. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA). pp. 212–218. 46

[19] Bultan, T., Fu, X., Hull, R., Su, J., 2003. Conversation specification: a new approach to design and analysis of e-service composition. In: Proceedings of the International Conference on World Wide Web (WWW). pp. 403–410. 44

[20] Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M., Patrizi, F., 2008. Automatic service composition and synthesis: The roman model. IEEE Data Engineering Bulletin 31 (3), 18–22. 2

[21] Cimatti, A., Pistore, M., Roveri, M., Traverso, P., 2003. Weak, strong, and strong cyclic planning via symbolic model checking. Artificial Intelligence Journal 147 (1-2), 35–84. 44

[22] Clarke, E., Emerson, E., 1982. Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (Ed.), Logics of Programs. Vol. 131 of LNCS. Springer, Berlin/Heidelberg, Ch. 5, pp. 52–71. 44

[23] Clarke, E. M., Grumberg, O., Peled, D., 1999. Model Checking. The MIT Press. 2, 29

[24] De Giacomo, G., De Masellis, R., Patrizi, F., 2009. Composition of partially observable services exporting their behaviour. In: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS). pp. 90–97. 43, 44

[25] De Giacomo, G., Felli, P., 2010. Agent composition synthesis based on ATL. In: Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS). pp. 499–506. 44

[26] De Giacomo, G., Patrizi, F., Felli, P., Sardina, S., 2010. Two-player game structures for generalized planning and agent composition. In: Proceedings of the National Conference on Artificial Intelligence (AAAI). pp. 297–302. 3

[27] De Giacomo, G., Patrizi, F., Sardina, S., May 2010. Agent programming via planning programs. In: Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS). pp. 491–498. 43, 44

[28] De Giacomo, G., Patrizi, F., Sardina, S., 2010. Generalized planning with loops under strong fairness constraints. In: Proceedings of Principles of Knowledge Representation and Reasoning (KR). pp. 351–361. 45

[29] De Giacomo, G., Sardina, S., 2007. Automatic Synthesis of New Behaviors from a Library of Available Behaviors. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI). pp. 1866–1871. 1, 2, 16, 21

[30] Deutsch, A., Sui, L., Vianu, V., 2007. Specification and verification of data-driven web applications. Journal of Computer and System Sciences 73 (3), 442–474. 44

[31] Fainekos, G. E., Girard, A., Kress-Gazit, H., Pappas, G. J., 2009. Temporal logic motion planning for dynamic robots. Automatica 45 (2), 343–352. 44

[32] Gelfond, M., Lifschitz, V., 1998. Action languages. Electronic Transactions of AI (ETAI) 2, 193–210. 4

[33] Genesereth, M., Love, N., 2005. General game playing: Overview of the AAAI competition. AI Magazine 26, 62–72. 44

[34] Georgeff, M. P., Lansky, A. L., 1987. Reactive Reasoning and Planning. In: Proceedings of the National Conference on Artificial Intelligence (AAAI). pp. 677–682. 2, 45

[35] Gerede, C. E., Hull, R., Ibarra, O. H., Su, J., 2004. Automated composition of e-services: Lookaheads. In: Proceedings of the International Joint Conference on Service Oriented Computing (ICSOC). pp. 252–262. 44

[36] Gerede, C. E., Ibarra, O. H., Ravikumar, B., Su, J., 2005. Online and minimum-cost ad hoc delegation in e-service composition. In: Proceedings of the IEEE International Conference on Services Computing (SCC). pp. 103–112. 44

[37] Ghallab, M., Nau, D., Traverso, P., 2004. Automated Planning: Theory and Practice. Morgan Kauffman. 10

[38] Harding, A., Ryan, M., Schobbens, P.-Y., 2005. A New Algorithm for Strategy Synthesis in LTL Games. In: Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 477–492. 3

[39] Henzinger, M. R., Henzinger, T. A., Kopke, P. W., 1995. Computing simulations on finite and infinite graphs. In: Procedings of the Annual Symposium on Foundations of Computer Science (FOCS). pp. 453–462. 3, 11

[40] Hull, R., 2005. Web services composition: A story of models, automata, and logics. In: Proceedings of the IEEE International Conference on Services Computing (SCC). pp. 18–19. 2

[41] Jarvis, B., Jarvis, D., Jain, L., 2007. Teams in multi-agent systems. In: Shi, Z., Shimohara, K., Feng, D. (Eds.), Intelligent Information Processing III. Vol. 228 of IFIP International Federation for Information Processing. Springer, Ch. 1, pp. 1–10. 45

[42] Jobstmann, B., Bloem, R., 2006. Optimizations for LTL synthesis. In: Proceedings of the Formal Methods in Computer Aided Design (FMCAD). IEEE Computer Society Press, pp. 117–124. 29, 40

[43] Jobstmann, B., Galler, S., Weiglhofer, M., Bloem, R., 2007. Anzu: A tool for property synthesis. In: Proceedings of the International Conference Computer Aided Verification (CAV). pp. 258–262. 29, 40

[44] Kabanza, F., Thiébaux, S., 2005. Search Control in Planning for Temporally Extended Goals. In: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS). pp. 130–139. 45

[45] Kesten, Y., Piterman, N., Pnueli, A., Jul. 2005. Bridging the gap between fair simulation and trace inclusion. Journal Information and Computation 200, 35–61. 3, 44

[46] Kress-Gazit, H., Fainekos, G. E., Pappas, G. J., 2007. Where's Waldo? Sensor-based temporal logic motion planning. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA). pp. 3116–3121. 44

[47] Kress-Gazit, H., Fainekos, G. E., Pappas, G. J., 2009. Temporal-logic-based reactive mission and motion planning. IEEE Transactions on Robotics 25 (6), 1370–1381. 44

[48] Kupferman, O., Vardi, M. Y., 1996. Module checking. In: Proceedings of the International Conference Computer Aided Verification (CAV). pp. 75–86. 2

[49] Kupferman, O., Vardi, M. Y., 1999. Church's problem revisited. The Bulletin of Symbolic Logic 5 (2), 245–263. 2

[50] Lundh, R., Karlsson, L., Saffiotti, A., 2007. Plan-based configuration of an ecology of robots. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA). pp. 64–70. 46

[51] Lundh, R., Karlsson, L., Saffiotti, A., 2008. Automatic configuration of multi-robot systems: Planning for multiple steps. In: Proceedings of the European Conference in Artificial Intelligence (ECAI). pp. 616–620. 46

[52] Lustig, Y., Vardi, M. Y., 2009. Synthesis from component libraries. In: Proceedings of the International Conference on Foundations of Software Science and Computational Structures (FOSSACS). Vol. 5504 of LNCS. Springer, pp. 395–409. 2, 44

[53] Marin, O., Bertier, M., Sens, P., 2003. DARX - a Framework for the Fault Tolerant Support of Agent Software. In: Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE). pp. 406–418. 29

[54] Martin, D. L., Burstein, M. H., McDermott, D. V., McIlraith, S. A., Paolucci, M., Sycara, K. P., McGuinness, D. L., Sirin, E., Srinivasan, N., 2007. Bringing semantics to web services with OWL-S. In: Proceedings of the International Conference on World Wide Web (WWW). pp. 243–277. 45

[55] McIlraith, S. A., Son, T. C., 2002. Adapting golog for composition of semantic web services. In: Proceedings of Principles of Knowledge Representation and Reasoning (KR). pp. 482–496. 45

[56] McIlraith, S. A., Son, T. C., Zeng, H., 2001. Semantic web services. IEEE Intelligent Systems 16 (2), 46–53. 45

[57] McMillan, K. L., 1993. Symbolic Model Checking. Kluwer Academic Publishers, Norwell, MA, USA. 40

[58] Milner, R., 1971. An algebraic definition of simulation between programs. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI). pp. 481–489. 3, 11

[59] Muscholl, A., Walukiewicz, I., 2008. A lower bound on web services composition. Logical Methods in Computer Science 4 (2). 3, 16, 40

[60] Nau, D. S., Au, T.-C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., Yaman, F., 2003. SHOP2: An HTN planning system. Journal of Artificial Intelligence Research 20, 379–404. 45

[61] Papazoglou, M. P., Traverso, P., Dustdar, S., Leymann, F., 2007. Service-oriented computing: State of the art and research challenges. IEEE Computer 40 (11), 38–45. 2

[62] Pettersson, O., 2005. Execution monitoring in robotics: A survey. Robotics and Autonomous Systems 53 (2), 73–88. 29

[63] Pistore, M., Barbon, F., Bertoli, P., Shaparau, D., Traverso, P., 2004. Planning and monitoring web service composition. In: Proceedings of the Artificial Intelligence: Methodology, Systems, and Applications (AIMSA). Vol. 3192 of LNCS. Springer, pp. 106–115. 44

[64] Pistore, M., Marconi, A., Bertoli, P., Traverso, P., 2005. Automated composition of web services by planning at the knowledge level. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI). pp. 1252–1259. 44

[65] Pistore, M., Traverso, P., 2001. Planning as model checking for extended goals in non-deterministic domains. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI). pp. 479–486. 45

[66] Pistore, M., Traverso, P., Bertoli, P., Marconi, A., 2005. Automated synthesis of composite BPEL4WS web services. In: Proceedings of the IEEE International Conference on Web Services (ICWS). pp. 293–301. 44

[67] Piterman, N., Pnueli, A., Sa'ar, Y., 2006. Synthesis of Reactive(1) Designs. In: Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI). pp. 364–380. 2, 3, 29, 31, 44

[68] Pnueli, A., Rosner, R., 1989. On the synthesis of a reactive module. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 179–190. 2, 44

[69] Pnueli, A., Shahar, E., 1996. A platform for combining deductive with algorithmic verification. In: Proceedings of the International Conference Computer Aided Verification (CAV). pp. 184–195. 3, 4, 29, 40

[70] Pynadath, D. V., Tambe, M., Chauvat, N., Cavedon, L., 2000. Toward team-oriented programming. In: Proceedings of the International Workshop on Agent Theories, Architectures, and Languages (ATAL). Springer, pp. 233–247. 45

[71] Ragab Hassen, R., Nourine, L., Toumani, F., 2008. Protocol-based web service composition. In: Proceedings of the International Joint Conference on Service Oriented Computing (ICSOC). Vol. 5364 of LNCS. Springer, Ch. 7, pp. 38–53. 44

[72] Ramadge, P. J., Wonham, W. M., 1987. Supervisory control of a class of discrete event processes. SIAM Journal on Control and Optimization 25, 206–230. 47

[73] Rao, A. S., 1996. AgentSpeak(L): BDI agents speak out in a logical computable language. In: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World. (Agents Breaking Away). Vol. 1038 of LNCS. Springer, pp. 42–55. 2, 45

[74] Saffiotti, A., Broxvall, M., 2005. PEIS ecologies: Ambient intelligence meets autonomous robotics. In: Proceedings of the International Conference on Smart Objects and Ambient Intelligence. pp. 275–280. 2, 46

[75] Sardina, S., De Giacomo, G., 2008. Realizing multiple autonomous agents through scheduling of shared devices. In: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS). pp. 304–312. 43, 44

[76] Sardina, S., De Giacomo, G., 2009. Composition of ConGolog programs. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI). pp. 904–910. 43, 44

[77] Sardina, S., Patrizi, F., De Giacomo, G., 2007. Automatic synthesis of a global behavior from multiple distributed behaviors. In: Proceedings of the National Conference on Artificial Intelligence (AAAI). pp. 1063–1069. 1, 2, 16, 21, 42, 43, 44

[78] Sardina, S., Patrizi, F., De Giacomo, G., 2008. Behavior composition in the presence of failure. In: Proceedings of Principles of Knowledge Representation and Reasoning (KR). pp. 640–650. 1

[79] Schoppers, M. J., 1987. Universal plans for reactive robots in unpredictable environments. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI). pp. 1039–1046. 45

[80] Sirin, E., Parsia, B., Wu, D., Hendler, J., Nau, D., Oct. 2004. HTN planning for web service composition using SHOP2. Journal Web Semantics: Science, Services and Agents on the World Wide Web 1 (4), 377–396. 45

[81] Sohrabi, S., Prokoshyna, N., McIlraith, S. A., 2006. Web service composition via generic procedures and customizing user preferences. In: Proceedings of the International Semantic Web Conference (ISWC). pp. 597–611. 45

[82] Sohrabi, S., Prokoshyna, N., Mcilraith, S. A., 2009. In: Borgida, A. T., Chaudhri, V. K., Giorgini, P., Yu, E. S. (Eds.), Conceptual Modeling: Foundations and Applications. Springer, Ch. Web Service Composition via the Customization of Golog Programs with User Preferences, pp. 319–334. 2

[83] Stroeder, T., Pagnucco, M., 2009. Realising deterministic behaviour from multiple non-deterministic behaviours. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI). pp. 936–941. 44

[84] Su, J. (Ed.), Sep. 2008. IEEE Data Engineering Bulletin. Vol. 31. IEEE Computer Society Press. 2

[85] Tan, L., Cleaveland, R., 2001. Simulation revisited. In: Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 480–495. 2

[86] Tripathi, A., Miller, R., 2001. Exception handling in agent-oriented systems. In: Romanovsky, A., Dony, C., Knudsen, J., Tripathi, A. (Eds.), Advances in Exception Handling Techniques. Vol. 2022 of LNCS. Springer, pp. 128–146. 29

[87] Vardi, M. Y., 1995. An automata-theoretic approach to fair realizability and synthesis. In: Proceedings of the International Conference Computer Aided Verification (CAV). pp. 267–278. 2

[88] Wonham, W., Ramadge, P., 1987. On the supremal controllable sub-language of a given language. SIAM Journal on Control and Optimization 25 (3), 637–659. 47

[89] Yadav, N., Sardina, S., 2011. Decision theoretic behavior composition. In: Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS). To appear. 47

A. TLV Implementation for the Painting Block Example

We list here the SMV code that completes the one presented in Figure 6. As for module `main`, we refer the reader to Figure 6, where the full encoding is reported. Concerning the code for module `Environment`, it is as follows:

```
MODULE Environment(act) -- Environment
VAR
 st: {ini,e1,e2,e3,e4};
INIT
 st = ini
TRANS
 case
  st = ini & act = start : next(st) = e1;
  act = none : next(st) = st;
  st = e1 & act = recharge : next(st) = e1;
  st = e1 & act = prepare : next(st) = e2;
  st = e2 & act in {paint,recharge} : next(st) = e2;
  st = e2 & act = dispose : next(st) = e1;
  st = e2 & act = clean : next(st) in {e2,e3}; -- nondet!
  st = e3 & act in {paint,clean}: next(st) = e3;
  st = e3 & act = dispose : next(st) = e4;
  st = e3 & act = recharge : next(st) = e2;
  st = e4 & act = prepare : next(st) = e3;
  st = e4 & act = recharge : next(st) = e1;
  TRUE : FALSE; -- no other transitions possible!
 esac
DEFINE
 initial := st = ini;
```

Observe that the environment has one dummy state `ini` and one dummy action `start`, which, when executed in the initial state, makes the environment move to state e_1. Every line in the `TRANS` section encodes a transition, that is, it defines the next state of the module (`next(st)`) given the environment's current state (`st`) and the action being performed, which is an *input* parameter (variable `act`).

We next list the code corresponding to the three available arms $\mathcal{B}_1$, $\mathcal{B}_2$, and $\mathcal{B}_3$. Their encoding is similar to that of the environment, though with some differences. Firstly, as the dynamics of each behavior—captured in the module's `TRANS` section—depends on both the action being performed by the behavior itself and the current environment state, both the action and the environment state appear as inputs (variables `act` and `env`) in each behavior module. As for the `TRANS` section, similarly to the environment's, each of its entries within the `case` body captures a behavior transition. In particular, observe that every behavior may be instructed to execute the dummy action `none` (second entry in `TRANS`), i.e., a no-op action that yields no state change in the module. Through this mechanism we implement the asynchronous execution of available behavior modules, as explained in Section 6. Secondly, to account for guards, the transitions occurring in a behavior module may contain (boolean) formulae involving the current state of the environment. For example, the fourth transition in the `ArmA` module states that the next state of the behavior is `a2`, provided: the current state is `a1`, the behavior is executing action `clean`, and the environment is in either state `e1` or `e2`. Finally, each behavior defines its initial, final, and failure conditions. In particular, behavior failure is accounted for by introducing the distinguished absorbing state `failed`, that the module reaches whenever no transition rule applies for the current action and environment state input, i.e., when the behavior cannot legally execute the requested action.

```
MODULE ArmA(act, env)
VAR
 st: {ini,failed,a1,a2};
INIT
 st = ini
TRANS
 case
  st = ini & act = start :
    next(st) = a1;
  act = none : next(st) = st;
  st = a1 & act in {dispose,recharge} :
    next(st) = a1;
  st = a1 & act=clean & env in {e1,e2}:
    next(st) = a2;
  st = a2 & act = recharge :
    next(st) = a2;
  st = a2 & act = dispose :
    next(st) = a1;
  TRUE : next(st) = failed;
 esac
DEFINE
 initial := st = ini;
 final:= st = a1;
 fail := state = failed;
```

```
MODULE ArmB(act, env)
VAR
  st: {ini,failed,b1,b2,b3,b4};
INIT
 st = ini
TRANS
 case
  st = ini & act = start :
      next(st) = b1;
  act = none : next(st) = st;
  st = b1 & act = prepare :
      next(st) = b2;
  st = b2 & act = clean :
     next(st) = b1;
  st = b2 & act = paint :
     next(st) in {b1,b3};
  st = b3 & act = recharge :
     next(st) = b1;
  st = b3 & act = prepare :
     next(st) = b4;
   st = b4 & act = clean :
     next(st) = b3;
  TRUE : next(st) = failed;
 esac
DEFINE
 initial := st = ini;
 final:= st = b1;
 fail := state = failed;
```

```
MODULE ArmC(act, env)
VAR
 st: {ini,failed,c1,c2};
INIT
 st = ini
TRANS
 case
  st = ini & act = start : next(st) = c1;
  act = none : next(st) = st;
  st = c1 & act = recharge : next(st) = c2;
  st = c2 & act = prepare : next(st) = c2;
  st = c2 & act = paint : next(st) = c1;
  TRUE : next(st) = failed; -- failed!
 esac
DEFINE
 initial := st = ini;
 final:= st = c1;
 fail := state = failed;
```

The target specification is even simpler, as the target may not include any non-deterministic transition:

```
MODULE Target(env, req)
VAR
  state: {ini,t1,t2,t3,t4,t5};
INIT
  state = ini & req = start
TRANS
 case
  state = ini & req = start  : next(state) = t1;
  req = none  : next(state) = state;
  state = t1 & req = prepare  : next(state) = t2;
  state = t2 & req = paint  : next(state) = t4;
  state = t2 & req = clean  : next(state) = t3;
  state = t3 & req = paint  : next(state) = t4;
  state = t4 & req = dispose  : next(state) = t5;
  state = t5 & req = recharge  : next(state) = t1;
 esac
DEFINE
 initial:= state = ini;
 final:= state = t1;
```

Using the target, we can then specify the client, which is meant to issue the request actions according, of course, to the target behavior:

```
MODULE Client(env)
VAR
 target : Target(env, req);
 req: {start,none,prepare,clean,paint,dispose,recharge};
INIT
 req = start
TRANS
 case
   next(tst) = t1 : next(req) = prepare;
   next(tst) = t2 : next(req) in {paint,clean} ;
   next(tst) = t3 : next(req) = paint;
   next(tst) = t4 : next(req) = dispose;
   next(tst) = t5 : next(req) = recharge;
   TRUE : next(req) = none;
 esac
DEFINE
 initial:= target.initial;
 tst := target.state;
 final:= target.final;
```

When the full specification is run against the $\text{TLV}_{\Box}$ system, the following output is obtained:

```
TLV version 4.18.4
...
Resources used: user time: 0.11 s
BDD nodes allocated: 125962
max amount of BDD nodes allocated: 125962
Bytes allocated: 2228288
...

Automaton States

State 1
sys.availsys.env.state = start_st,      sys.availsys.a1.state = start_st,
sys.availsys.a2.state = start_st,       sys.availsys.a3.state = start_st,
sys.client.target.state = start_st,     sys.client.req = start_op,
contr.a1op = start_op, contr.a2op = start_op, contr.a3op = start_op,

State 2
sys.availsys.env.state = e1,            sys.availsys.a1.state = a1,
sys.availsys.a2.state = b1,             sys.availsys.a3.state = c1,
sys.client.target.state = t1,           sys.client.req = prepare,
contr.a1op = none, contr.a2op = prepare, contr.a3op = none,

State 3
sys.availsys.env.state = e2,            sys.availsys.a1.state = a1,
sys.availsys.a2.state = b2,             sys.availsys.a3.state = c1,
sys.client.target.state = t2,           sys.client.req = paint,
contr.a1op = none, contr.a2op = paint, contr.a3op = none,

State 4
sys.availsys.env.state = e2,            sys.availsys.a1.state = a1,
sys.availsys.a2.state = b2,             sys.availsys.a3.state = c1,
sys.client.target.state = t2,           sys.client.req = clean,
contr.a1op = clean, contr.a2op = none, contr.a3op = none,

State 5
sys.availsys.env.state = e2,            sys.availsys.a1.state = a2,
sys.availsys.a2.state = b2,             sys.availsys.a3.state = c1,
sys.client.target.state = t3,           sys.client.req = paint,
contr.a1op = none, contr.a2op = paint, contr.a3op = none,

State 6
```

```
sys.availsys.env.state = e3,            sys.availsys.a1.state = a2,
sys.availsys.a2.state = b2,             sys.availsys.a3.state = c1,
sys.client.target.state = t3,           sys.client.req = paint,
contr.a1op = none, contr.a2op = paint, contr.a3op = none,

State 7
sys.availsys.env.state = e3,            sys.availsys.a1.state = a2,
sys.availsys.a2.state = b3,             sys.availsys.a3.state = c1,
sys.client.target.state = t4,           sys.client.req = dispose,
contr.a1op = dispose, contr.a2op = none, contr.a3op = none,

State 8
sys.availsys.env.state = e3,            sys.availsys.a1.state = a2,
sys.availsys.a2.state = b1,             sys.availsys.a3.state = c1,
sys.client.target.state = t4,           sys.client.req = dispose,
contr.a1op = dispose, contr.a2op = none, contr.a3op = none,

State 9
sys.availsys.env.state = e4,            sys.availsys.a1.state = a1,
sys.availsys.a2.state = b1,             sys.availsys.a3.state = c1,
sys.client.target.state = t5,           sys.client.req = recharge,
contr.a1op = recharge, contr.a2op = none, contr.a3op = none,

State 10
sys.availsys.env.state = e4,            sys.availsys.a1.state = a1,
sys.availsys.a2.state = b3,             sys.availsys.a3.state = c1,
sys.client.target.state = t5,           sys.client.req = recharge,
contr.a1op = none, contr.a2op = recharge, contr.a3op = none,

State 11
sys.availsys.env.state = e2,            sys.availsys.a1.state = a2,
sys.availsys.a2.state = b3,             sys.availsys.a3.state = c1,
sys.client.target.state = t4,           sys.client.req = dispose,
contr.a1op = dispose, contr.a2op = none, contr.a3op = none,

State 12
sys.availsys.env.state = e2,            sys.availsys.a1.state = a2,
sys.availsys.a2.state = b1,             sys.availsys.a3.state = c1,
sys.client.target.state = t4,           sys.client.req = dispose,
contr.a1op = dispose, contr.a2op = none, contr.a3op = none,

State 13
sys.availsys.env.state = e1,            sys.availsys.a1.state = a1,
sys.availsys.a2.state = b1,             sys.availsys.a3.state = c1,
sys.client.target.state = t5,           sys.client.req = recharge,
contr.a1op = recharge, contr.a2op = none, contr.a3op = none,

State 14
sys.availsys.env.state = e1,            sys.availsys.a1.state = a1,
sys.availsys.a2.state = b3,             sys.availsys.a3.state = c1,
sys.client.target.state = t5,           sys.client.req = recharge,
contr.a1op = none,  contr.a2op = recharge, contr.a3op = none,

State 15
sys.availsys.env.state = e2,            sys.availsys.a1.state = a1,
sys.availsys.a2.state = b3,             sys.availsys.a3.state = c1,
sys.client.target.state = t4,           sys.client.req = dispose,
contr.a1op = dispose, contr.a2op = none, contr.a3op = none,

State 16
sys.availsys.env.state = e2,            sys.availsys.a1.state = a1,
sys.availsys.a2.state = b1,             sys.availsys.a3.state = c1,
sys.client.target.state = t4,           sys.client.req = dispose,
contr.a1op = dispose, contr.a2op = none, contr.a3op = none,


 Automaton Transitions
```

```
From 1 to  2
From 2 to  3 4
From 3 to  15 16
From 4 to  5 6
From 5 to  11 12
From 6 to  7 8
From 7 to  10
From 8 to  9
From 9 to  2
From 10 to  2
From 11 to  14
From 12 to  13
From 13 to  2
From 14 to  2
From 15 to  14
From 16 to  13

Automaton has 16 states, and 21 transitions
```

The output states that an automaton with 16 states and 21 transition was successfully synthesized. Observe that the automaton encodes and accounts for the constraints of both the whole system and the client running the target, as well as the controller performing the composition. In fact, the obtained result can be regarded as a representation of the controller generator for the painting blocks example. States can be read as follows: an assignment to variables `sys.availsys.env.state`, `sys.availsys.a1.state`, `sys.availsys.a2.state`, `sys.availsys.a3.state`, and `sys.client.target.state`, forms the current state of the enacted system; an assignment to `sys.client.req` represents the action currently requested; and an assignment to `contr.a1op`, `contr.a2op`, and `contr.a3op`, represents a possible action delegations to available behaviors for fulfilling the current request.