Automatic Synthesis of New Behaviors from a Library of Available Behaviors

Giuseppe De Giacomo
Dipartimento di Informatica e Sistemistica
Università di Roma “La Sapienza”
Roma, Italy
degiacomo@dis.uniroma1.it

Sebastian Sardina
Department of Computer Science
RMIT University
Melbourne, Australia
ssardina@cs.rmit.edu.au

Abstract

We consider the problem of synthesizing a fully controllable target behavior from a set of available partially controllable behaviors that are to execute within a shared partially predictable, but fully observable, environment. Behaviors are represented with a sort of nondeterministic transition systems, whose transitions are conditioned on the current state of the environment, also represented as a nondeterministic finite transition system. On the other hand, the target behavior is assumed to be fully deterministic and stands for the behavior that the system as a whole needs to guarantee. We formally define the problem within an abstract framework, characterize its computational complexity, and propose a solution by appealing to satisfiability in Propositional Dynamic Logic, which is indeed optimal with respect to computational complexity. We claim that this problem, while novel to the best of our knowledge, can be instantiated to multiple specific settings in different contexts and can thus be linked to different research areas of AI, including agent-oriented programming and cognitive robotics, control, multi-agent coordination, plan integration, and automatic web-service composition.

1 Introduction

Imagine an intelligent system built from a variety of different components or devices operating (that is, performing actions) on the same shared environment. For example, consider the case of a blocks world scenario in which different kind or versions of robotic arms can move, paint, clean, dispose blocks, and so on. Next, imagine that the central system possesses some information about the logic of these components or devices—e.g., a particular arm type can paint or clean a block after picking it up, but another type of arm is only capable of painting blocks. However, the knowledge about these devices is *partial* in that their internal logic may potentially expose nondeterministic features. For instance, the system may have no information regarding the use of paint in a painting arm, and thus, it is always unable to predict whether the arm will run out of paint after a painting action. If it does, then the arm cannot be used to paint again until it is recharged. Nonetheless, the central system has *full observability* on the state of the devices in question: after the arm is used to paint a block, the central system will come to know whether such arm ran out of paint or not (maybe via a sensor light). Finally, the central system is also (partially) knowledgeable about the shared environment, in which all the components ought to execute. As with the devices, the system possesses a nondeterministic model of the real environment and has full observability on it too. The question then is: can the central system (always) guarantee a specific *deterministic* overall behavior by (partially) controlling the available devices or components in a step-by-step manner, that is, by instructing them on which action to execute next and observing, afterwards, the outcome in the device used as well as in the environment?

It is not hard to see that the above setting can be recast in a variety of forms. In the context of agent-oriented programming [6; 4], an intelligent agent may control the execution of a set of predefined nondeterministic agent plans in order to realize a desired deterministic plan-intention, which in turn, may have been obtained via planning [13; 8]. Similarly, when it comes to plan coordination [9], one can think of coordinating or merging approximate models of multiple agents' plans into a global fully deterministic multi-agent plan. Also, in the context of web-service composition [12; 1; 3],[1] existing web services may be composed on the Internet so as to implement a new complex web service. In all these cases, the challenge is to automatically synthesize a *fully controllable* module from a set of *partially controllable* existing modules executing in a *partially predictable* environment.

Under this setting, the technical contributions of this paper are threefold. First, we formally define the problem within an abstract framework. Second, we devise a formal technique to perform *automatic synthesis of the fully controllable module*. We show that the technique proposed is sound, complete, and terminating. Lastly, we characterize the computational complexity of the problem and show that the proposed technique is optimal with respect to computational complexity.

2 The setting

Let us start by defining the synthesis problem [15] that is the subject of our research. To that end, we develop an abstract framework based on (sort of) finite state transition systems.

[1]In particular, [3] can be seen as a simpler variant of the setting studied here, with no environment and deterministic behaviors.

Environment We assume to have a shared observable environment, which provides an abstract account of the observable effect and preconditions of actions. In giving such an account, we take into consideration that, in general, we have incomplete information about the actual effects and preconditions of actions. Thus, we allow the observable environment to be *nondeterministic* in general. In that way, the incomplete information on the actual world shows up as nondeterminism in our formalization.

Formally, an *environment* $\mathcal{E} = (\mathcal{A}, E, e_0, \delta_\mathcal{E})$ is characterized by the following five entities:

- $\mathcal{A}$ is a finite set of shared actions;
- E is a finite set of possible environment states;
- $e_0 \in E$ is the initial state of the environment;
- $\delta_\mathcal{E} \subseteq E \times \mathcal{A} \times E$ is the transition relation among states: $\delta_\mathcal{E}(e, a, e')$ holds when action a performed in state e may lead the environment to a successor state e'.

Note that our notion of the environment shares a lot of similarities with the so-called "transition system" in action languages [5]. Indeed, one can think of using that kind of formalism to compactly represent the environment in our setting.

Behavior A behavior is essentially a program for an agent or the logic of some available device or component. Such a program however leaves the selection of the action to perform next to the agent itself. More precisely, at each step the program presents to the agent a choice of available actions; the agent selects one of them; the action is executed; and so on.

Obviously, behaviors are not intended to be executed on their own, but they are executed in the environment (cf. above). Hence, we equip them with the ability of testing conditions (i.e., guards) on the environment when needed.

Formally, a *behavior* $\mathcal{B} = (S, s_0, G, \delta_\mathcal{B}, F)$ over an environment $\mathcal{E}$, is a characterized by the following entities:

- S is a finite set of behavior states;
- $s_0 \in S$ is the single initial state of the behavior;
- G is a set of guards, which are boolean functions $g : E \to \{\texttt{true}, \texttt{false}\}$, where E is the set of the environment states of $\mathcal{E}$;
- $\delta_\mathcal{B} \subseteq S \times G \times \mathcal{A} \times S$ is the behavior transition relation, where $\mathcal{A}$ is the set of actions of $\mathcal{E}$—we call the $G \times \mathcal{A}$ components of such tuples, the *label of the transition*;
- finally, $F \subseteq S$ is the set of final states of the behavior, that is, the states in which the behavior may stop executing, but does not necessarily have to.

Observe that, in general, behaviors are *nondeterministic* in the sense that they may allow more than one transition with the same action a and compatible guards evaluating to the same truth value.[2] As a result, the central system, when making its choice of which action to execute next, cannot be certain of which choices it will have later on, since that depends on what transition is actually executed. In other words, nondeterministic behaviors are only partially controllable.

[2]Note that this kind of nondeterminism is of a *devilish* nature, so as to capture the idea that through the choice of actions alone one cannot fully control the behavior.

We say that a behavior $\mathcal{B} = (S, s_0, G, \delta_\mathcal{B}, F)$, over the environment $\mathcal{E}$, is *deterministic* if there is no environment state e of $\mathcal{E}$ for which there exist two distinct transitions (s, g_1, a, s_1) and (s, g_2, a, s_2) in $\delta_\mathcal{B}$ such that $g_1(e) = g_2(e)$. Notice that given a state in a deterministic behavior and a legal action in that state, we always know exactly which is *the* next state of the behavior. In other words, deterministic behaviors are fully controllable through the selection of the action to perform next, while this is not the case for nondeterministic ones.

Runs and traces Given a behavior $\mathcal{B} = (S, s_0, G, \delta, F)$ and an environment $\mathcal{E} = (\mathcal{A}, E, e_0, \delta_\mathcal{E})$, we define the *runs of $\mathcal{B}$ on $\mathcal{E}$* as, possibly infinite, alternating sequences of the following form: $(s^0, e^0)a^1(s^1, e^1)a^2 \cdots$, where $s^0 = s_0$ and $e^0 = e_0$, and for every i we have that $(s^i, e^i)a^{i+1}(s^{i+1}, e^{i+1})$ is such that:

- there exists a transition $(e^i, a^{i+1}, e^{i+1}) \in \delta_\mathcal{E}$; and
- there exists a transition $(s^i, g^{i+1}, a^{i+1}, s^{i+1}) \in \delta_\mathcal{B}$, such that $g^{i+1}(e_i) = \texttt{true}$.

Moreover if the run is finite, that is, it is of the form $(s^0, e^0)a^1 \cdots a^\ell(s^\ell, e^\ell)$, then $s^\ell \in F$.

Apart from runs, we are also interested in traces generated by the behavior. A *trace* is a sequence of pairs (g, a), where $g \in G$ is a guard of $\mathcal{B}$ and $a \in \mathcal{A}$ is an action, of the form $t = (g^1, a^1) \cdot (g^2, a^2) \cdots$ such that there exists a run $(s^0, e^0)a^1(s^1, e^1)a^2 \cdots$, as above, where $g^i(e_{i-1}) = \texttt{true}$ for all i. If the trace $t = (g^1, a^1) \cdots (g^\ell, a^\ell)$ is finite, then there exists a finite run $(s^0, e^0)a^1 \cdots a^\ell(s^\ell, e^\ell)$ with $s^\ell \in F$.

The *traces of the deterministic behaviors* are of particular interest: any initial fragment of a trace leads to a single state in the behavior. In a sense, the deterministic behavior itself can be seen as a specification of a set of traces.

The system A *system* $\mathcal{S} = (\mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E})$ is formed by an observable environment $\mathcal{E}$ and n predefined nondeterministic behaviors $\mathcal{B}_i$, called the *available behaviors*. A *system configuration* is a tuple $(s_1, \ldots, s_n, e)$ denoting a snapshot of the system: behavior $\mathcal{B}_i$ is in state s_i and the environment $\mathcal{E}$ is in state e. We assume that the system has a specific component, called the *scheduler*, that is able to activate, stop, and resume the behaviors at each point in time.

The problem The problem we are interested in is the following: given a system $\mathcal{S} = (\mathcal{B}_1, \ldots, B_n, \mathcal{E})$ and a *deterministic* behavior, called the *target behavior* $\mathcal{B}_0$ over $\mathcal{E}$, *synthesize a program for the scheduler such that the target behavior is realized by suitably scheduling the available behaviors*.

In order to make this precise, we need to clarify which are the basic capabilities of the scheduler. The scheduler has the ability of activating-resuming one[3] of the many available behaviors by instructing it to execute an action among those that are possible in its current state (taking into account the environment). Also, the scheduler has the ability of keeping track (at runtime) of the current state of each available behavior.

[3]For simplicity, we assume that the scheduler activates/resumes only one behavior at each step, though our approach can be extended to the case where more available services are activated at each step.

Technically, such a capability is called *full observability* on the states of the available behaviors. Although other choices are possible [1], full observability is the natural choice in this context, since the available behaviors are already suitable abstractions for the *actual* behaviors, and hence there is no reason to make its states partially unobservable: if details have to be hidden, this can be done directly within the abstract behavior exposed, possibly making use of nondeterminism.

We are now ready to formally define our synthesis problem. Let the system be $\mathcal{S} = (\mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E})$, where $\mathcal{E} = (\mathcal{A}, E, e_0, \delta_{\mathcal{E}})$ is the environment and $\mathcal{B}_i = (S_i, s_{i0}, G_i, \delta_i, F_i)$ are the available behaviors. Let the target behavior be $\mathcal{B}_0 = (S_0, s_{00}, \delta_0, F_0)$.

A *system history* is an alternating sequence of system configurations and actions of the form $h = (s_1^0, \ldots, s_n^0, e^0) \cdot a^1 \cdot (s_1^1, \ldots, s_n^1, e^1) \cdots (s_1^{\ell-1}, \ldots, s_n^{\ell-1}, e^{\ell}) \cdot a^{\ell} \cdot (s_1^{\ell}, \ldots, s_n^{\ell}, e^{\ell})$ such that the following constraints hold:

- $s_i^0 = s_{i0}$ for $i \in \{1, \ldots, n\}$, that is, each behavior starts in its initial state;
- $e^0 = e_0$, that is, the environment starts in its initial state;
- at each step $0 \leq k \leq \ell$, there exists an $i \in \{1, \ldots, n\}$ such that $(s_i^k, g_i^{k+1}, a^{k+1}, s_i^{k+1}) \in \delta_i$ and for all $j \neq i$, $s_j^{k+1} = s_j^k$, that is, at each step in the history, only one of the behaviors, namely $\mathcal{B}_i$, has made a (legal) transition (according to its transition relation δ_i), while the other ones have remained still;
- at each step $0 \leq k \leq \ell$, we have that $(e^k, a^{k+1}, e^{k+1}) \in \delta_{\mathcal{E}}$, that is, the environment has also made a legal transition according to its transition relation.

A *scheduler program* is a function $P : \mathcal{H} \times \mathcal{A} \rightarrow \{1, \ldots, n, u\}$ that, given a history $h \in \mathcal{H}$ (where $\mathcal{H}$ is the set of all system histories as defined above) and an action $a \in \mathcal{A}$ to perform, returns the behavior (actually the behavior index) that is scheduled to perform the action. Observe that such a function may also return a special value u, for "undefined." This is a technical convenience to make P a total function returning values even for histories that are not of interest, or for actions that no behavior can perform after a given history.

Next, we define when a scheduler program is a composition that realizes the target behavior—a solution to the problem. First, we point out that, because the target behavior is a deterministic transition system, its behavior is completely characterized by the set of its traces, that is, by the set of infinite sequences of actions that are faithful to its transitions, and of finite sequences that in addition lead to a final state.

So, given a trace $t = (g^1, a^1) \cdot (g^2, a^2) \cdots$ of the target behavior, we say that a *scheduler program P realizes the trace t* iff for all ℓ and for all system histories $h \in \mathcal{H}_{t,P}^{\ell}$ ($\mathcal{H}_{t,P}^{\ell}$ is defined below) such that $g^{\ell+1}(e_h^{\ell}) = \texttt{true}$ in the last environment state e_h^{ℓ} of h, we have that $P(h, a^{\ell+1}) \neq u$ and $\mathcal{H}_{t,P}^{\ell+1}$ is nonempty, where the set of system histories $\mathcal{H}_{t,P}^{\ell}$ is inductively defined as follows:

- $\mathcal{H}_{t,P}^0 = \{(s_{10}, \ldots, s_{n0}, e_0)\}$;
- $\mathcal{H}_{t,P}^{\ell+1}$ is the set of $\ell + 1$-length system histories of the form $h \cdot a^{\ell+1} \cdot (s_1^{\ell+1}, \ldots, s_n^{\ell+1}, e^{\ell+1})$ such that:
  - $h \in \mathcal{H}_{t,P}^{\ell}$, where $(s_1^{\ell}, \ldots, s_n^{\ell}, e^{\ell})$ is the last system configuration in h;
  - $a^{\ell+1}$ is an action such that $P(h, a^{\ell+1}) = i$, with $i \neq u$, that is, the scheduler states that action $a^{\ell+1}$ at system history h should be executed in behavior $\mathcal{B}_i$;
  - $(s_i^{\ell}, g, a^{\ell+1}, s_i') \in \delta_i$ with $g(e^{\ell}) = \texttt{true}$, that is, behavior $\mathcal{B}_i$ *may* evolve from its current state s_i^{ℓ} to state s_i' w.r.t. the (current) environment state e^{ℓ};
  - $(e^{\ell}, a^{\ell+1}, e^{\ell+1}) \in \delta_{\mathcal{E}}$, that is, the environment may evolve from its current state e^{ℓ} to state $e^{\ell+1}$;
  - $s_i^{\ell+1} = s_i'$ and $s_j^{\ell+1} = s_j^{\ell}$, for $j \neq i$, that is, only behavior $\mathcal{B}_i$ is allowed to perform a step.

Moreover, as before, if a trace is finite and ends after m actions, and all along all its guards are satisfied, we have that all histories in $\mathcal{H}_{t,P}^m$ end with all behaviors in a final state. Finally, we say that a *scheduler program P realizes the target behavior* $\mathcal{B}_0$ if it realizes all its traces.[4]

In order to understand the above definitions, let us observe that, intuitively, the scheduler program realizes a trace if, as long as the guards in the trace are satisfied, it can choose at every step an available behavior to perform the requested action. If at a certain point a guard in the trace is not satisfied in the current environment state, then we may consider the trace finished (even if it is not in a final state). As before, however, because the available behaviors nondeterministically choose what transition to actually perform when executing an action, the scheduler program must be such that the scheduler will always be able to continue with the execution of the next action no matter how both the activated behavior and the environment evolve after each step. Finally, the last requirement makes sure that all available behaviors are left in a final state when a finite trace reaches its end with all guards satisfied.

Observe that, in general, a scheduler program could require infinite states. However, we will show later that if a scheduler that realizes the target behavior does exist, then there exists one with a *finite* number states. Note also that the scheduler has to observe the states of the available behaviors in order to decide which behavior to select next (for a given action requested by the target behavior). This makes these scheduler programs akin to an advanced form of conditional plans [16].

3 An example

We now come back to our original blocks world example in order to illustrate the abstract framework developed in the previous section. The complete scenario is depicted in Figure 1. The aim of the whole system is to paint existing blocks. Blocks can be processed by cleaning and painting them. Before processing a block, though, it is necessary to prepare it, for example, by moving it to a special processing location. Furthermore, only after a block has been disposed, can another block be prepared for processing. Finally, cleaning and painting may, sometimes, require resources, namely, water and paint, respectively: we assume there are two tanks, for water and paint, respectively, and that both are recharged simultaneously by pressing a recharging button.

[4]Recall that because the target behavior is deterministic, it can be seen as a specification of a, possibly infinite, set of traces.

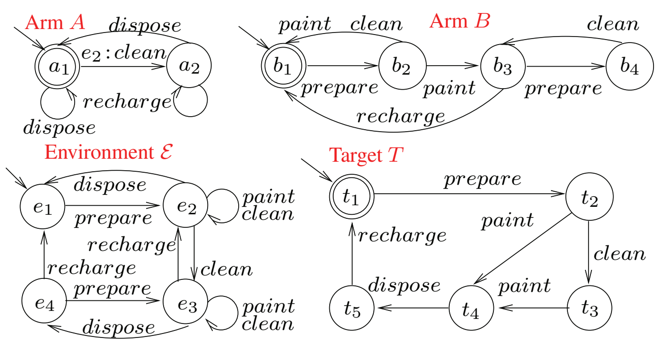

Figure 1: A painting blocks world scenario.

Now, the desired target behavior T that we want to achieve is as follows. First, a block is prepared for processing. Then, the block in question can either be painted right away or painted after being cleaned—some (dirty) blocks may need to be washed before being painted. Note that the decision of whether a block needs to be cleaned lays outside of our framework. After a block has been painted, it is disposed. Finally, the recharging button is pushed. See that this target behavior is "conservative," in that it always recharges the tanks after a block has been processed.

One can think of the above target behavior as *the* arm that one would like to have. However, such arm does not exist in reality. Instead, there are only two different arms available. The first arm A, a cleaning-disposing arm, is able to clean and dispose blocks. The second arm B is capable of preparing, cleaning, and painting blocks. Both arms are able to press the recharge button to refill the tanks.

So, the system is composed of the environment $\mathcal{E}$ and the two arms A and B shown in Figure 1. Let us now note a few interesting points. First, the environment $\mathcal{E}$ provides the (general) preconditions of actions in the domain (e.g., *dispose* can only be executed after a *prepare* action). The environment also includes some information about the water tank: in e_1 and e_2, the water tank is not empty; and in e_3 and e_4, the water tank is indeed empty. Notice that it is still conceivable to clean a block in state e_3, by some method that does not rely on water. However, because arm A does use water to clean blocks, it can only do it when the environment is in fact in state e_2. Second, we observe that whereas only the second arm can prepare a block, only the first arm can dispose a block. Lastly, the most interesting remark comes from arm B's internal logic, for which the system only has partial information. After painting a block, arm B may evolve nondeterministically to two different states: b_1 or b_3. Intuitively, the arm evolves to state b_3 as soon as the paint tank becomes empty; otherwise the arm evolves to state b_1. Once the arm runs out of paint, it can only clean blocks until the tanks are eventually recharged. Notice also that, unlike arm A, arm B does not require the environment to be in state e_2 to clean a block, as its cleaning mechanism does not rely on water.

We aim to automatically synthesize a scheduler program so as to realize the target behavior (T) by making use of the available behaviors (arms A and B) and considering the environment ($\mathcal{E}$). See the next section.

4 The synthesis technique

We are now ready to investigate how to check for the existence of a scheduler program that realizes the target behavior, and even more, how to actually compute it. We start with some preliminary considerations on the computational complexity that we should expect. The result by Muscholl and Walukiewicz in [14], which can be easily rephrased in our framework, gives us an EXPTIME lowerbound.

More precisely, let us call *empty environment* any environment of the form $\mathcal{E} = (\mathcal{A}, E, e_o, \delta_{\mathcal{E}})$, where $E = \{e_o\}$ (i.e., the (observable) environment has a single state), and $\delta_{\mathcal{E}} = \{(e_0, a, e_0) \mid a \in \mathcal{A}\}$, (i.e., there are no preconditions on actions, nor actions have any effect on the environment). Also, let us call *deterministic guardless behavior* any behavior of the form $\mathcal{B} = (S, s_0, \{g_{\texttt{true}}\}, \delta_{\mathcal{B}}, F)$, where $g_{\texttt{true}}$ is the constant function returning always `true`, and $\delta_{\mathcal{B}}$ is functional, i.e., for each state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$, there is at most one state s' such that $(s, a, s') \in \delta_{\mathcal{B}}$. Then, Muscholl and Walukiewicz's result can be stated as follows.

Theorem 4.1 (Muscholl & Walukiewicz 2005) *Checking the existence of a scheduler program that realizes a target deterministic guardless behavior in a system consisting of an empty environment and a set of available deterministic guardless behaviors is EXPTIME-hard.*[5]

Hence, checking the existence of a scheduler program in our general framework is at least exponential time. Next, we show that the problem is actually EXPTIME-complete, by resorting to a reduction to satisfiability in Propositional Dynamic Logic (PDL). Moreover, such a reduction can be exploited to generate the actual scheduler program which will be finite. In doing this, we extend the approach in [3], originally developed in the context of service composition to deal with empty environments and deterministic guardless behaviors. Dealing with a non-trivial environment, and especially dealing with the nondeterminism of the available behaviors, requires solving same subtle points that reflects the sophisticated notion of scheduler program that is needed for that.

4.1 Propositional Dynamic Logic

Propositional Dynamic Logic (PDL) is a modal logic specifically developed for reasoning about computer programs [7]. Syntactically, PDL formulas are built from a set $\mathcal{P}$ of atomic propositions and a set Σ of atomic actions:

$$\begin{array}{lll} \phi & \longrightarrow & P \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi \rightarrow \phi' \mid \\ & & \langle r \rangle \phi \mid [r]\phi \mid \texttt{true} \mid \texttt{false}, \\ r & \longrightarrow & a \mid r_1 \cup r_2 \mid r_1; r_2 \mid r^* \mid \phi?, \end{array}$$

where P is an atomic proposition in $\mathcal{P}$, r is a regular expression over the set of actions in Σ, and a is an atomic action in Σ. That is, PDL formulas are composed from atomic propositions by applying arbitrary propositional connectives, and modal operators $\langle r \rangle \phi$ and $[r]\phi$. Formula $\langle r \rangle \phi$ means that there exists an execution of r (i.e., a sequence of actions conforming to the regular expression r) reaching a state where ϕ

[5]In fact, Muscholl and Walukiewicz show EXPTIME-hardness in the setting of service composition studied in [2; 3], where all available services are deterministic [14].

holds; and formula $[r]\phi$ is intended to mean that all terminating executions of r reach a state where ϕ holds.

A PDL formula ϕ is satisfiable if there exists a model for ϕ—an interpretation where ϕ is true. Checking satisfiability of a PDL formula is EXPTIME-complete [7].

PDL enjoys two properties that are of particular interest for us [7]. The first is the *tree model property*, which says that every model of a formula can be unwound to a, possibly infinite, tree-shaped model (considering domain elements as nodes and partial functions interpreting actions as edges). The second is the *small model property*, which says that every satisfiable formula admits a finite model whose size (in particular the number of domain elements) is at most exponential in the size of the formula itself.

4.2 Reduction to PDL

Let $\mathcal{S}=(\mathcal{B}_1,\ldots,\mathcal{B}_n,\mathcal{E})$ be a system, where $\mathcal{E}=(\mathcal{A},E,e_0,\delta_{\mathcal{E}})$ is the environment and $\mathcal{B}_i=(S_i,s_{i0},G_i,\delta_i,F_i)$ are the available behaviors over $\mathcal{E}$, and let $\mathcal{B}_0=(S_0,s_{00},\delta_0,F_0)$ be the target behavior (over $\mathcal{E}$ as well). Then, we build a PDL formula Φ to check for satisfiability as follows.

As actions in Φ, we have the actions $\mathcal{A}$ in $\mathcal{E}$. As atomic propositions, we have:[6]

- one atomic proposition e for each state e of $\mathcal{E}$, which intuitively denotes that $\mathcal{E}$ is in state e;
- one atomic proposition s for each state s of $\mathcal{B}_i$, for $i\in\{0,1,\ldots,n\}$, denoting that $\mathcal{B}_i$ is in state s;
- atomic propositions F_i, for $i\in\{0,1,\ldots n\}$, denoting that $\mathcal{B}_i$ is in a final state;
- atomic propositions $exec_{ia}$, for $i\in\{1,\ldots n\}$ and $a\in\mathcal{A}$, denoting that a will be executed next by behavior $\mathcal{B}_i$;
- one atomic proposition $undef$ denoting that we reached a situation where the scheduler can be left undefined.

Let us now build the formula Φ. For representing the transitions of the target behavior $\mathcal{B}_0$, we construct a formula ϕ_0 as the conjunction of (for each s of $\mathcal{B}_0$ and e of $\mathcal{E}$):

- $s\wedge e\rightarrow\langle a\rangle\mathtt{true}\wedge[a]s'$, for each transition $(s,g,a,s')\in\delta_0$ such that $g(e)=\mathtt{true}$, encoding that the target behavior can do an a-transition, whose guard g is satisfied, by going from state s to state s';
- $s\wedge e\rightarrow[a]undef$, for each a such that for no g and s' we have $(s,g,a,s')\in\delta_0$ with $g(e)=\mathtt{true}$. This takes into account that the target behavior cannot perform an a-transition.

For representing the transitions of each available behavior $\mathcal{B}_i$, we construct a formula ϕ_i as the conjunction of:

- the formula
$$s\wedge e\wedge exec_{ia}\rightarrow\bigwedge_{(s',e')\in\Delta}(\langle a\rangle(s'\wedge e'))\wedge[a](\bigvee_{(s',e')\in\Delta}(s'\wedge e')),$$
where $\Delta=\{(s',e')\mid(s,g,a,s')\in\delta_i,\ g(e)=\mathtt{true},\ (e,a,e')\in\delta_{\mathcal{E}}\}$, for each environment state e, each s of $\mathcal{B}_i$, and each $a\in\mathcal{A}$. These assertions encode that if the current environment state is e and the available behavior $\mathcal{B}_i$ is in state s and is selected for the execution of an action a (i.e., $exec_{ia}$ is true), then for each possible a-transition of $\mathcal{B}_i$ with its guard true in e and of $\mathcal{E}$, we have a possible a-successor in the models of Φ;
- $s\wedge e\wedge exec_{ia}\rightarrow[a]\mathtt{false}$, for each environment state e of $\mathcal{E}$, and each state s of $\mathcal{B}_i$ such that for no g, s', and e', we have that $(s,g,a,s')\in\delta_i$ with $g(e)=\mathtt{true}$ and $(e,a,e')\in\delta_{\mathcal{E}}$. This states that if the current environment state is e and $\mathcal{B}_i$, whose current state is s, is selected for the execution of a, but a cannot be executed by $\mathcal{B}_i$ in e, then there is no a-successor in the models of Φ;
- $s\wedge\neg exec_{ia}\rightarrow[a]s$, for each state s of $\mathcal{B}_i$ and each action a. This assertion encodes that if behavior $\mathcal{B}_i$ is in state s and is not selected for the execution of a, then if a is performed (by some other available behavior), $\mathcal{B}_i$ does not change state.

In addition, we have the formula ϕ_{add}, of general constraints, obtained as the conjunction of:

- $s\rightarrow\neg s'$, for all pairs of states s,s' of $\mathcal{B}_i$, and for $i\in\{0,1,\ldots,n\}$, stating that propositions representing different states of $\mathcal{B}_i$ are disjoint;
- $F_i\leftrightarrow\bigvee_{s\in F_i}s$, for $i\in\{0,1,,\ldots,n\}$, highlighting the final states of $\mathcal{B}_i$;
- $undef\rightarrow[a]undef$, for each action $a\in\Sigma$, stating that once a situation is reached where $undef$ holds, then $undef$ holds also in all successor situations;
- $\neg undef\wedge\langle a\rangle\mathtt{true}\rightarrow\bigvee_{i\in\{1,\ldots,n\}}exec_{ia}$, for each $a\in\mathcal{A}$, denoting that, unless $undef$ is true, if a is performed, then at least one of the available behaviors must be selected for the execution of a;
- $exec_{ia}\rightarrow\neg exec_{ja}$ for each $i,j\in\{1,\ldots,n\}$, $i\neq j$, and each $a\in\mathcal{A}$, stating that only one available behavior is selected for the execution of a;
- $F_0\rightarrow\bigwedge_{i\in\{1,\ldots,n\}}F_i$, stating that when the target behavior is in a final state, so are all the available behaviors.

Finally, we define Φ as

$$Init\wedge[\mathbf{u}](\phi_0\wedge\bigwedge_{i\in\{1,\ldots,n\}}\phi_i\wedge\phi_{add}),$$

where *Init* stands for $e_0\wedge s_{00}\wedge s_{01}\wedge\cdots\wedge s_{0n}$, and represents the initial state of the environment $\mathcal{E}$ and of all behaviors $\mathcal{B}_i$ (including the target), and $\mathbf{u}=(\bigcup_{a\in\Sigma}a)^*$, which acts as the *master modality* [7], is used to force $\phi_0\wedge\bigwedge_{i\in\{1,\ldots,n\}}\phi_i\wedge\phi_{add}$ to be true in every point of the model. Note that $\mathbf{u}$ is the only complex program that appears in the PDL formula Φ. We can now state our main result.

Theorem 4.2 *The PDL formula Φ, constructed as above, is satisfiable iff there exists a scheduler program for the system $\mathcal{S}=(\mathcal{B}_1,\ldots,\mathcal{B}_n,\mathcal{E})$ that realizes the target behavior $\mathcal{B}_0$.*

Proof (sketch). "*If*": PDL has the tree-model property. Hence, if Φ is satisfiable then it has a model that is tree shaped. Each node in this tree can be put in correspondence with a history, and from the truth value assignment of the

[6] In this paper, we are not concerned with compact representations of the states of the environment $\mathcal{E}$ and the behaviors $\mathcal{B}_i$. However, we observe that if states are succinctly represented (e.g., in binary format) then, in general, we can exploit such a representation in Φ to get a corresponding compact formula as well.

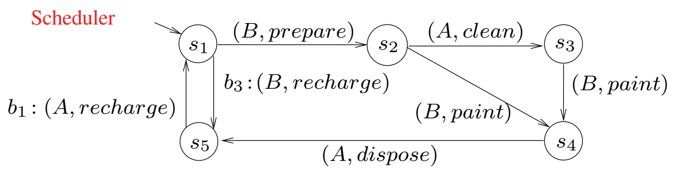

Figure 2: The painting blocks scheduler program. b_1 : $(A, recharge)$ means that arm B is in state b_1 and action $recharged$ is performed by arm A.

propositions $exec_{ia}$ in the node one can reconstruct the scheduler program. "*Only if*": if a scheduler program that realizes $\mathcal{B}_0$ exists, one can use it to build a tree model of Φ. □

Observe that the size of Φ is polynomially with respect to $\mathcal{E}$, $\mathcal{B}_1, \ldots, \mathcal{B}_n$ and $\mathcal{B}_0$. Hence, from the EXPTIME-completeness of satisfiability in PDL and Theorem 4.1, we get the following result:

Theorem 4.3 *Checking the existence of a scheduler program that realizes a target behavior $\mathcal{B}_0$ relative to a system $\mathcal{S} = (\mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E})$ is EXPTIME-complete.*

Finally, by the finite-model property of PDL (i.e., if a formula is satisfiable, it is satisfiable in a model that is at most exponential in the size of the formula), we get a systematic procedure for synthesizing the composition:

Theorem 4.4 *If there exists a scheduler program that realizes a target behavior $\mathcal{B}_0$ relative to a system $(\mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E})$, then there exists one that requires a finite number of states. Moreover such a finite state program can be extracted from a finite model of Φ.*

To end, let us return to our example of Section 3. The corresponding PDDL formula Φ^{blocks}, obtained as explained above, has a a finite model from which we can extract the scheduler program depicted in Figure 2 (after having projected out irrelevant propositions and applied minimization techniques to reduce the number of states). Such scheduler realizes the target behavior T by appealing to the two available arms A and B. As one can observe, even in this simplistic scenario, the existence of a scheduler and its corresponding program is far from trivial. For instance, it is critical to make the correct decision on which machine must recharge the tanks at every step—such a choice would depend on how arm B evolves after painting a block. Also, the scheduler must be able to terminate both arms in their corresponding final states whenever the target behavior is its final state t_1.

5 Conclusion

In this paper, we have looked at automatic synthesis of a fully controllable module from a set of partially controllable existing modules that execute in a partially predictable environment. The kind of problem that we dealt with is clearly a form of reactive process synthesis [15; 10]. Standard techniques for such problems are based on automata on infinite trees, which however relay on critical steps, such as Safra's construction for complementation, that have resisted efficient implementation for a long time [11]. Instead, we based our synthesis on a reduction to satisfiability in PDL [7] with a limited use of the reflexive-transitive-closure operator. Such kind of PDL satisfiability shares the same algorithms that are behind the success of the description logic-based reasoning systems used for OWL,[7] such as FaCT, Racer, and Pellet.[8] Hence, its applicability in our context appears to be quite promising from a practical point of view.

Acknowledgments The authors would like to thank Daniela Berardi, Diego Calvanese, Rick Hull, Massimo Mecella, and Maurizio Lenzerini for discussion and insights on the issues treated in this paper. The first author was partially supported by the the European FET basic research project FP6-7603 Thinking Ontologies (TONES). The second author was supported by the Australian Research Council and Agent Oriented Software (grant LP0560702).

References

[1] D. Berardi, D. Calvanese, G. De Giacomo, R. Hull, and M. Mecella. Automatic composition of transition-based semantic web services with messaging. In *Proc. of VLDB*, 2005.

[2] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-Services that export their behavior. In *Proc. of ICSOC*, pages 43–58, 2003.

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

[4] J. R. Firby. *Adaptive Execution in Complex Dynamic Domains*. PhD thesis, Yale University, 1989.

[5] M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions of AI (ETAI)*, 2:193–210, 1998.

[6] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proc. of AAAI*, pages 677–682, 1987.

[7] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. The MIT Press, 2000.

[8] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence Journal*, 101:99–134, 1998.

[9] M. J. Katz and J. S. Rosenschein. The generation and execution of plans for multiple agents. *Computers and Artificial Intelligence*, 12(1):5–35, 1993.

[10] O. Kupferman and M. Y. Vardi. Synthesis with incomplete information. In *Proc. of ICTL*, 1997.

[11] O. Kupferman and M. Y. Vardi. Safraless decision procedures. In *Proc. of FOCS*, pages 531–542, 2005.

[12] S. McIlraith and T. C. Son. Adapting Golog for programming the semantic web. In *Proc. of KR*, pages 482–493, 2002.

[13] N. Meuleau, L. Peshkin, K.-E. Kim, and L. P. Kaelbling. Learning finite-state controllers for partially observable environments. In *Proc. of UAI*, pages 427–436, 1999.

[14] A. Muscholl and I. Walukiewicz. A lower bound on web services composition. Submitted, 2005.

[15] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. of POPL*, pages 179–190, 1989.

[16] J. Rintanen. Complexity of planning with partial observability. In *Proc. of ICAPS*, pages 345–354, 2004.

[7]`www.omg.org/uml/`

[8]`www.cs.man.ac.uk/~sattler/reasoners.html`