Behavior Composition as Fully Observable Non-Deterministic Planning

Miquel Ramirez, Nitin Yadav and Sebastian Sardina
School of Computer Science and IT
RMIT University
Melbourne, Australia
{miquel.ramirez,nitin.yadav,sebastian.sardina}@rmit.edu.au

Abstract

The behavior composition problem involves the automatic synthesis of a controller able to “realize” (i.e., implement) a target behavior module by suitably coordinating a collection of partially controllable available behaviors. In this paper, we show that the existence of a composition solution amounts to finding a strong cyclic plan for a special non-deterministic planning problem, thus establishing the formal link between the two synthesis tasks. Importantly, our results support the use of non-deterministic planning systems for solving composition problems in an off-the-shelf manner. We then empirically evaluate three state-of-the-art synthesis systems (a domain-independent automated planner and two game solvers based on model checking techniques) on various non-trivial composition instances. Our experiments show that while behavior composition is EXPTIME-complete, the current technology is already able to handle instances of significant complexity. Our work is, as far as we know, the first serious experimental work on behavior composition.

Introduction

The problem of composing (i.e., coordinating) a collection of available behavior modules to implement a desired complex, but non-existent, target behavior module has recently received substantial attention in the literature. The problem, in its various forms, has been studied in many areas of Computer Science, including (web) services (Berardi et al. 2008; Balbiani, Cheikh, and Feuillade 2008), AI reasoning about action (Sardina, Patrizi, and De Giacomo 2008; Stroeder and Pagnucco 2009; De Giacomo, Patrizi, and Sardina 2013), verification (Lustig and Vardi 2009), and even robotics (Bordignon et al. 2007). From a general AI perspective, a behavior refers to the abstract operational model of a device or program, and is generally represented as a non-deterministic transition system. In a smart building setting, for instance, one may look for a so-called controller that is able to coordinate the execution of a set of devices installed in a house—e.g., automatic blinds and lights, audio and screen devices, video cameras, etc.—such that it appears as if a complex entertainment system was actually being run. A solution to the problem is called a composition.

The literature in the topic has shown how the composition problem can be effectively solved—that is, how to check for composition existence—by relying on various techniques, such as PDL satisfiability (De Giacomo and Sardina 2007), direct search (Stroeder and Pagnucco 2009), LTL/ATL synthesis (Lustig and Vardi 2009; De Giacomo and Felli 2010), and computation of special kind of simulation relations (Sardina, Patrizi, and De Giacomo 2008; Berardi et al. 2008). However, all such proposals have remained so far, at the “proof-of-concept” level and hence no thorough empirical evaluation has been performed to date.

In this paper, we present what is, up to our knowledge, the first in-depth experimental work on behavior composition that provides insights on the practical limits of some of the state-of-the-art synthesis techniques available to date. In particular, we shall focus on two competitive approaches to synthesis, namely, automated planning and model checking based game solvers. The contributions are twofold.

First, to be able to resort to planning technology, we prove formally Sardina, Patrizi, and De Giacomo (2008)’s observation that the composition problem—being a synthesis task—is closely related to planning under incomplete information (Daniele, Traverso, and Vardi 2000). More concretely, we show that solving a composition problem amounts to finding a strong-cyclic plan for a special fully-observable non-deterministic (FOND) planning problem. In doing so, a major challenge arises: the composition problem is intrinsically a generalized type of planning for safety-goals of the form “maintain ϕ,” (De Giacomo and Felli 2010; De Giacomo, Patrizi, and Sardina 2013), whereas most of the work on non-deterministic planning has considered reachability goals, that is, goals of the form “achieve ϕ.” Furthermore, such safety-goals will generally be evaluated over infinite runs of the underlying system, as the target module may encode control processes that may repeat infinitely (e.g., elevator controllers, smart house embedded systems, etc.). To address this mismatch, we rely on the recent work of Patrizi et al. (2011), who proposed a novel technique that allows a classical planner to dynamically define a reachability goal whose achievement guarantees the possibility of looping over a finite plan infinitely many times. By adapting such technique to our non-deterministic setting, our encoding enables the use of existing planners off-the-shelf, that is, without requiring any modification or extension.

∗∗We acknowledge the support of the Australian Research Council under a Discovery Project (grant DP10010332). 
Copyright © 2013, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.
The second contribution of this paper involves an empirical evaluation of three existing state-of-the-art systems that are able to synthesize such type of non-classical plans, namely, one automatic FOND planner and two model checking based game solver systems. In particular, we evaluate our encoding proposal using state-of-the-art FOND planner PRP (Muise, McIlraith, and Beck 2012) and compare it with two competitive game solver verification frameworks, namely, McMAS (Lomuscio, Qu, and Raimondi 2009) and NuGaT (based on NuSMV (Cimatti et al. 2000)), on various non-trivial classes of composition instances. Interestingly, the results obtained suggest that, despite the high computational complexity of the task at hand, the existing tools can already handle realistically sized composition instances.

The rest of the paper is organized as follows. In the next section, we provide a quick overview on behavior composition and planning under fully-observable non-deterministic domains. After that, we show how to encode a behavior composition problem into a FOND planning problem and prove the soundness and completeness of such encoding. We then report on our empirical evaluation of three state-of-the-art synthesis tools in five different classes of composition instances. We close the paper with a discussion of our findings and future work.

Preliminaries

The Behavior Composition Framework

Informally, the composition task involves the automatic synthesis of a controller that is able to suitably delegate each action request, compatible with a desired virtual target behavior module, to one of the available partially controllable behavior modules. All behaviors are modelled using finite, possibly non-deterministic, transition systems. We follow the framework in (De Giacomo and Sardina 2007; Stroder and Pagnucco 2009; De Giacomo, Patrizi, and Sardina 2013), except that we shall ignore, wlog, the shared environment space where behaviors are meant to execute and interact with.

A behavior stands for the operational model of a program or device. For example, in a smart house scenario, behaviors can represent video cameras, automatic blinds, vacuum cleaners, or web browsers. Similarly, in a factory setting, a behavior may stand for a gripper arm, a paint machine, a moving robot, or a laser device. In general, behaviors provide, step by step, the user a set of actions that it can perform (relative to its specification). At each step, the behavior can be instructed to execute one of the legal actions, causing the behavior to transition to a successor state, and thereby providing a new set of applicable actions.

Formally, a behavior is a tuple \( B = (B, A, b_0, \varrho) \), where:

- \( B \) is the finite set of behavior’s states;
- \( A \) is a set of actions;
- \( b_0 \in B \) is the initial state;
- \( \varrho \) is the finite set of behavior’s states;
- \( (b, a, b') \in \varrho \) or \( b \xrightarrow{a} b' \) denotes that action \( a \) executed in state \( b \) may lead the behavior to successor state \( b' \).

Note that we allow behaviors to be non-deterministic: one cannot know beforehand what actions will be available to execute after an action is performed, as the next set of applicable actions would depend on the successor state in which the behavior happens to be in. Hence, we say that non-deterministic behaviors are only partially controllable. A deterministic behavior is one where there is no state \( b \in B \) and action \( a \in A \) for which there exist two transitions \( b \xrightarrow{a} b' \) and \( b \xrightarrow{a} b'' \) in \( B \) with \( b' \neq b'' \). A deterministic behavior is fully controllable.

An system is a collection of behaviors at disposal (e.g., all devices installed in a smart house, aircraft, or factory) Technically, an available system is a tuple \( S = (B_1, \ldots, B_n) \), where \( B_i = (B_i, A_i, b_{0i}, \varrho_i) \), for \( i \in \{1, \ldots, n\} \), is a behavior, called an available behavior in the system. Finally, a target behavior \( T = (T, A_T, t_0, \varrho_T) \) is a deterministic behavior that represents the desired—though not available—functionality, that is, the module that one would like to have but is not readily accessible. For the sake of legibility and easier notation, we shall assume, wlog, that targets are non-blocking that is, they do not have any terminating state with no outgoing transition.\(^2\)

Figure 1 depicts a universal home entertainment system in a smart house scenario. Target \( T_{\text{ENT}} \) encapsulates the desired functionality, which involves first switching on the lights when entering the room, then providing various entertainment options (e.g., listening to music, watching a movie, playing a video game, etc.), and finally stopping active modules and switching off the lights. There are three available devices installed in the house that can be used to bring about such desired behavior, namely, a game device \( B_G \), an audio device \( B_A \), and a light \( B_L \). Note that action web in the device \( B_G \) is non-deterministic: if the device happens to evolve to state \( a_3 \), then the device needs to be reset to start functioning again. It can be shown that the desired target module

\(^2\)As customary, e.g., in LTL verification, this can be easily achieved by introducing “fake” loop transitions.
The behavior composition task is stated as follows: Given a system \( S \) and a target behavior \( T \), it is possible to (partially) control the available behaviors in \( S \) in a step-by-step manner—by instructing them on which action to execute next and observing, afterwards, the outcome in the behavior used—so as to “realize” the desired target behavior? In other words, by adequately controlling the system, it appears as if one was actually executing the target module.

To define a solution to the behavior composition problem, we consider the notion of a controller as a component able to activate, stop, and resume any of the available behaviors, and to instruct them to execute an (allowed) action. Note that the controller has full observability on the available behaviors. Formally, a controller for target \( T \) on system \( S \) is a partial function \( C : \mathcal{H}_S \times A \rightarrow \{1, \ldots, n\} \), which, given a history \( h \in \mathcal{H}_S \) of the available system (where \( \mathcal{H}_S \) is, basically, the set of all finite traces of the asynchronous product of the available behaviors) and a requested (target-compatible) action, returns the index of an available behavior to which the action in question is delegated for execution.

Intuitively, a controller (fully) realizes a target behavior if for every trace (i.e., run) of the target, at every step, the controller returns the index of an available behavior that can perform the requested action. Formally, one first defines when a controller \( C \) realizes a trace of the target \( T \). Though not required for this paper, the reader is referred to (De Giacomo and Sardina 2007) for details on how to formally characterize trace realization. Then, a controller \( C \) realizes the target behavior \( T \) iff it realizes all its traces. In that case, \( C \) is said to be an exact composition for target \( T \) on system \( S \).

**Fully Observable Non–Deterministic Planning**

Let \( F \) be a set of propositions and \( L_F = F \cup \{ \neg p \mid p \in F \} \) the set of literals over \( F \). The complement of a literal \( l \) is denoted by \( \overline{l} \); this notation trivially extends to sets of literals. A state \( s \) is a consistent subset of \( L_F \) such that \( |s| = |F| \).

A fully-observable non-deterministic (FOND) planning problem is a tuple \( \mathcal{P} = (F, I, A, G) \), where \( F \) is a set of propositional atoms (i.e., fluents), \( I \) is a conjunction of fluents \( p \in F \) which are true initially, and \( G \) is a (consistent) conjunction of literals \( l \in L_F \) which are required to be made true by selecting actions \( a \in A \) (Daniele, Traverso, and Vardi 2000). An action \( a \in A \) is a pair \( \langle \text{Pre}_a, \text{Eff}_a \rangle \) where \( \text{Pre}_a \) is a DNF formula defined over \( L_F \) and \( \text{Eff}_a \) is a conjunction of conditional (non-deterministic) effects (Anderson, Smith, and Weld 1998) of the form \( C \rightarrow E \), where \( C \) is a conjunction of literals from \( L_F \) and \( E = e_1 \lor \cdots \lor e_n \), where \( n \geq 1 \) and each \( e_i \) is a conjunction over \( L_F \), representing all the possible (exclusive) effects of the action when \( C \) holds true—exactly one-of the \( n \) effects must apply (Bonet and Givan 2005). When \( n = 1 \), we say that action \( a \) is deterministic; otherwise, \( a \) is non-deterministic. For legibility, we compactly denote conditional effects of the form \( C \rightarrow E \) by just \( E \).

The semantics of the a FOND problem \( \mathcal{P} \) as above is given by a non-deterministic state model \( \Pi_T = \langle S, \pi_0, S_G, A, A(s), \delta((a,s)) \rangle \), where:

- \( S \subseteq 2^F \) is the finite set of states of \( \Pi_T \);
- \( \pi_0 \in S \) is the (single) initial state corresponding exactly to \( I \), such that \( \pi_0 \models I \rightarrow \bigwedge_{p \in I} \neg p \), where \( I \cap \pi_0 \) is the set of propositions not in \( I \), that is, \( I \cap \pi_0 = \{ p \in F \mid I \not\models p \} \).
- Note the closed world semantics in defining \( \pi_0 \);
- \( S_G = \{ s \in S \mid s \models G \} \) is the set of goal states;
- \( A(s) = \{ a \mid s \models \text{Pre}_a \} \) is the applicability relation stating what actions are feasible (for execution) in state \( s \); and
- \( \delta((a,s), t) = \{ s' \mid \text{for all } C' \rightarrow \bigwedge_{\ell=1}^n e_\ell \in \text{Eff}_a \text{ s.t. } s' \models C' \} \) is the non-deterministic transition function denoting all possible states \( s' \) resulting from executing action \( a \) on state \( s \). Basically, for every conditional effect of the action when condition \( C \) holds, exactly one-of the \( n \) possible effects ensue.

Solutions to FOND planning problems are policies \( \pi : S \rightarrow A \), with the intended meaning that action \( \pi(s) \) is to be performed in state \( s \), that are guaranteed to transform the initial state \( s_0 \) into some goal state \( s_g \in S_G \). A policy is closed w.r.t. a state \( s \) if and only if the set of reachable states from \( s \) using \( \pi \), denoted \( S_r(s) \), is a subset of the domain of \( \pi \), that is, \( S_r(s) \subseteq \text{Dom}(\pi) \). A policy is deemed proper w.r.t. a state \( s \) if goal states can be reached using policy \( \pi \) from all \( \pi \)-reachable states \( s' \in S_r(s) \). A policy \( \pi \) is acyclic if for all possible executions \( \tau = s_0s_1s_2 \cdots \) of \( \pi \) from \( s_0 \), it holds that \( s_i \neq s_j \), for any \( i \neq j \).

Finally, a policy \( \pi \) is a valid solution for a FOND planning problem iff \( \pi \) is both closed and proper w.r.t. the initial state \( s_0 \). In turn, valid policies \( \pi \) can be further classified as either strong or strong–cyclic plans (Daniele, Traverso, and Vardi 2000), depending on whether \( \pi \) is acyclic or not, respectively.

**Compiling Behavior Composition into Non–Deterministic Planning**

The link between behavior composition and planning is not a novel idea. In fact, it was first suggested by Sardina, Patrizi, and De Giacomo (2008), noting that the task in both problems consists in deriving a control function. In planning, this control function maps world states onto actions, while in behavior composition, the control function maps system histories and actions onto components that actually execute actions. In this section, we formalize this connection by proposing a scheme to map behavior composition problems into FOND planning tasks.

From now on, let \( S = \{ B_1, \ldots, B_n \} \) be an available system, where \( B_i = \langle B_i, A_i, b_{0i}, g_i \rangle \), for \( i \in \{1, \ldots, n\} \), and \( T = (T, A_T, t_0, g_T) \) be a target module. We define the non-deterministic planning domain \( D_{\langle S, T \rangle} = \langle F, I, A \rangle \) as follows. The set of \( F \) is built from the following fluents:

- \( tgr(t) \), for each \( t \in T \), denoting that target \( T \) is in state \( t \); and
- \( beh(i, b) \), for each \( i \in \{1, \ldots, n\} \) and \( b \in B_i \), denoting that behavior \( B_i \) is in state \( b \);
• \(req(a)\) and \(srv(a)\), for each \(a \in A_T\), denoting that action \(a\) has been requested and served, respectively;
• \(ready\), denoting that the target \(T\) is ready to issue a new action request.

The initial state \(I\) encodes the starting configuration of all available behaviors as well as the target module:

\[
I = \text{ready} \land \text{tgt}(t_0) \land \bigwedge_{1 \leq i \leq n} \text{beh}(i, b_{i0}).
\]

The set of actions \(A\) is made up of three distinguished action schemas modeling the dynamics of \(S\) and \(T\) and designed to execute in a fixed sequence as follows. First, the target module issues a valid action request (via action \(\text{generate}(t)\)). Then, a delegation of such action to some behavior is performed (via action \(\text{delegate}(i, a)\)). Finally, the target evolves and becomes ready to issue a new request (via action \(\text{progress}(t, a)\)). We formally define these next.

Action \(\text{generate}(t)\) non-deterministically selects what action is requested next when the target is in state \(t\). To issue a request from state \(t\), the target needs to be “ready” and in state \(t\). The effect states that one-of the possible request from state \(t\) will ensue and the target is not “ready” anymore:

\[
\text{generate}(t) = \langle \text{ready} \land \text{tgt}(t), \bigoplus_{a \in R(T, t)} \text{req}(a) \land \neg \text{ready} \rangle,
\]

where \(R(B, s) = \{ a \mid s \xrightarrow{a} s' \text{ in } B \}\) stands for the actions that can be performed by behavior \(B\) in state \(s\). Note the non-determinism on which action is selected.

To delegate an action to a behavior \(B_i\), the action has to be “pending” (that is, currently requested) and the behavior is non-deterministic on which action is selected. When that happens, the action is considered just “served” (that is, fulfilled) and not requested anymore, and the corresponding behavior evolves as per its transition module. Formally, action \(\text{delegate}(i, a) = \langle \text{Pre}_{\text{del}}, \text{Eff}_{\text{del}} \rangle\), where:

\[
\text{Pre}_{\text{del}} = \bigvee_{(b, a, b') \in \mathcal{E}_i} (\text{beh}(i, b) \land \text{req}(a));
\]
\[
\text{Eff}_{\text{del}} = \text{srv}(a) \land \neg \text{req}(a) \land \bigoplus_{(b, a, b') \in \mathcal{E}_i} \text{trans}(b, a, b'),
\]

where \(\text{trans}(b, a, b') \triangleq (\text{beh}(i, b') \land \neg \text{beh}(i, b))\) when \(b \neq b'\), and \(\text{trans}(b, a, b') = \text{beh}(i, b)\), otherwise.

The final step involves updating the target module via the execution of action \(\text{progress}(t, a)\), defined for each \(t \in T\) and \(a \in R(T, t)\) as follows:

\[
\text{progress}(t, a) = \langle \text{srv}(a) \land \text{tgt}(t), \text{ready} \land \neg \text{srv}(a) \land \bigoplus_{(t, a, t') \in \mathcal{G}_T} \text{trans}(t, a, t') \rangle,
\]

where \(\text{trans}(t, a, t') \triangleq (\text{tgt}(t') \land \neg \text{tgt}(t))\) when \(t \neq t'\), and \(\text{trans}(t, a, t') = \text{tgt}(t)\), otherwise.

Looking at the preconditions of the three actions, it is easy to see that execution of the domain will involve sequences of the form \(\text{generate}(t) \cdot \text{delegate}(i, a) \cdot \text{progress}(t, a)\), expressing a complete action-request and delegation cycle.

While the above encoding of the dynamics behind a behavior composition problem is relatively straightforward and intuitive, it is still not expressive enough to support the synthesis of composition controllers (i.e., policies) for target behaviors containing loops. The fact is that when it comes to behavior composition, one does not plan for an achievement goal, as standard in FOND planning, but for a safety property; “always satisfy request.” Nonetheless, computing finite plans which satisfy certain safety constraints using classical planners has received considerable attention in the last years (Baier, Bacchus, and McIlraith 2009; Bauer and Haslum 2010). However, none of these approaches deal with the general problem of computing finite plans that are meant to be executed infinitely many times, as required in many control scenarios (e.g., elevator controllers, smart houses, etc). It is precisely those type of plans that are required in order to handle target modules containing loops that may yield infinite execution runs.

To be able to deal with safety goals over infinite runs of the target module, we resort to the recent novel technique proposed in (Patrizi et al. 2011) and adapt it for our non-deterministic safety-goal synthesis scenario. Roughly speaking, the idea is to find a classical finite plan encoding a so-called lasso. Namely, a sequence of actions \(\pi_1\), mapping the initial state of a system into some state \(s\), is followed by a second action sequence \(\pi_2\) that maps \(s\) into itself, which is assumed to be repeated infinitely often. Their technique, then, amounts to non-deterministically selecting the current state of the system as “start looping” configuration, and then trying to reach the exact same configuration a second time. This allows the authors to synthesize finite plans for temporally extended goals over infinite runs using classical planners. In our case, we are dealing with a non-deterministic domain. Nonetheless, the same idea generalizes to the set of executions of a strong cyclic plan \(\pi\), where there may be a (possibly empty) prefix of the execution of \(\pi\) achieving the state \(s\) where the loop starts, and subsequent actions in the execution lead to a terminal state that can be mapped into \(s\).

So, we further extend \(\mathcal{D}(S, T)\) above with auxiliary fluents and actions and obtain the following non-deterministic planning problem \(\mathcal{P}(S, T) = (F', F', A', G)\). First, let \(\hat{F} = \{ \text{beh}(i, b) \mid i \in \{1, \ldots, n\}, b \in B_i \} \cup \{ \text{tgt}(t) \mid t \in T \}\) be the subset of \(F\) containing all the fluents used to keep track of the state of all behaviors. The set of fluents is defined as:

\[
F' = F \cup \{ \text{rec}(p), \text{clsd}(p) \mid p \in \hat{F} \} \cup \{ \text{Linit}, \text{Lend}, \text{canEnd} \},
\]

where \(\text{rec}(p)\) and \(\text{clsd}(p)\)—\(p\) is recorded and closed—are (auxiliary) fluents that will be used by the planner to “record” the underlying composition configuration to which we aim to return after executing a lasso-type plan/policy. In turn, fluents \(\text{Linit}\) and \(\text{Lend}\) denote that the (search for the) lasso loop has already been initiated (and is yet active) and completed, respectively, whereas fluent \(\text{canEnd}\) is used to rule out empty lasso-loops with no action delegation.

To better understand the new fluents, we next explain the two new actions—loop and end—used to “guess” the lasso in the policy. The action \(\text{loop} = \langle \text{Pre}_{\text{loop}}, \text{Eff}_{\text{loop}} \rangle\) is used to commit to a particular configuration of \(\langle S, T \rangle\) as the
achievement goal to be bring about—the start of the lasso:

\[
\begin{align*}
\text{Pre}_{\text{loop}} &= \neg L_{\text{init}} \land \text{ready}; \\
\text{Eff}_{\text{loop}} &= L_{\text{init}} \land \neg \text{canEnd} \land \bigwedge_{p \in F} [p \rightarrow (\text{rec}(p) \land \neg \text{clsd}(p))].
\end{align*}
\]

That is, a loop can be “started” if it has not yet started and the overall system is in a “ready” state where a new action-request and delegation cycle is about to start. The effects of loop includes “recording” each true fluent \( p \) into \( \text{rec}(p) \) and marking them as not “closed”—they need to be eventually re-instantiated. In addition, it makes \( L_{\text{init}} \) true to signal the start of the loop synthesis and \( \text{canEnd} \) false to avoid “empty” loops (see end action next).

Second, the action \( \text{end} = \langle \text{Pre}_{\text{end}}, \text{Eff}_{\text{end}} \rangle \) is used to mark, or more precisely “guess,” the end of the lasso loop. The loop can be ended if it has already started and is legal to end it. The action causes the closure of every proposition that was previously recorded—via a loop action—and has been re-instantiated:

\[
\begin{align*}
\text{Pre}_{\text{end}} &= L_{\text{init}} \land \text{canEnd}; \\
\text{Eff}_{\text{end}} &= \neg \text{ready} \land \neg L_{\text{init}} \land L_{\text{end}} \land \bigwedge_{p \in F} [p \land \text{rec}(p) \rightarrow \text{clsd}(p)].
\end{align*}
\]

To be able to guess the end of the lasso, at least one action-request and delegation cycle must ensue after the start of the lasso loop, that is, we do not allow end to follow right after loop as this will imply an “empty” loop. To achieve this, we just add proposition \( \text{canEnd} \) to the effects of action \( \text{progress}(t, a) \), and define \( A' = A \cup \{ \text{loop}, \text{end} \} \).

The initial state of \( P(S, T) \) extends that of \( D(S, T) \) to account for the fact that no commitment has been made on any particular configuration, that is, \( I' = I \land \bigwedge_{p \in F} \text{clsd}(p) \). Observe that under \( I' \), propositions \( \text{canEnd}, L_{\text{init}}, L_{\text{end}}, \) and \( \text{rec}(p) \), for all \( p \in F_S \), do not hold initially.

Finally, the goal encodes the requirement of being in a state where the lasso loop has ended and every proposition previously recorded has been re-instantiated (i.e., they are not “open” anymore). Formally:

\[
G = L_{\text{end}} \land \bigwedge_{p \in F} \text{clsd}(p).
\]

Note that due to the encoding, the fact that all opened fluents have been closed is enough to guarantee that the state of the behavior composition domain has come back to that where the loop action was executed before. This is because the propositions encoding the state of each behavior are mutually exclusive. Also, observe that once action end is executed, no more actions are applicable. Therefore, end results in a non-goal terminal state, whenever the state where action end is done cannot be mapped into the state where loop was done.

The main result shows that there is a one-to-one relationship between strong cyclic solution plans \( \pi \) for \( P(S, T) \) and controllers \( C \) that are an composition solutions for target \( T \) on available system \( S \):

**Theorem 1.** Let \( (S, T) \) be a behavior composition problem instance and \( P(S, T) \) the resulting FOND planning problem obtained from the above encoding. Then, there exists a composition solution \( C \) for \( (S, T) \) if and only if there exists a strong cyclic plan \( \pi \) for \( P(S, T) \).

**Proof sketch.** We start by recalling that a controller \( C \) is a composition solution for \( T \) on system \( S \) if for all the traces of \( T \), \( C \) selects, at each step, a behavior \( B_i \) able to execute the corresponding requested action. We then observe that there exists an exact correspondence between steps \( s \xrightarrow{a,i} s' \) in histories \( h \in H_S \) of the enacted system \( E_S \) (De Giacomo, Patrizi, and Sardina 2013) and the start and end planning states in a sequence of planning actions generate(\( t \)) \cdot delegate(\( i, a \)) \cdot progress(\( t, a \)). Next, if a strong cyclic solution plan \( \pi \) exists for \( P(S, T) \), then the paths in the execution structure \( K \) (Cimatti et al. 2003) induced by \( \pi \) correspond to the histories \( h \) of \( H_S \), and moreover the valuation of \( \hat{F} \) in terminal \( P(S, T) \)'s of \( K \) encodes some previous (composition) state in \( h \), yielding thus the lasso loop. Finally, we demonstrate how all infinite traces of \( T \) can be (finitely) accounted by putting together these “looping” histories \( h \) encoded in \( K \).
are denoted as lower case letters.

In order to assess the performance of the state-of-the-art FOND planners on composition planning tasks, we have designed five different synthetic benchmarks, identified with the letters $A, C, A_U, R_1$ and $R_2$. Each benchmark contains behavior composition instances made up of behavior and target modules with specific structural features, as shown in Figure 2:

A & C Available behaviors of type $B_1$ and targets of type $T_1$ and $T_2$, respectively. Both solvable and unsolvable instances are generated.

$A_U$: Like benchmark $A$, but for every behavior of type $B_1$, we included several behaviors of type $B_2$, as “noise.” Problems in this benchmark are all solvable and account for scenarios where the composition solver has to deal with a mix of reliable behaviors ($B_1$) and unreliable behaviors ($B_2$). Note that type $B_2$ behaviors may become useless at any point, if they evolve to state 1.

$R_1$ & $R_2$: Available (deterministic) behaviors are generated randomly with up to five states, resulting in modules like $B_3$ with intricate structures, and target modules of type $T_1$ and $T_2$, respectively.

Benchmarks $A$, $C$, $R_1$ and $R_2$ feature between 10 and 60 available behaviors (i.e., devices), while we allowed for much larger (up to 120) set of behaviors in benchmark $A_U$.

As such, the number of behaviors is large enough to preclude any manual composition. See that the various examples in the literature, e.g., (De Giacomo and Sardina 2007; Stroeder and Pagnucco 2009; Yadav and Sardina 2012), never contain more than three available behaviors and are still not trivial to solve manually. When it comes to the structure of behaviors, target modules of type $T_1$ and $T_2$ capture, intuitively, sets and trees of (repeating) processes that the target user may request. In turn, available behaviors $B_1$ and $B_2$ stand for one action actuators, while behaviors of type $B_3$ with up to five states are able to model simple though concrete devices like switches, automatic blinds and lights, vacuum cleaners, and microwaves, among others. In addition, in benchmarks $R_1$ and $R_2$, the various $B_3$ type devices may overlap in their functionalities (i.e., actions). One could then argue that the number and structures of behaviors used in our benchmarks are enough to capture some real-world settings, such as smart house rooms equipped with a few dozen (simple) devices.

The structure of behaviors in benchmarks $A$, $C$ and $A_U$ was deliberately made simple to highlight the power of the domain-independent strategy at the heart of state-of-the-art FOND planners (Yoon, Fern, and Givan 2007). This strategy amounts to first finding a weak plan (Cimatti et al. 2003) for $P_{(S,T)}$, by using a classical planner on an automatically derived deterministic relaxation of $P_{(S,T)}$, and then using such a plan to guide the search for strong cyclic plans over the original FOND problem $P_{(S,T)}$. Concretely, actions that are not part of a weak plan are ruled out, as they will never be part of any strong cyclic plan. In benchmarks $R_1$ and $R_2$, the structure of the problems generated is more complex, leading to a greater variety of interaction between behaviors’ internal states and target requests.

We have evaluated the performance of state-of-the-art FOND planner PRP over the five benchmarks described above. We chose PRP over other FOND planners such as FIP (Fu et al. 2011), since PRP has been shown to clearly outperform FIP over the International Planning Competi
tion FOND benchmark (Muise, McIlraith, and Beck 2012). Behavior Composition problems can be also mapped into safety games (De Giacomo and Felli 2010), which are then solved with game solvers typically using model checking techniques. In order to get a better picture of the feasibility of our mapping of behavior composition problems into FOND problems, we compared PRP performance with that of two state-of-the-art verification frameworks, McMAS (Lomuscio, Qu, and Raimondi 2009) and NuGaT, a two-player safety game solver implemented on top of NuSMV (Cimatti et al. 2000). We did not modify in any way the source code of any of the software systems used, and relied on the default options suggested by the software authors, with the exception of NuGaT, where we found that allowing it to use dynamic variable ordering during BDD construction had a critical effect on its performance performance (see discussion below).

Figure 2: Structure of behaviors used in the benchmarks. Actions requested by targets and implemented by behaviors are denoted as lower case letters.

**Experimental Evaluation**

In order to assess the performance of the state-of-the-art FOND planners on composition planning tasks $P_{(S,T)}$ we have designed five different synthetic benchmarks, identified with the letters $A, C, A_U, R_1$ and $R_2$. Each benchmark contains behavior composition instances made up of behavior and target modules with specific structural features, as shown in Figure 2:

A & C Available behaviors of type $B_1$ and targets of type $T_1$ and $T_2$, respectively. Both solvable and unsolvable instances are generated.

$A_U$: Like benchmark $A$, but for every behavior of type $B_1$, we included several behaviors of type $B_2$, as “noise.” Problems in this benchmark are all solvable and account for scenarios where the composition solver has to deal with a mix of reliable behaviors ($B_1$) and unreliable behaviors ($B_2$). Note that type $B_2$ behaviors may become useless at any point, if they evolve to state 1.

$R_1$ & $R_2$: Available (deterministic) behaviors are generated randomly with up to five states, resulting in modules like $B_3$ with intricate structures, and target modules of type $T_1$ and $T_2$, respectively.

Benchmarks $A$, $C$, $R_1$ and $R_2$ feature between 10 and 60 available behaviors (i.e., devices), while we allowed for much larger (up to 120) set of behaviors in benchmark $A_U$.

1Unfortunately, as far as we know, there are no readily available benchmarks for behavior composition.

Experiments were run on a dual-processor Intel E8500, 3.2GHz CPUs, 8 GB of RAM, with processes being allowed to run for a max of 30 minutes and use up to 2 GB of physical memory.
The results of our experimental evaluation are summarized in Table 1. For each of the benchmarks, we report the number of composition instances (column #I), the number of problems for which each solver finished and did not finish its execution within the allotted time (columns #S and #T, resp.), and, lastly, the average run-time in seconds reported for each solver (column AT). For PRP, we also report the number of problems that exhausted the memory limit (column #M); neither NuGaT nor McMAS ever exhausted the available memory resources. The proportion of problems in benchmarks A and C for which there does not exist a composition solution is about 50% (65 out of 144 in benchmark A; 135 out of 252 in benchmark C). In these benchmarks, there was no significant difference between the average times for problems which had a solution and those which did not.

We note the remarkable performance of both NuGaT and PRP on benchmark A: both solve all the problems within 30 minutes and show similar run-times. In comparison, McMAS can only solve 60% of the problems in the benchmark, with run-times several orders of magnitude bigger than both NuGaT and PRP. The performance of McMAS is indeed in line with previous observations on the performance of verification tools on classical planning domains (Patrizi et al. 2011), or the performance of FOND planners with embedded classical planners such as NDP (Kuter and Nau 2008) or FIP (Fu et al. 2011) compared to FOND planners that rely on a verification framework like MBP (Cimatti et al. 2003). Indeed, the size of the OBDD formula reported by McMAS, when it solves a problem, are already in the billions of nodes for \(|S| = 30\).

The outstanding performance of NuGaT is explained by the usage of dynamic variable ordering on the BDD formulas denoting set of states of the fully combined system \(S \times T\) when performing the strong reachability analysis required to solve a safety-game. While the construction of BDD using dynamic variable ordering may be slower, their size is also reduced, and on the particular class of formulas conveyed by the problems in our benchmarks, the speedup of the fixed-point computation performed by NuGaT fully amortizes the overhead associated with the dynamic variable ordering scheme. When we ran NuGaT over this same benchmark without enabling dynamic variable ordering, NuGaT performance was similar to that of McMAS.

PRP outperforms NuGaT on benchmark \(A_U\). In Figure 3, we can see that as \(|S|\) grows, PRP advantage over NuGaT becomes apparent, finding solutions up to two order of magnitudes faster than NuGaT. In comparison, McMAS is only able to handle the smallest set of problems in the benchmark, which feature 15 behavior modules.

On the other hand, PRP performance degrades on benchmarks \(C, R_1,\) and \(R_2\). In benchmark C, both NuGaT and PRP are able to solve all the problems within 30 minutes but, as shown in Figure 4, PRP becomes orders of magnitude slower than NuGaT as the level of non-determinism in the planning task \(P_{(S,T)}\) increases. When the target module \(T\) can request several actions in each target state, as is indeed the case with target type \(T_2\) depicted in Figure 2, the number of possible executions of a policy \(\pi\) becomes bounded by \(O(b^d)\), where \(b\) is the maximum number of possible requests per target state and \(d\) the length of the maximal loop induced by \(T\). The strong cyclic plans for planning tasks in benchmark \(A\) and \(A_2\) have, in contrast, a linear number of executions. This is made evident in benchmarks \(R_1\) and \(R_2\), where for smaller problems—in terms of number of fluent and actions—than the largest on benchmark \(A_U\), PRP is either running out of time or, interestingly, out of memory well before the 30 minutes time limit. When increasing the time allowed from 30 minutes to 4 hours we observe that for benchmark \(R_1\), 2 more tasks are solved by PRP, running out of memory in 8 additional tasks. On benchmark \(R_2\), the effect of increasing the time limit similarly removes the time outs, PRP solving 5 additional tasks, and running out of memory in 3 more tasks. NuGaT performance profile is not affected by the increase in CPU time on benchmarks \(R_1\), yet it solves 5 more tasks on \(R_2\). Although NuGaT lacks the heuristic machinery that guides PRP policy search so effectively in benchmarks \(A, C\) and \(A_U\), it still shows a robust performance profile. At the same time, when the structure of the behavior composition problems is not exploited by PRP domain-independent heuristics, the blind search performed by NuGaT on a more succinct search space pays off, and NuGaT performance is clearly better than that of PRP.

<table>
<thead>
<tr>
<th>#I</th>
<th>#S</th>
<th>#T</th>
<th>AT</th>
<th>#S</th>
<th>#T</th>
<th>AT</th>
<th>#S</th>
<th>#T</th>
<th>AT</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A)</td>
<td>144</td>
<td>144</td>
<td>0</td>
<td>0.13</td>
<td>90</td>
<td>54</td>
<td>112</td>
<td>144</td>
<td>0</td>
</tr>
<tr>
<td>(C)</td>
<td>252</td>
<td>252</td>
<td>0</td>
<td>0.12</td>
<td>171</td>
<td>81</td>
<td>85</td>
<td>252</td>
<td>0</td>
</tr>
<tr>
<td>(A_U)</td>
<td>192</td>
<td>192</td>
<td>11.9</td>
<td>12</td>
<td>180</td>
<td>197.4</td>
<td>192</td>
<td>0</td>
<td>0.9</td>
</tr>
<tr>
<td>(R_1)</td>
<td>56</td>
<td>31</td>
<td>25</td>
<td>98.75</td>
<td>4</td>
<td>52</td>
<td>195.3</td>
<td>18</td>
<td>7</td>
</tr>
<tr>
<td>(R_2)</td>
<td>55</td>
<td>29</td>
<td>26</td>
<td>218</td>
<td>3</td>
<td>52</td>
<td>1104</td>
<td>10</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 1: Performance of NuGaT, McMAS and PRP on the proposed set of benchmarks (see details in the text).

Figure 3: Run-time comparison on benchmark \(A_U\). Each data point is the average of the run-times for all 12 problems with the same number of available behaviors.
Conclusions

In this paper we have provided, as far as we are aware, the first empirical analysis on solving behavior composition problems with state-of-the-art synthesis techniques and tools. In particular, we have used automated planning and model checking based techniques to synthesize controllers able to bring about a (virtual) target desired module by suitably coordinating a set of available modules. To that end, we first showed how behavior composition can be formally related to finding strong-cyclic solutions for a non-deterministic planning problem, thus finally providing the link between behavior composition and planning. Then, we performed an empirical evaluation of three synthesis tools—one FOND planner and two game solvers—to solve behavior composition instances over five different type of instances.

Our findings suggest that despite the high computational complexity of the problem at hand, current tools can handle non-trivial instances that are arguably of sizes expected in many real-world settings, such as smart houses, web-services, or manufacturing environments. For example, a couple of dozen behaviors with three to five states would be enough to account for smart spaces with a plethora of simple devices like automatic lights and blinds, video cameras, TVs, vacuum cleaners, among others.

When it comes to reducing behavior composition to a planning task, our encoding relies on two main ingredients. First, we transform a safety (i.e., maintenance) goal of the planning task, our encoding relies on two main ingredients. First and simplest, one needs to change the classical planner embedded in PRP in order to strengthen the planning heuristic. Recent work in classical planning (Keyder, Hoffman, and Haslum 2012) shows how to obtain very strong non-admissible heuristics by keeping track a suitable set of critical conjunctions while approximating the delete-relaxation (no other components of the planner, such as the search, are modified). While identifying this set of conjunctions could be difficult in general, it is fairly easy, in our context, to identify good candidates for such conjunctions as part of the process of building composition planning tasks \( T(\mathcal{S}, \mathcal{T}) \). The empirical results also make us curious about NuGaT’s fixed-point computation algorithm and compressed representation of the non-deterministic state model underlying \( (\mathcal{S}, \mathcal{T}) \), so as to wonder whether state-of-the-art planners can be pushed forward by looking at model checking based synthesis techniques. We plan to pursue these ideas in our future work.

References


Bauer, A., and Haslum, P. 2010. LTL goal specifications


