

## Behavior Composition in the Presence of Failure

**Sebastian Sardina**

Department of Computer Science  
RMIT University  
Melbourne, Australia

sebastian.sardina@rmit.edu.au

**Fabio Patrizi and Giuseppe De Giacomo**

Dipartimento di Informatica e Sistemistica  
Sapienza Università di Roma  
Roma, Italy

{fabio.patrizi, degiacomo}@dis.uniroma1.it

### Abstract

In this paper we articulate theoretical bases for *robust* behavior composition of multiple modules (e.g., agents, devices, etc.) by relying on the formal notion of *simulation*. Specifically, 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. Both behaviors and environment are represented as finite state transition systems. While previous solutions to this problem assumed full reliability, here we consider *unforeseen potential failures*, such as a module, or the environment, unexpectedly changing its state, or a module becoming temporarily unavailable or dropping out permanently. Based on the notion of *simulation*, we propose an alternative synthesis approach that allows for *refining* the solution at hand, either *on-the-fly* or *parimoniously*, so as to cope with failures. Interestingly, it turns out that the proposed simulation-based technique is computationally an improvement over previously known methods that assumed full-reliability.

### Introduction

In this paper we articulate theoretical bases for robust behavior composition of multiple modules (e.g., agents, devices, etc.). Specifically, 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 (De Giacomo & Sardina 2007; Sardina, Patrizi, & De Giacomo 2007). A behavior stands for the logic of any artifact that is able to operate in the environment. 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, all acting in the same environment. The aim is to realize a desired (intelligent) virtual painting system by suitably “combining” the available arms.

Technically, we abstract the actual behaviors and 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 nondeterministic 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 the area of Verification (Pitman, Pnueli, & Sa’ar 2006; Tan & Cleaveland 2001; Kupferman & Vardi 1996; Alura, Henzinger, & Kupferman 2002; Clarke, Grumberg, & Peled 1999).

Solving the composition problem consists in automatically synthesizing —(Pnueli & Rosner 1989)— a *controller* that coordinates the (partially controllable) available behaviors to obtain the target behavior (De Giacomo & Sardina 2007). This synthesis problem can be recast in a variety of forms within several sub-areas of AI and CS, including web-service composition (McIlraith & Son 2002; Berardi *et al.* 2005; Muscholl & Walukiewicz 2007), agent-oriented programming (Georgeff & Lansky 1987), robotics (Pettersson 2005), planning (Ghallab, Nau, & Traverso 2004), and plan coordination and monitoring (Katz & Rosenschein 1993; Grosz & Kraus 1996; Tripathi & Miller 2001).

In the literature, the above behavior composition setting has so far always been studied assuming *full reliability* of all available modules and, as a result, the (default) approach for dealing with behavior failures is to “re-plan” for a new solution, if any, from scratch. It is obvious that full reliability is an unrealistic assumption in many dynamic settings, where modules may become unexpectedly unavailable for various reasons. For instance, an agent (e.g., a RoboCup robot player) may, at some point, break down or opt not to participate in the composition anymore, possibly because it has agreed to join another behavior composition. It could also be the case that, while still cooperating, the agents may move too far apart losing the communication. The unavailability of a behavior may be temporary, i.e., the behavior will eventually resume operation, or permanent, i.e., the behavior will not participate any more in the overall system.

In this paper, we propose a solution for the composition problem that is able to cope with unexpected behavior failures in an incremental, and often fully reactive, way. Specifically, we propose a novel technique to synthesize the controller that is based on the formal notion of *simulation* (Milner 1971; Henzinger, Henzinger, & Kopke 1995). We argue that, when it comes to behavior failures, the composi-

tion solution obtained is *robust* in two ways. First, it can handle temporary behavior unavailability as well as 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. Second, the composition solution can be *parsimoniously refined* when a module becomes permanently unavailable, or unexpectedly resumes operation.

Interestingly, the results here show that the computational complexity of synthesizing such robust solutions remains the same as in the case of full-reliability. In fact, the technique we propose improves the known results by better characterizing the sources of complexity (cf. Theorem 2).

We remark that it is not the objective of this work to guarantee up-front to stand (any) potential failures. That could possibly be achieved by extending each behavior with a distinguished “failure” state and adding corresponding transitions from where failure may occur. Instead, if we think of each available module’s transition system as a “contract,” what we want is to address *unforeseen* breaches of such contract. The failures we investigate here can therefore be seen as the “core” ways of breaking the contract represented by the transition systems.

The rest of the paper is organized as follows. We first describe the general setting and problem we are concerned with. After that, we explain the role of potential failures within such framework. Then, we propose a new approach to the problem at hand by appealing to the notion of simulation. In the next two sections, we show how the new approach can be used to cope with the discussed failures. We end the paper by drawing some conclusions.

## The Framework

The setting we are concerned with is that in (De Giacomo & Sardina 2007), summarized below. For the sake of brevity we make some minor and non-substantial simplifications with respect to the original one. (In particular, we drop “final states” in transition systems—every state may be considered “final” in De Giacomo & Sardina 2007’s terminology.)

**Environment and behaviors** We assume to have a shared fully observable environment, which provides an abstract account of actions’ preconditions and effects, and a mean of communication among modules. In doing so, we take into consideration that, in general, we have *incomplete information* about the actual preconditions and effects of actions (akin to an action theory). Therefore, we allow the environment to be *nondeterministic* in general. In other words, the incomplete information on the actual world, and hence its partial predictability, shows up as nondeterminism in our setting. 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’s states;
- $e_0 \in E$  is the initial state;
- $\rho \subseteq E \times \mathcal{A} \times E$  is the transition relation among states:  $\langle e, a, e' \rangle \in \rho$ , or  $e \xrightarrow{a} e'$  in  $\mathcal{E}$ , denotes that action  $a$  per-

formed in state  $e$  may lead the environment to a successor state  $e'$ .

A behavior is essentially a program for an agent—or the logic of some available device—which provides, step by step, the agent with a set of actions that can be performed. Precisely, at each step, the agent selects one action among those provided and executes it. Then, a new set of actions is provided, the agent selects one, executes it, and so on. Obviously, behaviors are not intended to be executed on their own but, rather, to interact with the environment (cf. above). Hence, they are equipped with the ability to test conditions (i.e., guards) on the environment, when needed. Formally, a *behavior* over an environment  $\mathcal{E} = \langle \mathcal{A}, E, e_0, \rho \rangle$  is a tuple  $\mathcal{B} = \langle B, b_0, G, \varrho \rangle$ , where:

- $B$  is the finite set of behavior’s states;
- $b_0 \in B$  is the initial state;
- $G$  is a set of *guards*, that is, boolean functions  $g : E \mapsto \{\text{true}, \text{false}\}$ ;
- $\delta \subseteq B \times G \times \mathcal{A} \times B$  is the behavior’s transition relation, where  $\langle b, g, a, b' \rangle \in \varrho$ , or  $b \xrightarrow{g,a} b'$  in  $\mathcal{B}$ , denotes that action  $a$  executed in behavior state  $b$ , when the environment is in a state  $e$  such that  $g(e) = \text{true}$ , may lead the behavior to a successor state  $b'$ .

Observe that behaviors are, in general, *nondeterministic*, that is, given a state and an action, there may be several transitions whose guards evaluate to `true`. Consequently, when choosing the action to execute next, one cannot be certain of the resulting state, and hence of which actions will be available later on, since this depends on what particular transition happens to take place. In other words, nondeterministic behaviors are only *partially controllable*.

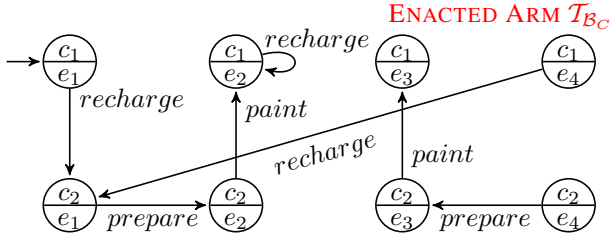
We say that a behavior  $\mathcal{B} = \langle B, b_0, G, \varrho \rangle$  over an environment  $\mathcal{E} = \langle \mathcal{A}, E, e_0, \rho \rangle$  is *deterministic* if there are no behavior state  $b \in B$  and no environment state  $e \in E$  for which there exist two transitions  $b \xrightarrow{g_1,a} b'$  and  $b \xrightarrow{g_2,a} b''$  in  $\mathcal{B}$  such that  $b' \neq b''$  and  $g_1(e) = g_2(e) = \text{true}$ . Notice that, given a state in a deterministic behavior and a legal action in that state, we always know exactly *the* next behavior’s state. In other words, deterministic behaviors are indeed *fully controllable* through the selection of the next action to perform.

A *system*  $\mathcal{S} = \langle \mathcal{B}_1, \dots, \mathcal{B}_n, \mathcal{E} \rangle$  is built from an environment  $\mathcal{E}$  and  $n$  predefined, possibly nondeterministic, *available behaviors*  $\mathcal{B}_i$  over  $\mathcal{E}$ . A *target behavior* is a *deterministic* behavior over  $\mathcal{E}$  that represents the fully controllable desired behavior to be obtained through the available behaviors.

**Example 1.** Figure 1 depicts an *extended* version of the painting arms scenario described in (De Giacomo & Sardina 2007). The overall aim of the system is to process existing blocks, which can be cleaned and painted. Before being processed, a block needs to be prepared; only one block at a time can be processed. Cleaning and painting require resources, namely, water and paint, respectively: we assume there are two tanks, for water and paint, and that both are recharged simultaneously by pressing a recharging button.

The nondeterministic environment  $\mathcal{E}$  provides the general rules of the domain. For instance, blocks can be painted or





Observe that some joint states may be reached (only) when other behaviors are also acting: state  $\langle c_1, e_4 \rangle$  would be reached after actions *prepare*, *clean*, and *dispose* are executed. ■

**Controller and composition** The *controller* is a system component able to activate, stop, and resume any of the available behaviors, and to instruct them to execute an action among those allowed in their current state (of course, also taking the environment into account). The controller has *full observability* on the 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 the natural one in this context, since both the available behaviors and the environment are already suitable abstractions for *actual* modules: if details have to be hidden, this can be done directly within the abstract behaviors exposed, by means of nondeterminism.

To formally define controllers, we first need the following technical notions. A *trace* for a given enacted behavior  $\mathcal{T}_B = \langle S, \mathcal{A}, s_0, \delta \rangle$  is a, possibly infinite, sequence of the form  $s^0 \xrightarrow{a^1} s^1 \xrightarrow{a^2} \dots$ , such that (i)  $s^0 = s_0$ ; and (ii)  $s^j \xrightarrow{a^{j+1}} s^{j+1}$  in  $\mathcal{T}_B$ , for all  $j > 0$ . A *history* is just a finite prefix  $h = s^0 \xrightarrow{a^1} \dots \xrightarrow{a^\ell} s^\ell$  of a trace. We denote  $s^\ell$  by *last*( $h$ ), and  $\ell$  by *length*( $h$ ). The notions of trace and history extend immediately to enacted system behaviors: system traces have the form  $s^0 \xrightarrow{a^1, k^1} s^1 \xrightarrow{a^2, k^2} \dots$ , and system histories have the form  $s^0 \xrightarrow{a^1, k^1} \dots \xrightarrow{a^\ell, k^\ell} s^\ell$ .

Let  $\mathcal{S} = \langle \mathcal{B}_1, \dots, \mathcal{B}_n, \mathcal{E} \rangle$  be a system and  $\mathcal{H}$  be the set of its system histories (i.e., histories of  $\mathcal{T}_S$ ). A *controller* for system  $\mathcal{S}$  is a function  $P : \mathcal{H} \times \mathcal{A} \mapsto \{1, \dots, n, u\}$  which, given a system history  $h \in \mathcal{H}$  and an action  $a \in \mathcal{A}$  to perform, selects a behavior—actually, returns its index—to delegate  $a$  to for execution. For technical convenience, a special value  $u$  (“undefined”) may be returned, thus making  $P$  a total function which returns a value even for irrelevant histories or actions that no behavior can perform after a given history.

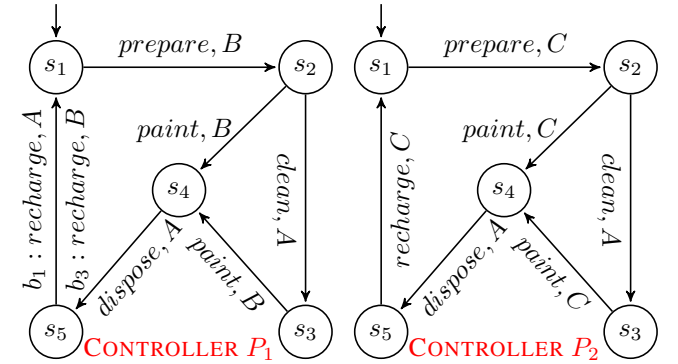
The problem we are interested in is the following: given a system  $\mathcal{S} = \langle \mathcal{B}_1, \dots, \mathcal{B}_n, \mathcal{E} \rangle$  and a deterministic *target* behavior  $\mathcal{B}_t$  over  $\mathcal{E}$ , *synthesize a controller  $P$  which realizes the target behavior  $\mathcal{B}_t$  by suitably delegating each action requested by  $\mathcal{B}_t$  to one of the available behaviors  $\mathcal{B}_i$  in  $\mathcal{S}$ . A solution to such problem is called a *composition*.*

Intuitively, the controller realizes a target if for every trace of the enacted target, at every step, it returns the index of an available behavior that can perform the requested action.

Note that these controllers are somewhat akin to an advanced form of conditional plans and, in fact, the problem itself is related to planning (Ghallab, Nau, & Traverso 2004), 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.

One can formally define when a controller realizes the target behavior—a solution to the problem—as done in (De Giacomo & Sardina 2007). In particular, one first defines when a controller  $P$  *realizes a trace* of the target  $\mathcal{B}_t$ . Then, since the target behavior is a deterministic transition system, and thus its behavior is completely characterized by its set of traces, one defines that a *controller  $P$  realizes the target behavior  $\mathcal{B}_t$*  iff it realizes all its traces.

**Example 3.** Let  $P_1$  and  $P_2$  be the two finite controllers depicted below. Their main difference has to do with the arm used to paint blocks: while  $P_1$  uses arm  $\mathcal{B}_B$ , the latter uses arm  $\mathcal{B}_C$ . Also,  $P_1$  recharges the tanks with either  $\mathcal{B}_A$  or  $\mathcal{B}_B$ , depending on  $\mathcal{B}_B$ 's state: if arm  $\mathcal{B}_B$  is in state  $b_1$ , then arm  $\mathcal{B}_A$  is used to recharge; and if arm  $\mathcal{B}_B$  is in state  $b_3$ , then arm  $\mathcal{B}_B$  is used instead. On the other hand, controller  $P_2$  always uses arm  $\mathcal{B}_C$  to recharge the tanks.



The controller  $P_1$  is indeed a composition of  $\mathcal{B}_T$  on  $\mathcal{E}$ , that is,  $P_1$  realizes all the traces of  $\mathcal{T}_{\mathcal{B}_T}$ . This is not the case for controller  $P_2$ , which does not even realize the simple one-action trace  $\langle t_1, e_1 \rangle \xrightarrow{\text{prepare}} \langle t_2, e_2 \rangle$  of  $\mathcal{T}_{\mathcal{B}_T}$ .

Finally, take  $P'_1$  to be like  $P_1$  but with the edge from  $s_5$  to  $s_1$  re-labeled “recharge, A” (i.e., action *recharge* is to be *always* delegated to arm  $\mathcal{B}_A$ ). Then,  $P'_1$  would only realize those traces where behavior  $\mathcal{B}_B$  always happens to evolve to state  $b_1$  after doing a *paint* action. Because of that,  $P'_1$  would not count as a solution either. ■

We close this section by pointing out that techniques for checking the existence of (and indeed synthesizing) a controller are known (De Giacomo & Sardina 2007; Sardina, Patrizi, & De Giacomo 2007). Such techniques are based on a reduction to PDL satisfiability (Harel, Kozen, & Tiuryn 2000), and provide an EXPTIME upper-bound to the computational complexity, being at most exponential in the number of states of the available behaviors, of the environment, and of the target behavior. Note that this bound is actually tight since EXPTIME-hardness was shown in (Muscholl & Walukiewicz 2007).

## On Behavior Failures

In discussing the above behavior composition problem, we have *implicitly* assumed that the available component modules are fully reliable—they are always available and behave “correctly” relative to the behavior/environment specification provided to the system.

Nonetheless, there are many situations and domains in which assuming full reliability of components is not adequate. For example, in multi-agent complex and highly dynamic domains, one cannot rely on the total availability nor on the reliability of all the existing modules. There are a variety of reasons why modules may stop being available at some point or another. 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 so on. Similarly, behaviors may possibly re-appear into the system at a later stage, thus creating new “opportunities” for the overall system.

As mentioned before, behaviors’ and environment’s specifications can be seen as contracts, and failures, as the ones above, as breaches of such contracts. We identify five core ways of breaking contracts, namely:<sup>1</sup>

- (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 such an opportunity and start delegating actions to it again.

The composition techniques in (De Giacomo & Sardina 2007; Sardina, Patrizi, & De Giacomo 2007) do not address the above cases, since they assume that controllers always deal with fully reliable modules. As a consequence, upon any of the above failures, we are only left with the option of “re-planning” from scratch for a whole new controller.

What we shall propose in the remainder of this paper is an alternative way of solving the composition problem (i.e., synthesizing controllers) that is *intrinsically more robust*. Roughly speaking, this alternative approach deals 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.

<sup>1</sup>Obviously, we assume an infrastructure that is able to distinguish between these failures.

## Composition via Simulation

Let us next present our approach for synthesizing composition solutions that are suitable for dealing with faults. Such an approach is inspired by that presented in (Berardi *et al.* 2008), developed in the context of service composition and based on the standard notion of *simulation* (Milner 1971; Henzinger, Henzinger, & Kopke 1995). Intuitively, a (transition) system  $S_1$  “simulates” another system  $S_2$  if it (i.e.,  $S_1$ ) is able to *match* all of  $S_2$ ’s moves. Due to (devilish) nondeterminism of the available behaviors and the environment, we cannot use the off-the-shelf notion of simulation, but a variant which we call ND-simulation.

Let  $\mathcal{S} = \langle \mathcal{B}_1, \dots, \mathcal{B}_n, \mathcal{E} \rangle$  be a system,  $\mathcal{B}_t$  be the target behavior over  $\mathcal{E}$ , and let  $\mathcal{T}_\mathcal{S} = \langle S_\mathcal{S}, \mathcal{A}, \{1, \dots, n\}, s_{S0}, \delta_\mathcal{S} \rangle$  and  $\mathcal{T}_t = \langle S_t, \mathcal{A}, s_{t0}, \delta_t \rangle$  be the enacted system and enacted target behaviors corresponding to  $\mathcal{S}$  and  $\mathcal{B}_t$ , 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.  $env(s_t) = env(s_\mathcal{S})$ ;
2. for all  $a \in \mathcal{A}$ , there exists a  $k \in \{1, \dots, n\}$  such that for all transitions  $s_t \xrightarrow{a} s'_t$  in  $\mathcal{T}_t$ :
  - there exists a transition  $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$  in  $\mathcal{T}_\mathcal{S}$  with  $env(s'_\mathcal{S}) = env(s'_t)$ ; and
  - for all transitions  $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$  in  $\mathcal{T}_\mathcal{S}$  with  $env(s'_\mathcal{S}) = env(s'_t)$ , we have  $\langle s'_t, s'_\mathcal{S} \rangle \in R$ .

In words, if a pair is in the ND-simulation, then (i) they share the same environment; and (ii) for all moves of the target (with respect to the environment), there exists a behavior  $\mathcal{B}_k$ , which regardless of its nondeterminism, always evolves to a *successor state* which is still in the ND-simulation relation with the target. Intuitively, the (enacted) system can “match” every possible move of the (enacted) target.

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  $s_t \preceq s_\mathcal{S}$ , iff there exists an ND-simulation  $R$  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 coinductive definition. As a result, the relation  $\preceq$  is itself an ND-simulation, and it is in fact the *largest ND-simulation relation*, i.e., all ND-simulation relations are contained in  $\preceq$ . The largest ND-simulation can be computed by the following *NDS* algorithm.

---

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

---

- 1:  $\mathcal{R} := S_t \times S_\mathcal{S} \setminus \{ \langle s_t, s_\mathcal{S} \rangle \mid env(s_t) \neq env(s_\mathcal{S}) \}$
  - 2: **repeat**
  - 3:  $\mathcal{R} := (\mathcal{R} \setminus \mathcal{C})$ , where  $\mathcal{C}$  is the set of  $\langle s_t, s_\mathcal{S} \rangle \in \mathcal{R}$  such that there exists  $a \in \mathcal{A}$  for which 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  $env(s'_\mathcal{S}) = env(s'_t)$ ; or
    - (b) there exists a transition  $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$  in  $\mathcal{T}_\mathcal{S}$  such that  $env(s'_\mathcal{S}) = env(s'_t)$  but  $\langle s'_t, s'_\mathcal{S} \rangle \notin \mathcal{R}$ .
  - 4: **until**  $(\mathcal{C} = \emptyset)$
  - 5: **return**  $\mathcal{R}$
- 

Roughly speaking, the algorithm works by iteratively re-

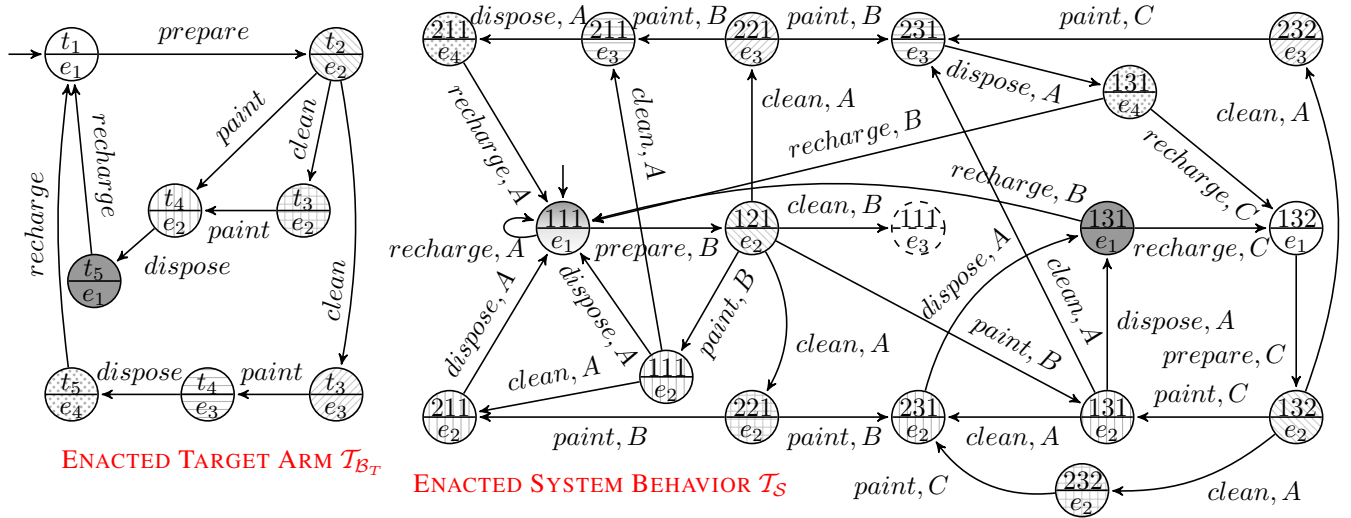


Figure 2: The largest ND-simulation relation between the enacted target behavior  $\mathcal{T}_{B_T}$  and (a part of) the enacted system behavior  $\mathcal{T}_S$  is shown using patterns. A state in  $\mathcal{T}_S$  ND-simulates the states in  $\mathcal{T}_{B_T}$  that shares its pattern, e.g.,  $\langle\langle a_1, b_3, c_1 \rangle, e_2\rangle$  in  $\mathcal{T}_S$  ND-simulates state  $\langle t_4, e_2 \rangle$  in  $\mathcal{T}_{B_T}$ . Dashed states in  $\mathcal{T}_S$  ND-simulate no state in  $\mathcal{T}_{B_T}$  (e.g., state  $\langle\langle a_1, b_1, c_1 \rangle, e_3\rangle$ ).

moving those tuples for which the conditions of the ND-simulation definition do not apply.

**Example 4.** Figure 2 shows a fragment of the largest ND-simulation relation for our painting blocks-world example. For instance, state  $\langle\langle a_1, b_3, c_2 \rangle, e_2\rangle$  in  $\mathcal{T}_S$  ND-simulates state  $\langle t_2, e_2 \rangle$  in  $\mathcal{T}_{B_T}$ , shown in the picture by the same filling pattern. So, every conceivable action taken in  $\langle t_2, e_2 \rangle$  can be replicated in  $\langle\langle a_1, b_3, c_2 \rangle, e_2\rangle$ , and moreover, this property propagates to the new resulting states. Observe that state  $\langle\langle a_1, b_1, c_1 \rangle, e_1\rangle$  in  $\mathcal{T}_S$  ND-simulates two states in  $\mathcal{T}_{B_T}$ :  $\langle t_1, e_1 \rangle$  and  $\langle t_5, e_1 \rangle$ . ■

The next result shows that checking for the existence of a composition can be reduced to checking whether there exists an ND-simulation between the enacted target and the enacted system that includes their respective initial states.

**Theorem 1.** Let  $\mathcal{S} = \langle \mathcal{B}_1, \dots, \mathcal{B}_n, \mathcal{E} \rangle$  be a system and  $\mathcal{B}_t$  a target behavior over  $\mathcal{E}$ . Let  $\mathcal{T}_t = \langle \mathcal{S}_t, \mathcal{A}, s_{t0}, \delta_t \rangle$  and  $\mathcal{T}_S = \langle \mathcal{S}_S, \mathcal{A}, \{1, \dots, n\}, s_{S0}, \delta_S \rangle$  be the enacted target behavior and the enacted system behavior for  $\mathcal{B}_t$  and  $\mathcal{S}$ , respectively. A controller  $P$  for a system  $\mathcal{S}$  that is a composition of the target behavior  $\mathcal{B}_t$  over  $\mathcal{E}$  exists iff  $s_{t0} \preceq s_{S0}$ .

*Proof (sketch).* We prove the two directions separately.

*If.* Given  $s_{t0} \preceq s_{S0}$  we show how to build a controller  $P$  that is a composition. We proceed as follows. We observe that given a history  $h$ , we can extract the resulting state of the enacted system  $s_S$  as  $last(h)$ . Moreover, we can extract the sequence of actions performed in  $h$  and the resulting environment state, and hence the state of the enacted target behavior, say  $s_t$ . Now, if tuple  $\langle s_t, s_S \rangle$  is in the largest ND-simulation, that is  $s_t \preceq s_S$ , then for every action  $a \in \mathcal{A}$  that the target may execute in  $s_t$ , there is some index  $k_a$  which maintains the ND-simulation. We then define  $P(h, a) = k_a$ . If, instead  $s_t \not\preceq s_S$ , then function  $P(h, a)$  can assume any value, in particular,  $P(h, a) = u$ . It can be shown that such

controller  $P$  is indeed a composition.

*Only-if.* We assume there exists a controller  $P$  that is a composition. Let us define relation  $R$  as the set of tuples  $\langle s_t, s_S \rangle$  for which there exists a history  $h$  obtained by running a controller  $P$  from the initial state  $s_{S0}$  such that the resulting states of the enacted target and the enacted system after history  $h$  are  $s_t$  and  $s_S$ , respectively. It can be shown that such relation  $R$  is indeed an ND-simulation of  $\mathcal{T}_t$  by  $\mathcal{T}_S$  and therefore  $R \subseteq \preceq$ . As a result, considering that  $\langle s_{t0}, s_{S0} \rangle \in R$  (by just taking  $h$  to be the initial history where no action has yet been performed), it follows that  $s_{t0} \preceq s_{S0}$ , hence the thesis holds. □

Theorem 1 gives us a straightforward method for checking the existence of a composition. Namely: (i) compute the largest ND-simulation relation of  $\mathcal{T}_t$  by  $\mathcal{T}_S$ ; and (ii) check whether  $\langle s_{t0}, s_{S0} \rangle$  is in this relation.

From the computational point of view, the algorithm *NDS* above computes the largest ND-simulation relation  $\preceq$  between  $\mathcal{T}_t$  and  $\mathcal{T}_S$  in polynomial time in the size of  $\mathcal{T}_t$  and  $\mathcal{T}_S$ . Since in our case the number of states of  $\mathcal{T}_S$  is exponential in the number of available behaviors  $\mathcal{B}_1, \dots, \mathcal{B}_n$ , we get that we can compute the largest ND-simulation relation  $\preceq$  in exponential time in the number of available behaviors. As a result, the new technique is a notable improvement with respect to the ones based on reduction to PDL (De Giacomo & Sardina 2007; Sardina, Patrizi, & De Giacomo 2007), which are exponential also in the number of states of the behaviors and of the environment.<sup>2</sup>

**Theorem 2.** Checking for the existence of compositions by computing the largest ND-simulation relation  $\preceq$  can be done

<sup>2</sup>Though in light of the result in here, a better complexity analysis involving the specific PDL satisfiability procedures could be carried out.

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.

Considering that the composition problem itself is EXPTIME-hard (Muscholl & Walukiewicz 2007), this is the best we can hope for.

Once we have computed the ND-simulation, synthesizing a controller becomes an easy task. In fact, there is a well-defined procedure that, given an ND-simulation, builds a finite state program that returns, at each point, the set of available behaviors capable of performing a target-conformant action. We call such a program *controller generator*.

Formally, let  $\mathcal{S} = \langle \mathcal{B}_1, \dots, \mathcal{B}_n, \mathcal{E} \rangle$  be a system,  $\mathcal{B}_t$  a target behavior over  $\mathcal{E}$ , and let  $\mathcal{T}_S = \langle S_S, \mathcal{A}, \{1, \dots, n\}, s_{S0}, \delta_S \rangle$  and  $\mathcal{T}_t = \langle S_t, \mathcal{A}, s_{t0}, \delta_t \rangle$  be 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  $CG = \langle \Sigma, \mathcal{A}, \{1, \dots, n\}, \partial, \omega \rangle$ , where:

1.  $\Sigma = \{ \langle s_t, s_S \rangle \in S_t \times S_S \mid s_t \preceq s_S \}$  is the set of states of CG, formed by those pairs of  $\mathcal{T}_t$ 's and  $\mathcal{T}_S$ 's states that are in the largest ND-simulation relation; given a state  $\sigma = \langle s_t, s_S \rangle$  we denote  $s_t$  by  $com_t(\sigma)$  and  $s_S$  by  $com_S(\sigma)$ .
2.  $\mathcal{A}$  is the finite set of shared actions.
3.  $\{1, \dots, n\}$  is the finite set of available behavior indexes.
4.  $\partial \subseteq \Sigma \times \mathcal{A} \times \{1, \dots, n\} \times \Sigma$  is the *transition relation*, where  $\langle \sigma, a, k, \sigma' \rangle \in \partial$ , or  $\sigma \xrightarrow{a,k} \sigma'$  in CG, iff
  - $com_t(\sigma) \xrightarrow{a} com_t(\sigma')$  in  $\mathcal{T}_t$ ;
  - $com_S(\sigma) \xrightarrow{a,k} com_S(\sigma')$  in  $\mathcal{T}_S$ ;
  - for all  $com_S(\sigma) \xrightarrow{a,k} s'_S$  in  $\mathcal{T}_S$ ,  $\langle com_t(\sigma'), s'_S \rangle \in \Sigma$ .
5.  $\omega : \Sigma \times \mathcal{A} \mapsto 2^{\{1, \dots, n\}}$  is the *output function*, where  $\omega(\sigma, a) = \{k \mid \exists \sigma' \text{ s.t. } \sigma \xrightarrow{a,k} \sigma' \text{ in CG}\}$ .

Thus, CG is a finite state transducer that, given an action  $a$  (compliant with the target behavior), outputs, through function  $\omega$ , the set of *all* available behaviors that can perform  $a$  next according to the largest ND-simulation  $\preceq$ . Observe that computing CG from the relation  $\preceq$  is easy, since it involves checking *local* conditions only.

If there exists a composition of  $\mathcal{B}_t$  by  $\mathcal{S}$ , then  $s_{t0} \preceq s_{S0}$  and CG does include state  $\sigma_0 = \langle s_{t0}, s_{S0} \rangle$ . In such cases, we get actual controllers, called *generated controllers*, which are compositions of  $\mathcal{B}_t$  by  $\mathcal{S}$ , by picking up, at each step, one available behavior among those returned by  $\omega$  in CG.

Formally we proceed as follows. A *trace for CG* starting from  $\sigma^0$  is a finite or infinite sequence of the form  $\sigma^0 \xrightarrow{a^1, k^1} \sigma^1 \xrightarrow{a^2, k^2} \dots$ , such that  $\sigma_j \xrightarrow{a^{j+1}, k^{j+1}} \sigma_{j+1}$  in CG, for all  $j$ . A *history for CG* starting from state  $\sigma^0$  is a prefix of a trace starting from state  $\sigma^0$ . By using histories, one can introduce *CG-controllers*, which are functions  $CGP_{\text{CHOOSE}} : \mathcal{H}_{CG} \times \mathcal{A} \mapsto \{1, \dots, n, u\}$ , where  $\mathcal{H}_{CG}$  is the set of CG histories starting from any state in  $\Sigma$ , and defined as follows:  $CGP_{\text{CHOOSE}}(h_{CG}, a) = \text{CHOOSE}(\omega(\text{last}(h_{CG}), a))$ ,

for all  $h_{CG} \in \mathcal{H}_{CG}$ , where CHOOSE stands for a choice function that chooses one element among those returned by  $\omega(\text{last}(h_{CG}), a)$ . Let us assume that the controller generator CG of  $\mathcal{S}$  for  $\mathcal{B}_t$  includes state  $\sigma_0 = \langle s_{t0}, s_{S0} \rangle$ . Then, for each CG's history  $h_{CG} = \sigma^0 \xrightarrow{a^1, k^1} \dots \xrightarrow{a^\ell, k^\ell} \sigma^\ell$  starting from  $\sigma^0 = \sigma_0$ , we can obtain its corresponding system history  $proj_{\mathcal{S}}(h_{CG})$ , called the *projected system history*, as follows:

$proj_{\mathcal{S}}(h_{CG}) = com_{\mathcal{S}}(\sigma^0) \xrightarrow{a^1, k^1} \dots \xrightarrow{a^\ell, k^\ell} com_{\mathcal{S}}(\sigma^\ell)$ , i.e., we take the “system” component of each CG state  $\sigma^i$  in the history. Moreover, from a CG-controller  $CGP_{\text{CHOOSE}}$ , we obtain the corresponding *generated controller* as the function  $P_{\text{CHOOSE}} : \mathcal{H} \times \mathcal{A} \mapsto \{1, \dots, n, u\}$ , where  $\mathcal{H}$  is the set of system histories starting from  $s_{S0}$ , defined as follows. For each system history  $h$  and action  $a$ : (i) if  $h = proj_{\mathcal{S}}(h_{CG})$  for some CG history  $h_{CG}$ , then  $P_{\text{CHOOSE}}(h, a) = CGP_{\text{CHOOSE}}(h_{CG}, a)$ ; else (ii)  $P_{\text{CHOOSE}}(h, a) = u$ .

Through generated controllers, we can relate CGs to compositions and show that, one gets *all* controllers that are compositions by considering all choice functions for CHOOSE. Notably, while each specific composition may be an infinite state program, the controller generator CG, which in fact includes them all, is always finite.

**Theorem 3.** *If CG includes the state  $\sigma_0 = \langle s_{t0}, s_{S0} \rangle$ , then every controller generated by CG is a composition of the target behavior  $\mathcal{B}_t$  by system  $\mathcal{S}$ .*

**Theorem 4.** *Every controller that is a composition of the target behavior  $\mathcal{B}_t$  by system  $\mathcal{S}$  can be generated by CG.*

Intuitively, CG is analogous to a sort of “meta-plan” or a stateful nondeterministic “complete universal plan,” which keeps all the existing plans at its disposal and decides which one to follow for the next action, possibly with contingent decisions.

**Example 5.** The controller generator (CG), with the largest ND-simulation at hand, can decide how to delegate actions as the target arm  $\mathcal{B}_T$ 's requests come in. For instance, if a *clean* action is requested after a block has been prepared, the CG knows it ought to delegate such request to arm  $\mathcal{B}_A$  so as to stay within the ND-simulation. While physically possible, delegating such action 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. ■

## Reactive Adaptability

Next we show that Theorems 3 and 4 give us a sound and complete technique for dealing with failure cases (a), (b), and (c) without any re-planning. As a matter of fact, once we have the controller generator CG, actual compositions can be generated “*just-in-time*,” as (target compliant) actions are requested. What is particularly interesting about CG-controllers is that one can *delay the choice* performed by CHOOSE until run-time, where one can take into account contingent information, e.g., about availability of behaviors. This gives the controller a great flexibility, which, in a sense, can “switch” compositions online as needed. We call such CG-controller, *just-in-time CG-controller*, and denote it by  $CGP_{jit}$ .

**Freezing of behaviors**  $CGP_{jit}$  already addresses temporary freezing of behaviors, i.e., failure case (a). In particular, if a behavior is temporarily frozen, then  $CGP_{jit}$  simply avoids choosing it, and continues with one of the other possible choices.<sup>3</sup> Obviously, if no other choices are possible, then  $CGP_{jit}$  shall wait for the behavior to come back.

**State change of behaviors and environment**  $CGP_{jit}$  also addresses unexpected changes in the internal state of behaviors and/or of the environment, that is, failure cases (b) and (c).<sup>4</sup> To understand this, let us denote by  $\mathcal{T}_S(z_S)$  the variant of the enacted system behavior whose initial state is  $z_S$  instead of  $s_{S0}$ . Similarly, let us denote by  $\mathcal{T}_t(z_t)$  the enacted target behavior whose initial state is  $z_t$  instead of  $s_{t0}$ . Now suppose that the state of the enacted system behavior changes, unexpectedly, to state  $\hat{s}_S$ , 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 from  $\hat{s}_S$  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  $\hat{s}_S$  (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. Therefore, the largest ND-simulation between  $\mathcal{T}_t(\hat{s}_t)$  and  $\mathcal{T}_S(\hat{s}_S)$  is the ND-simulation  $\preceq$  we already have. This implies that we can still use the very same controller generator  $CG$  (and the same just-in-time  $CG$ -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  $CG$  (and  $CGP_{jit}$  too). Put it all together, we only need to check whether  $\hat{s}_t \preceq \hat{s}_S$ , and, if so, continue to use  $CGP_{jit}$  (now from the  $CG$  history of length 0:  $\langle \hat{s}_t, \hat{s}_S \rangle$ ).

**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, suppose the target is in state  $t_3$ , the environment in state  $e_3$ , and the available behaviors  $\mathcal{B}_A$ ,  $\mathcal{B}_B$ , and  $\mathcal{B}_C$  are in their states  $a_2$ ,  $b_2$ , and  $c_2$ , respectively. That is,  $\mathcal{T}_{\mathcal{B}_T}$  is in state  $\langle t_3, e_3 \rangle$  whereas  $\mathcal{T}_S$  is in state  $\langle \langle a_2, b_2, c_1 \rangle, e_3 \rangle$ . Suppose that, in an unexpected way, the environment happens to change to state  $e_2$ —someone has re-charged the water tank. All that is needed in such case is to check that the new states of  $\mathcal{T}_{\mathcal{B}_T}$  and  $\mathcal{T}_S$ , namely  $\langle t_3, e_2 \rangle$  and  $\langle \langle a_2, b_2, c_1 \rangle, e_2 \rangle$ , are still in the ND-simulation. Since they are, the CG continues the realization of the target from such (new) enacted states. ■

**Computing reactive compositions on-the-fly** We close the section by observing that  $CGP_{jit}$ , that is  $CGP_{CHOOSE}$  with CHOOSE resolved at run-time, (and  $CG$  for the matter) can be computed *on-the-fly* by storing only the ND-simulation  $\preceq$ . In fact, at each point, the only information required for the next choice is  $\omega(\sigma, a)$ , where  $\sigma \in \Sigma$  (recall

<sup>3</sup>If more information is at hand,  $CGP_{jit}$  may use it to choose in an informed way, though this is out of the scope of this paper.

<sup>4</sup>Although hardly as meaningful as the ones above, unforeseen changes in the target's state can be accounted for in a similar way.

$\Sigma = \preceq$ ) 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  $\preceq$ .

## Parsimonious Refinement

When considering failure cases (d) and (e), a simple reactive approach is not sufficient and more complex refinement techniques are required. We show then how to do the composition refinement in an intelligent manner. Let us start by defining a parametric version of the algorithm for computing the largest ND-simulation. Such a version, called  $NDSP$ , takes two extra parameters:  $\mathcal{R}_{init}$ , the starting relation from which the largest ND-simulation is extracted; and  $\mathcal{R}_{sure}$ , a relation containing tuples already known to be in the ND-simulation to be computed.

---

**Algorithm 2**  $NDSP(\mathcal{T}_t, \mathcal{T}_S, \mathcal{R}_{init}, \mathcal{R}_{sure})$

---

- 1:  $\mathcal{R} := \mathcal{R}_{init} \setminus \mathcal{R}_{sure}$
  - 2:  $\mathcal{R} := \mathcal{R} \setminus \{ \langle s_t, s_S \rangle \mid env(s_t) \neq env(s_S) \}$
  - 3: **repeat**
  - 4:  $\mathcal{R} := (\mathcal{R} \setminus \mathcal{C})$ , where  $\mathcal{C}$  is the set of  $\langle s_t, s_S \rangle \in \mathcal{R}$  such that there exists  $a \in \mathcal{A}$  for which 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  $s_S \xrightarrow{a,k} s'_S$  in  $\mathcal{T}_S$  such that  $env(s'_t) = env(s'_S)$ ; or
    - (b) there exists a transition  $s_S \xrightarrow{a,k} s'_S$  in  $\mathcal{T}_S$  such that  $env(s'_t) = env(s'_S)$  but  $\langle s'_t, s'_S \rangle \notin \mathcal{R} \cup \mathcal{R}_{sure}$ .
  - 5: **until**  $(\mathcal{C} = \emptyset)$
  - 6: **return**  $\mathcal{R} \cup \mathcal{R}_{sure}$
- 

The next result shows that the output of algorithm  $NDSP$  coincides with that of  $NDS$ , provided its two new parameters are used adequately.

**Lemma 5.** *Let  $\mathcal{S}$  be a system and  $\mathcal{B}_t$  a target behavior. If  $\mathcal{R}_{sure} \subseteq NDS(\mathcal{T}_t, \mathcal{T}_S) \subseteq \mathcal{R}_{init}$ , then  $NDSP(\mathcal{T}_t, \mathcal{T}_S, \mathcal{R}_{init}, \mathcal{R}_{sure}) = NDS(\mathcal{T}_t, \mathcal{T}_S)$ .*

*Proof (sketch).* 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 iterations. It can be shown, by induction on  $i$ , that  $\mathcal{R}_2^i \cup \mathcal{R}_{sure} \subseteq \mathcal{R}_1^i \subseteq NDS(\mathcal{T}_t, \mathcal{T}_S)$  and that  $NDS(\mathcal{T}_t, \mathcal{T}_S) \subseteq \mathcal{R}_2^i \cup \mathcal{R}_{sure}$ . Hence, since at the limit  $\mathcal{R}_2^i \cup \mathcal{R}_{sure} = NDSP(\mathcal{T}_t, \mathcal{T}_S, \mathcal{R}_{init}, \mathcal{R}_{sure})$ , the thesis follows. □

Next, we introduce convenient notations to shrink and expand systems and ND-simulation relations. Consider a system  $\mathcal{S} = \langle \mathcal{B}_1, \dots, \mathcal{B}_n, \mathcal{E} \rangle$  and a set of behavior indexes  $W \subseteq \{1, \dots, n\}$ . 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  $\mathcal{S} = \mathcal{S}(\{1, \dots, n\})$ ). Let  $\mathcal{T}_t$  be an enacted target behavior over  $\mathcal{E}$ . We denote by  $\preceq_W$  the largest ND-simulation relation of  $\mathcal{T}_t$  by  $\mathcal{T}_{\mathcal{S}(W)}$ . Let  $U \subseteq \{1, \dots, n\}$  such that  $W \cap U = \emptyset$ . We denote by  $\preceq_W \otimes U$ , the relation obtained from  $\preceq_W$  by (trivially) putting all behaviors  $\mathcal{B}_i$ , with  $i \in U$ , back into the system. Formally, we can define such operation as follows

(without loss of generality, assume  $W = \{1, \dots, \ell\}$  and  $U = \{\ell + 1, \dots, m\}$ ):

$$\begin{aligned} \preceq_W \otimes U = & \\ & \{ \langle s_t, s' \rangle \mid s' = \langle b^1, \dots, b^\ell, b^{\ell+1}, \dots, b^m, e \rangle \\ & \text{such that } \langle s_t, \langle b^1, \dots, b^\ell, e \rangle \rangle \in \preceq_W \text{ and} \\ & b^i \text{ is a state of } \mathcal{B}_i, \text{ for } i \in \{\ell + 1, \dots, m\} \}. \end{aligned}$$

When “putting back” a set of behaviors into the system in this way, we are guaranteed to (already) get an ND-simulation for the (expanded) system  $\mathcal{S}(W \cup U)$ . Observe, however, that it may not necessarily be the largest one.

**Lemma 6.** *Let  $W, U \subseteq \{1, \dots, n\}$  such that  $W \cap U = \emptyset$ . Then,*

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

*Proof.* Without loss of generality, take  $W = \{1, \dots, \ell\}$ , and  $U = \{\ell + 1, \dots, m\}$ . Suppose that  $\langle \langle t, e \rangle, \langle b^1, \dots, b^\ell, b^{\ell+1}, \dots, b^m, e' \rangle \rangle \in \preceq_W \otimes U$ . Due to the definition of operation  $\otimes$ , it is the case that  $\langle t, e \rangle \preceq_W \langle b^1, \dots, b^\ell, e' \rangle$ . 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 definition for system  $\mathcal{S}(W)$ . Clearly then  $\langle t, e \rangle \preceq_{W \cup U} \langle b^1, \dots, b^\ell, b^{\ell+1}, \dots, b^m, e' \rangle$ . 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 remove capabilities of the other behaviors. This shows that  $\preceq_W \otimes U$  is an ND-simulation 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 of  $\mathcal{T}_t$  by  $\mathcal{T}_{\mathcal{S}(W \cup U)}$ .  $\square$

Let  $F \subseteq W$  be the (indexes of the) behaviors that happen to become permanently unavailable. We denote by  $\preceq_{W \setminus F}$  the relation obtained from  $\preceq_W$  by *projecting out* all (failed) behaviors  $\mathcal{B}_i$  such that  $i \in F$ . Interestingly, the new largest ND-simulation after failure is in fact *contained* in the relation obtained by merely projecting out the failed components from the ND-simulation at hand right before the failure. Specifically, we have:

**Lemma 7.** *Let  $W, F \subseteq \{1, \dots, n\}$  such that  $F \subseteq W$ . Then,*

- $\preceq_{W \setminus F} \subseteq \preceq_{W \setminus F}$ ;
- $\preceq_{W \setminus F}$  may not be an ND-simulation of  $\mathcal{T}_t$  by  $\mathcal{T}_{\mathcal{S}(W \setminus F)}$ .

*Proof.* By Lemma 6,  $\preceq_{(W \setminus F)} \otimes F \subseteq \preceq_{(W \setminus F) \cup F}$ , that is,  $\preceq_{(W \setminus F)} \otimes F \subseteq \preceq_W$ . By projecting out  $F$  on both relations, we get  $\preceq_{(W \setminus F)} \otimes F|_F \subseteq \preceq_{W \setminus F}$ . Then, since  $\preceq \otimes X|_X = \preceq$  for any  $\preceq$  and  $X$ ,  $\preceq_{(W \setminus F)} \subseteq \preceq_{W \setminus F}$  follows.

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

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

**Permanent unavailability** When a behavior becomes permanently unavailable (cf. case (d)), one cannot rely on waiting for it to resume when the composition really needs it. Instead, one can either continue the composition and just “hope for the best,” that is, hope that the failed behavior will not be required, or one can “refine” the current composition to continue guaranteeing the full realization of the target.

The following theorem guides such a refinement. Due to Lemma 7, it is enough just to start the NDSP algorithm from the relation obtained by merely projecting out the failed components, generally resulting in substantially less algorithm iterations. 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 will be considered.

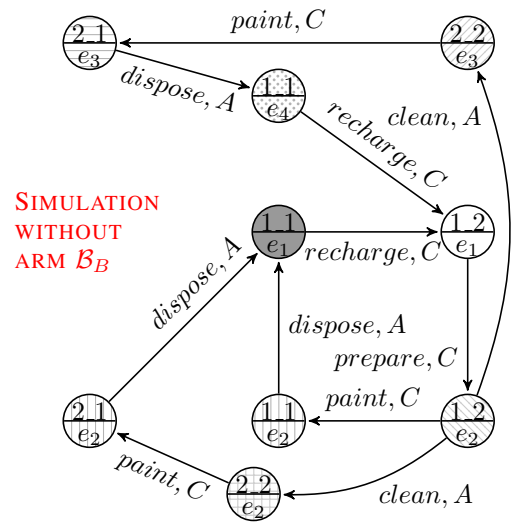
**Theorem 8.** *Let  $\mathcal{S} = \langle \mathcal{B}_1, \dots, \mathcal{B}_n, \mathcal{E} \rangle$  be a system and  $\mathcal{B}_t$  a target behavior over  $\mathcal{E}$ . Let  $W \subseteq \{1, \dots, n\}$  be the (indexes of the) behaviors currently working in  $\mathcal{S}$ , and let  $F \subseteq \{1, \dots, n\}$ , with  $F \subseteq W$ , be the (indexes of the) behaviors that become permanently unavailable. Then,*

$$\preceq_{(W \setminus F)} = \text{NDSP}(\mathcal{T}_t, \mathcal{T}_{\mathcal{S}(W \setminus F)}, \preceq_{W \setminus F}, \beta),$$

for every  $\beta$  such that  $\beta \subseteq \preceq_{(W \setminus F)}$ .

*Proof.* It follows from Lemmas 5 and 7.  $\square$

**Example 7.** Suppose that arm  $\mathcal{B}_T$  is being successfully realized by means of controller  $P_1$ . At some point, however, arm  $\mathcal{B}_B$  suddenly breaks down in state  $b_3$ , just after painting a block. With  $\mathcal{B}_B$  out, controller  $P_1$  cannot guarantee the target anymore. Interestingly, though, controller  $P_2$  can now keep realizing  $\mathcal{B}_T$  from the new (unexpected) sub-system. To handle such failure case, first behavior  $\mathcal{B}_B$  is projected out from the ND-simulation relation  $\preceq_{\{A, B, C\}}$ , thus getting  $\preceq_{\{A, B, C\} \setminus \{B\}}$ . Then, the new largest ND-simulation relation is computed using NDSP and starting from relation  $\preceq_{\{A, B, C\} \setminus \{B\}}$ , thus getting  $\preceq_{\{A, C\}}$ , see picture below.



Observe that tuple  $\langle \langle t_3, e_3 \rangle, \langle \langle a_2, c_1 \rangle, e_3 \rangle \rangle$  would indeed be in relation  $\preceq_{\{A, B, C\} \setminus \{B\}}$ , but it would later be

filtered out by the *NDSP* algorithm—the original tuple  $\langle\langle t_3, e_3 \rangle, \langle\langle a_2, b_2, c_1 \rangle, e_3 \rangle\rangle \in \preceq_{\{A,B,C\}}$  relied on  $\mathcal{B}_B$  for maintaining the ND-simulation. Finally, if arm  $\mathcal{B}_B$  happens to resume, then the CG comes back to the ND-simulation of Figure 2. ■

**Resumed behaviors** Consider now the case in which while behaviors with indexes in  $W$  are currently operating, some behaviors that are supposed to be permanently unavailable, unexpectedly become available again, cf. case (e). Let the indexes of such behaviors be  $U$ , with  $U \cap W = \emptyset$ . Obviously, this could never reduce the capabilities of the whole system, but could enhance it with more choices. To exploit them, one needs to compute the new largest ND-simulation  $\preceq_{(W \cup U)}$ . In doing so, one can leverage on the fact that  $\preceq_{(W \cup U)}$  contains the relation  $\preceq_W \otimes U$  (cf. Lemma 6) by completely avoiding consideration (for potential filtering) of those tuples in  $\preceq_W \otimes U$ , that is, we pass those tuples as the “sure set” to the *NDSP* algorithm.

**Theorem 9.** *Let  $\mathcal{S} = \langle \mathcal{B}_1, \dots, \mathcal{B}_n, \mathcal{E} \rangle$  be a system and  $\mathcal{B}_t$  a target behavior over  $\mathcal{E}$ . Let  $W \subseteq \{1, \dots, n\}$  be the (indexes of the) behaviors currently working in  $\mathcal{S}$ , and  $U \subseteq \{1, \dots, n\}$ , with  $W \cap U = \emptyset$ , be the (indexes of the) resumed behaviors. Then,*

$$\preceq_{(W \cup U)} = \text{NDSP}(\mathcal{T}_t, \mathcal{T}_{\mathcal{S}(W \cup U)}, \alpha, \preceq_W \otimes U),$$

for every  $\alpha$  such that  $\preceq_{(W \cup U)} \subseteq \alpha$ .

*Proof.* It follows from Lemmas 5 and 6. □

Observe that  $U$  could even include new behaviors not included in  $\{1, \dots, n\}$ —the thesis of Lemma 6 would still hold.

**Reusing previous computed ND-simulations** Suppose that we have already computed and stored the ND-simulations for the sets of indexes in  $\mathcal{W}$  (of course,  $\{1, \dots, n\} \in \mathcal{W}$ ), and suppose we are to compute the ND-simulation  $\preceq_W$  for  $W \notin \mathcal{W}$ . Let us then define:

$$\bar{\alpha} = \bigcap_{\{W' \in \supset_W^{\mathcal{W}}\}} \preceq_{W'} |_{(W' \setminus W)};$$

$$\bar{\beta} = \bigcup_{\{W' \in \subseteq_W^{\mathcal{W}}\}} \preceq_{W'} \otimes (W \setminus W');$$

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

$$\begin{aligned} \supset_W^{\mathcal{W}} &= \{W' \in \mathcal{W} \mid W \subseteq W' \wedge \forall V \in \mathcal{W}. W \subseteq V \rightarrow V \not\subseteq W'\}; \\ \subseteq_W^{\mathcal{W}} &= \{W' \in \mathcal{W} \mid W' \subseteq W \wedge \forall V \in \mathcal{W}. V \subseteq W \rightarrow W' \not\subseteq V'\}. \end{aligned}$$

Then, by using the above Theorems 8 and 9 we get that:

$$\preceq_W = \text{NDSP}(\mathcal{T}_t, \mathcal{T}_{\mathcal{S}(W)}, \bar{\alpha}, \bar{\beta}).$$

Notice that by using  $\text{NDSP}(\mathcal{T}_t, \mathcal{T}_{\mathcal{S}}, \bar{\alpha}, \bar{\beta})$  to compute  $\preceq_W$ , we *maximally reuse the computations already done* to devise other ND-simulations. Of course, once we have computed  $\preceq_W$ , we can immediately compute *CGP<sub>jit</sub>* on-the-fly as before.

## Conclusions

In this paper, we presented a simulation-based technique for behavior composition (De Giacomo & Sardina 2007) which radically departs from previous approaches. Such technique is a substantial improvement over the previous ones from the complexity-theoretic perspective (it is exponential in the number, and not the size, of the available behaviors). More importantly, it produces flexible solutions that are ready to handle exceptional circumstances unforeseen at specification time, avoiding re-planning altogether in significant cases and bounding it in others.

We remark that the proposed technique is quite suitable for optimized implementations. First, optimized techniques exist for computing simulation, such as those in (Henzinger, Henzinger, & Kopke 1995; Tan & Cleaveland 2001; Gentilini, Piazza, & Policriti 2003), and implemented in systems such as CWB-NC.<sup>5</sup> Second, it is known that a relationship exists between simulation and checking invariance properties in temporal-logic-based model checkers and synthesis systems, see e.g., (Vardi & Fisler 1999; Asarin *et al.* 1998). In fact, we are currently implementing the technique proposed in this paper using the synthesis system TLV,<sup>6</sup> see e.g., (Piterman, Pnueli, & Sa’ar 2006). Another option would be to exploit ATL-based verifiers, such as Mocha,<sup>7</sup> which can check game-structures for properties such as invariants, and extract winning strategies for them, see e.g., (Alura, Henzinger, & Kupferman 2002).

the

The kind of failures we have considered here can be seen as core forms of breach-of-contract with respect to the specification. Of course other forms of failures are possible (Tripathi & Miller 2001; Pettersson 2005; Marin, Bertier, & Sens 2003), but they essentially assume more information at hand upon a failure, e.g., a module may state unavailability duration and/or the state, or possible states, it will join back. Moreover, such additional information may be of statistical or probabilistic nature. Exploiting such information for failure reaction opens interesting directions for future work.

require more information unavailability duration and/or the state, or possible back, cf Conclusions.

## Acknowledgments

The authors would like to thank the anonymous reviewers for their interesting comments. The first author was supported by the Australian Research Council and Agent Oriented Software (grant LP0560702), and the National Science and Engineering Research Council of Canada under a PDF fellowship. The other authors were partially supported by the the European FET basic research project FP6-7603 Thinking Ontologies (TONES).

## References

- Alura, R.; Henzinger, T. A.; and Kupferman, O. 2002. Alternating-time temporal logic. *Journal of the ACM* 49(5):672–713.

<sup>5</sup><http://www.cs.sunysb.edu/~cwb/>

<sup>6</sup><http://www.cs.nyu.edu/acsys/tlv/>

<sup>7</sup><http://www.cis.upenn.edu/~mocha/>

- Asarin, E.; Maler, O.; Pnueli, A.; and Sifakis, J. 1998. Controller synthesis for timed automata. In *Proceedings of the IFAC Conference on System Structure and Control*, 469–474.
- Berardi, D.; Calvanese, D.; De Giacomo, G.; Hull, R.; and Mecella, M. 2005. Automatic composition of transition-based semantic web services with messaging. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 613–624.
- Berardi, D.; Cheikh, F.; De Giacomo, G.; and Patrizi, F. 2008. Automatic service composition via simulation. *International Journal of Foundations of Computer Science* 19(2):429–452.
- Clarke, E. M.; Grumberg, O.; and Peled, D. A. 1999. *Model checking*. Cambridge, MA, USA: The MIT Press.
- De Giacomo, G., and 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)*, 1866–1871.
- Gentilini, R.; Piazza, C.; and Policriti, A. 2003. From bisimulation to simulation: Coarsest partition problems. *Journal of Automated Reasoning* 31(1):73–103.
- Georgeff, M. P., and Lansky, A. L. 1987. Reactive reasoning and planning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 677–682.
- Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.
- Grosz, B. J., and Kraus, S. 1996. Collaborative plans for complex group action. *Artificial Intelligence Journal* 86(2):269–357.
- Harel, D.; Kozen, D.; and Tiuryn, J. 2000. *Dynamic Logic*. The MIT Press.
- Henzinger, M. R.; Henzinger, T. A.; and Kopke, P. W. 1995. Computing simulations on finite and infinite graphs. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS)*, 453–462.
- Katz, M. J., and Rosenschein, J. S. 1993. The generation and execution of plans for multiple agents. *Computers and Artificial Intelligence* 12(1):5–35.
- Kupferman, O., and Vardi, M. Y. 1996. Module checking. In *Proceedings of the 8th International Conference on Computer Aided Verification (CAV)*, 75–86. London, UK: Springer-Verlag.
- Marin, O.; Bertier, M.; and Sens, P. 2003. Darx - A framework for the fault tolerant support of agent software. In *Proceedings of the 14th IEEE International Symposium on Software Reliability Engineering (ISSRE)*.
- McIlraith, S., and Son, T. C. 2002. Adapting Golog for programming the semantic web. In *Principles of Knowledge Representation and Reasoning (KR)*, 482–493.
- Milner, R. 1971. An algebraic definition of simulation between programs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 481–489.
- Muscholl, A., and Walukiewicz, I. 2007. A lower bound on web services composition. In *Proceedings of the 10th Int. Conf. on Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 4423 of *Lecture Notes in Computer Science (LNCS)*. Springer.
- Pettersson, O. 2005. Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems* 53(2):73–88.
- Piterman, N.; Pnueli, A.; and Sa’ar, Y. 2006. Synthesis of reactive(1) designs. In Emerson, E. A., and Namjoshi, K. S., eds., *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3855 of *Lecture Notes in Computer Science (LNCS)*, 364–380. Charleston, SC, USA: Springer.
- Pnueli, A., and Rosner, R. 1989. On the synthesis of a reactive module. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 179–190.
- Sardina, S.; Patrizi, F.; and De Giacomo, G. 2007. Automatic synthesis of a global behavior from multiple distributed behaviors. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1063–1069.
- Tan, L., and Cleaveland, R. 2001. Simulation revisited. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *LNCS*, 480–495.
- Tripathi, A., and Miller, R. 2001. Exception handling in agent-oriented systems. In *Exception Handling*, volume 2022 of *Lecture Notes in Computer Science (LNCS)*, 128–146. Springer-Verlag.
- Vardi, M., and Fisler, K. 1999. Bisimulation and model checking. In *Proceedings of the Conference on Correct Hardware Design and Verification Methods (CHARME)*, 338–341.



**2008**

This paper was published in the *Proceedings, Eleventh International Conference on Principles of Knowledge Representation and Reasoning*, edited by Gerhard Brewka and Jérôme Lang (Menlo Park, California: AAAI Press).

The 2008 KR conference was held in Sydney, Australia, 16–19 September, 2008.