Composition of ConGolog Programs

Sebastian Sardina
School of Computer Science and IT
RMIT University
Melbourne, Australia
sebastian.sardina@rmit.edu.au

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

Abstract

We look at composition of (possibly nonterminating) high-level programs over situation calculus action theories. Specifically the problem we look at is as follows: given a library of available ConGolog programs and a target program not in the library, verify whether the target program executions be realized by composing fragments of the executions of the available programs; and, if so, synthesize a controller that does the composition automatically. This kind of composition problems have been investigated in the CS and AI literature, but always assuming finite states settings. Here, instead, we investigate the issue in the context of infinite domains that may go through an infinite number of states as a result of actions. Obviously in this context the problem is undecidable. Nonetheless, by exploiting recent results in the AI literature, we devise a sound and well characterized technique to actually solve the problem.

1 Introduction

In this paper, we study the composition of possibly nonterminating high-level programs over action theories. We assume:

- an *action theory*, expressed in the *situation calculus* [Reiter, 2001], describing how actions affect the state of affairs of the domain of interest;
- a library of *available high-level programs* over such action theory, expressed in (a significant fragment of) ConGolog [De Giacomo *et al.*, 2000], and which may stand for behavioral descriptions of actual devices (e.g., a controller for an elevator or a coffee delivery robot), the capabilities or logic of some services (e.g., a web-service), or even descriptions of typical operational procedures in the domain (e.g., a business process); and
- a *target program* over the same action theory that is not in the library, expressed again in ConGolog, which may stand for a behavior of interest that does not directly correspond to any of the available modules.

The problem we investigate is as follows: verify whether, for a given initial configuration of the world, expressed as a possibly infinite database (i.e., a categorical theory), the target program executions can be "*realized*" by composing fragments of the executions of the available programs so as to *mimic* the (virtual) transitions (i.e., elementary steps) of the target program at each point in time. If so, synthesize a *delegator* that does the composition automatically.

This kind of composition problem has been investigated first in the CS literature, e.g., [Berardi *et al.*, 2003; Traverso and Pistore, 2004; Lustig and Vardi, 2009],[1] and then also in AI, e.g., [De Giacomo and Sardina, 2007; Sardina *et al.*, 2008], but always assuming finite state settings.[2] Here, instead, we investigate the problem in a setting that allows us to consider potentially infinite domains that may go through an infinite number of states as actions are performed.

Specifically, we formally define what it means for a set of ConGolog programs to *mimic* the transitions of a target program, by using a greatest fixpoint second-order formula based on a suitable adaptation for our context of the formal notion of Simulation [Milner, 1971; Sardina *et al.*, 2008].

Obviously, in checking such fixpoint formula over ConGolog programs is undecidable in general. Nonetheless, by exploiting recent ideas in the AI literature [Pirri and Reiter, 1999; Kelly and Pearce, 2007; Claßen and Lakemeyer, 2008], we are able to devise a sound and well characterized procedure to solve the problem. The technique is based on three basic ingredients: *(i)* computation of the simulation through fixpoint approximates [Tarski, 1955], hoping to be able to compute the fixpoint in a *finite* number of iterations; *(ii)* use of the characteristic graphs introduced by Claßen and Lakemeyer [2008], to finitely cope with the potential infinite branching of ConGolog programs (due to ConGolog's $\pi x.\delta$ construct); and *(iii)* use of regression [Reiter, 2001] to get formulas that talk only about the initial situation, thus allowing us to drop the action theory and the second-order foundational axioms for situations altogether.

[1] Notice that ConGolog has already been considered in the context of web-service composition in [McIlraith and Son, 2002], by exploiting procedural abstraction. But the form of composition studied there is profoundly different from the one considered here.

[2] A notable exception is [Berardi *et al.*, 2005], where programs were executed over a database which may go through an infinite set of configuration. However, the techniques proposed there were again based on being able, under suitable assumptions, to reduce the setting to a finite state one.

2 Preliminaries

The *situation calculus* is a logical language specifically designed for representing and reasoning about dynamically changing worlds [Reiter, 2001]. All changes to the world are the result of *actions*, which are terms in the language. We denote action variables by lower case letters a, action names by capital letters A, and non-variable action terms by α, possibly with subscripts. A possible world history is represented by a term called a *situation*. The constant S_0 is used to denote the initial situation where no actions have yet been performed. Sequences of actions are built using the function symbol do, such that $do(a, s)$ denotes the successor situation resulting from performing action a in situation s. Relations whose truth values vary from situation to situation are called *fluents*, and are denoted by predicate symbols taking a situation term as their last argument (e.g., $Holding(x, s)$).

Within the language, one can formulate action theories that describe how the world changes as the result of the available actions. Here, we concentrate on *basic action theories* [Pirri and Reiter, 1999; Reiter, 2001]. A basic action theory $\mathcal{D}$ is the union of the following disjoint sets: the foundational, domain independent, axioms of the situation calculus (Σ); precondition axioms stating when actions can be legally performed ($\mathcal{D}_{poss}$); successor state axioms describing how fluents change between situations ($\mathcal{D}_{ssa}$); unique name axioms for actions ($\mathcal{D}_{una}$); and axioms describing the initial configuration of the world ($\mathcal{D}_{S_0}$). A special predicate *Poss*(a, s) is used to state that action a is executable in situation s; precondition axioms in $\mathcal{D}_{poss}$ characterize such predicate. In turn, successor state axioms encode the causal laws of the world being modelled; they take the place of the so-called effect axioms, but can also provide a solution to the frame problem. For example, in a music matchbox setting, a song is pending to be played if the it has just been requested, via the matchbox interface, and it is available in some CD, or the song was already pending for playback and the matchbox has *not* just started playing it:[3]

$$\begin{aligned} &Pending(song, do(a, s)) \equiv \\ &\quad (a = requestSong(song) \wedge (\exists cd) InDisk(song, cd)) \vee \\ &\qquad Pending(song, s) \wedge a \neq playBack(song). \end{aligned}$$

Once the dynamical system is modeled as a basic action theory, one can pose queries about its behavior or evolution as logical entailment queries relative to the theory.

High-Level Programs. To represent and reason about complex actions or processes obtained by suitably executing atomic actions, various so-called *high-level programming languages* have been defined, such as Golog [Levesque *et al.*, 1997], which includes usual structured constructs and constructs for nondeterministic choices, ConGolog [De Giacomo *et al.*, 2000], which extends Golog to accommodate concurrency, and IndiGolog [Sardina *et al.*, 2004], which provides means for interleaving planning and execution.

Here we concentrate on a fragment of ConGolog, which includes most constructs of the language, with the notable exception of (recursive) procedures:

α	atomic action
$\phi?$	test for a condition
$\delta_1; \delta_2$	sequence
$\delta_1 \vert \delta_2$	nondeterministic branch
$\pi x.\delta$	nondeterministic choice of argument
δ^*	nondeterministic iteration
if ϕ **then** δ_1 **else** δ_2 **endIf**	conditional
while ϕ **do** δ **endWhile**	while loop
$\delta_1 \Vert \delta_2$	concurrency

Above, α is an action term, possibly with parameters, and ϕ is situation-suppressed formula, that is, a formula in the language with all situation arguments in fluents suppressed. We denote by $\phi[s]$ the situation calculus formula obtained from ϕ by restoring the situation argument s into all fluents in ϕ.

Note the presence of nondeterministic constructs, which allow the loose specification of programs by leaving "*gaps*" that ought to be resolved by the executor. Program $\delta_1|\delta_2$ allows for the nondeterministic choice between programs δ_1 and δ_2; while $\pi x.\delta$ executes program δ for *some* nondeterministic choice of a legal binding for variable x (observe that such a choice is, in general, unbounded). δ^* performs δ zero or more times. Program $\delta_1 \| \delta_2$ expresses the concurrent execution (interpreted as interleaving) of programs δ_1 and δ_2.

As an example, consider the nondeterministic controller $\delta_{matchbox}$ for a music matchbox that serves users' requests:

```
while True do
    if $(\neg Playing \wedge (\exists song) Pending(song))$ then
      $(\pi song, disk)$
        $(Pending(song) \wedge InDisk(song, disk))?;$
        $selectSong(song);$
        $loadDisk(disk);$
        $playBack(song)$
    else $wait$
endWhile
```

That is, when the matchbox is idle and there is a song pending to be played, the controller selects a song that has been requested and the CD in which the song is, loads such CD, and starts the playback. When the matchbox is playing a song or there are no pending songs to be played, the device just waits. A full music system is modeled by taking program $\delta_{music} = (\delta_{matchbox} \| \delta_{EXO})$, where $\delta_{EXO} = (\pi a.Exog(a)?; a)^*$ represents the (external) environment, capable of executing any exogenous action (e.g., $requestSong(song)$) at anytime.

Formally, the semantics of ConGolog is specified in terms of single-steps, using the following two predicates [De Giacomo *et al.*, 2000]: *(i) Final*(δ, s), which holds if program δ may legally terminate in situation s; and *(ii) Trans*(δ, s, δ', s'), which holds if one step of program δ in situation s may lead to situation s' with δ' remaining to be executed. The definitions of *Trans* and *Final* for the constructs used in this paper are shown below:

$$\begin{aligned} &\mathit{Final}(\alpha, s) \equiv \texttt{False} \\ &\mathit{Final}(\phi?, s) \equiv \phi[s] \\ &\mathit{Final}(\delta_1; \delta_2, s) \equiv \mathit{Final}(\delta_1, s) \wedge \mathit{Final}(\delta_2, s) \\ &\mathit{Final}(\delta_1 | \delta_2, s) \equiv \mathit{Final}(\delta_1, s) \vee \mathit{Final}(\delta_2, s) \\ &\mathit{Final}(\pi x.\delta, s) \equiv \exists x.Final(\delta, s) \\ &\mathit{Final}(\delta^*, s) \equiv \texttt{True} \\ &\mathit{Final}(\delta_1 \| \delta_2, s) \equiv \mathit{Final}(\delta_1, s) \wedge \mathit{Final}(\delta_2, s) \end{aligned}$$

[3]Here, we implicitly quantify all free variables universally.

$$\begin{array}{l}
\mathit{Trans}(\alpha, s, \delta', s') \equiv s' = do(\alpha, s) \wedge \mathit{Poss}(\alpha, s) \wedge \delta' = \texttt{True}?\\
\mathit{Trans}(\phi?, s, \delta', s') \equiv \texttt{False}\\
\mathit{Trans}(\delta_1; \delta_2, s, \delta', s') \equiv\\
\qquad \mathit{Trans}(\delta_1, s, \delta_1', s') \wedge \delta' = \delta_1'; \delta_2 \vee\\
\qquad \mathit{Final}(\delta_1, s) \wedge \mathit{Trans}(\delta_2, s, \delta', s')\\
\mathit{Trans}(\delta_1 | \delta_2, s, \delta', s') \equiv\\
\qquad Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s')\\
\mathit{Trans}(\pi x.\delta, s, \delta', s') \equiv \exists x. Trans(\delta, s, \delta', s')\\
\mathit{Trans}(\delta^*, s, \delta', s') \equiv \mathit{Trans}(\delta, s, \delta'', s') \wedge \delta' = \delta''; \delta^*\\
\mathit{Trans}(\delta_1 \| \delta_2, s, \delta', s') \equiv\\
\qquad \mathit{Trans}(\delta_1, s, \delta_1', s') \wedge \delta' = \delta_1' \| \delta_2 \vee\\
\qquad \mathit{Trans}(\delta_2, s, \delta_2', s') \wedge \delta' = \delta_1 \| \delta_2'
\end{array}$$

Following [Claßen and Lakemeyer, 2008], and differently form [De Giacomo *et al.*, 2000], the test construct $\phi?$ here does not yield any transition, but it is final when satisfied. In other words, it is a *synchronous* version of the original test construct (it does not allow interleaving). With this choice, the ConGolog constructs for conditional and while-loop, which are based on synchronous tests, are immediately definable in terms of the other constructs: **if** ϕ **then** δ_1 **else** δ_2 **endIf** $= \phi?; \delta_1 | \neg\phi?; \delta_2$ and **while** ϕ **do** δ **endWhile** $= (\phi?; \delta)^*; \neg\phi?$.

Also, in this paper, we shall require that in programs of the form $\pi x.\delta$, the variable x occurs in some non-variable action term in δ, disallowing the occurrence of x *only* in tests and as an action itself. In this way, $\pi x.\delta$ acts as a construct for the nondeterministic choices of action parameters. Finally, we assume wlog that each occurrence of the construct $\pi x.\delta$ in a program uses a unique fresh variable x—no two occurrences of such a construct use the same variable.

Observe that, since we are not considering recursive procedures here, we do not need to resort to a second-order definition of *Trans* and *Final*, though we still need to consider programs as terms, cf. [De Giacomo *et al.*, 2000].

From now on, we will denote by *Axioms* the situation calculus action theory $\mathcal{D}$ that formalizes the domain of interest plus the axioms for *Trans* and *Final*.

3 ConGolog Composition

The problem we are interested in is the following. Given a basic action theory, available programs $\delta_1^0, \ldots, \delta_n^0$, and a target ConGolog program δ_t^0, we want to "execute" δ_t^0 by concurrently executing $\delta_1^0, \ldots, \delta_n^0$, while controlling their interleaving in a suitable way. In other words, we provide to the client the ability of writing *virtual* target programs over a specific domain of interest, but instead of executing such target programs directly, we actually execute programs from a library of available programs in a way that mimics the target.

As an example, consider the target program $\delta_t = [(a; b | d; c); h]^*$ in an environment where, for simplicity, all actions are always possible. Such program is virtual, and assume all we can do is to execute concurrently two available programs at hand, namely, $\delta_1 = (a; h \mid c \mid a; b)^*$ and $\delta_2 = (b \mid d; h)^*$. It is not difficult to see that, by intelligently scheduling δ_1 and δ_2, one can realize any execution of δ_t. For instance, if δ_t starts by requesting the execution of action a, then one should execute δ_1 one step by selecting its first nondeterministic program $(a; h)$; after that δ_t may only request action b followed by h, which can be realized by advancing programs δ_2 first and δ_1 then; finally, δ_t may next either stop, in which case both δ_1 and δ_2 can be legally stopped as well, or start again, in which case the two available programs can be restarted. An analogous argument applies when δ_t happens to request action d initially. What is important to note here is that we do not assume to have control on the way that δ_t may (virtually) execute, while we do have control on the way the available programs are executed. In other words, we have no control on the interpreter executing δ_t and δ_t's nondeterminism is therefore "devilish," while we have total control on the interpreter executing the concurrent program $\delta_1 \| \ldots \| \delta_n$, whose nondeterminism is "angelic."

To side-step the issue of offline vs. online execution of programs (cf. Conclusion), we assume here to have *complete information* on the initial situation, that is, $\mathcal{D}_{s_0}$ is indeed a possibly infinite *database*. Observe that while this is obviously a simplification it does not help wrt the main difficulty of this setting: programs can be legally nonterminating (they describe processes), and the number of configurations (pairs program-situation) a program goes through is potentially infinite. Indeed, the number of states the domain of interest goes through as actions are performed are infinite and unrestricted, if not by the action theory. Also, the states of the programs can be infinite due to the presence of the $\pi x.\delta$ construct, which introduces unbounded branching—there are potentially infinitely many possible remaining programs after program $\pi x.a(x); \delta(x)$ executes its first action $a(x)$, namely, $\delta(t)$ for each possible term t in the domain. This implies that the techniques coming from model checking-based verification and synthesis that have emerged as effective lately, and been used for instance in [Sardina *et al.*, 2008], cannot be applied.

To formally define what it means for a program to "*mimic*" another one, we rely on the formal notion of *simulation* [Milner, 1971; Sardina *et al.*, 2008]. In our setting, such a notion can be captured by a second-order formula that makes use of *Trans* and *Final*. Specifically, we define the predicate $\mathit{Sim}(\delta_t, \delta_1, \ldots, \delta_n, s)$ as the largest predicate S satisfying the condition $\Theta[S](\delta_t, \delta_1, \ldots, \delta_n, s)$:

$$\begin{array}{l}
\mathit{Sim}(\delta_t, \delta_1, \ldots, \delta_n, s) \equiv\\
\quad \exists S. \big(S(\delta_t, \delta_1, \ldots, \delta_n, s) \wedge\\
\qquad \forall \delta_t, \delta_1, \ldots, \delta_n, s. \Theta[S](\delta_t, \delta_1, \ldots, \delta_n, s)\big),
\end{array}$$

where $\Theta[S](\delta_t, \delta_1, \ldots, \delta_n, s)$ stands for the following formula:

$$\begin{array}{l}
S(\delta_t, \delta_1, \ldots, \delta_n, s) \rightarrow\\
\quad (\mathit{Final}(\delta_t, s) \rightarrow \bigwedge_{i=1}^n \mathit{Final}(\delta_i, s)) \ \wedge\\
\quad (\forall \delta_t', s' \mathit{Trans}(\delta_t, s, \delta_t', s') \rightarrow\\
\qquad \bigvee_{i=1}^n [\exists \delta_i'. \mathit{Trans}(\delta_i, s, \delta_i', s') \wedge\\
\qquad\qquad S(\delta_t', \delta_1, \ldots, \delta_{i-1}, \delta_i', \delta_{i+1}, \ldots, \delta_n, s')]).
\end{array}$$

By Knaster&Tarski Theorem [Tarski, 1955], the predicate *Sim* has the following notable properties:

Proposition 1. *Sim satisfies the condition* Θ*, that is,*

$$\forall \delta_t, \delta_1, \ldots, \delta_n, s. \Theta[\mathit{Sim}](\delta_t, \delta_1, \ldots, \delta_n, s)$$

is valid. Moreover, every predicate S satisfying the condition Θ is "smaller" than Sim, that is, the following is valid:

$$S(\delta_t, \delta_1, \ldots, \delta_n, s) \rightarrow \mathit{Sim}(\delta_t, \delta_1, \ldots, \delta_n, s).$$

Intuitively, the formula $\mathit{Sim}(\delta_t, \delta_1, \ldots, \delta_n, s)$ says that *(i)* if the target program δ_t may legally terminate in s, so can all the available programs $\delta_1, \ldots, \delta_n$; and that *(ii)* whatever transition the target program δ_t may make in the current situation s, such a transition can be "mimicked" by one of the available programs δ_i while the other programs remain still, and at the next step the same is true again, and again forever.

Once we have defined *Sim*, it is easy to write a formula that actually returns the "mimicking" transition:

$$\begin{aligned}&\mathit{SimTrans}_i(\delta_t, \delta_1, \ldots, \delta_n, s, \delta'_t, s', \delta'_i) \equiv\\ &\quad \mathit{Trans}(\delta_t, s, \delta'_t, s') \wedge \mathit{Trans}(\delta_i, s, \delta'_i, s') \wedge\\ &\qquad \mathit{Sim}(\delta_t, \delta_1, \ldots, \delta_{i-1}, \delta'_i, \delta_{i+1}, \ldots, \delta_n, s').\end{aligned}$$

$\mathit{SimTrans}_i(\delta_t, \delta_1, \ldots, \delta_n, s, \delta'_t, s', \delta'_i)$ says that a target transition from (δ_t, s) to (δ'_t, s') can be *mimicked* by legally advancing the i-th available program to δ'_i. By the definitions of *Sim* and *SimTrans* we get:

Proposition 2. *The following formula is valid:*

$$\begin{aligned}&\mathit{Sim}(\delta_t, \delta_1, \ldots, \delta_n, s) \rightarrow\\ &\quad (\forall \delta'_t, s'.\mathit{Trans}(\delta_t, s, \delta'_t, s') \rightarrow\\ &\qquad \textstyle\bigvee_{i=1}^{n} \exists \delta'_i.\mathit{SimTrans}_i(\delta_t, \delta_1, \ldots, \delta_n, s, \delta'_t, s', \delta'_i)).\end{aligned}$$

Hence, if $\mathit{Sim}(\delta_t, \delta_1, \ldots, \delta_n, s)$ is currently true, then we can use the formulas $\mathit{SimTrans}_i$ to choose how to mimic the target transitions *now*, knowing that, in the future, we will be able to continue mimicking the target. This is the base of an interpreter that executes the target program by actually *delegating* the target transitions to the available programs, thus realizing the composition. The interpreter, or *delegator*, is shown in Procedure 1.

Procedure 1 DELEGATOR$(\delta_t, \delta_1, \ldots, \delta_n)$

```
1:  if $\neg\mathit{Sim}(\delta_t, \delta_1, \ldots, \delta_n, S_0)$ then
2:      fail
3:  end if
4:  loop
5:      if $\mathit{Final}(\delta_t, s)$ then
6:          Ask whether to stop
7:          if stop then
8:              exit
9:          end if
10:     end if
11:     Ask $\delta'_t, s'$ s.t. $\mathit{Trans}(\delta_t, s, \delta'_t, s')$
12:     Choose $i, \delta_i$ s.t. $\mathit{SimTrans}_i(\delta_t, \delta_1, \ldots, \delta_n, s, \delta'_t, s', \delta'_i)$
13:     Execute the transition from $(\delta_i, s)$ to $(\delta'_i, s')$
14:     $\delta_t := \delta'_t$; $s = s'$; $\delta_i := \delta'_i$
15: end loop
```

Observe that the choices at line 12 are guaranteed to be possible being $\mathit{Sim}(\delta_t, \delta_1, \ldots, \delta_n, S_0)$ true. Also, observe that, with the assumption of complete information on the initial situation, all checks on formulas consist in formula evaluation, though over a possibly infinite model (cf. Conclusion).

4 The Technique

It remains to find means to check for $\mathit{Sim}(\delta_t, \delta_1, \ldots, \delta_n, s)$. Notice that this is a hard problem in general even if we have complete information on the initial database. The difficulty is that the interpretation structure to be considered is in general infinite, while direct algorithms to check for simulation work only for finite cases.

In order to tackle the infinite case, we follow an approach inspired by [Pirri and Reiter, 1999; Kelly and Pearce, 2007; Claßen and Lakemeyer, 2008]: *reduce the verification of a second-order formula wrt the whole basic action theory, to the verification of a first-order formula that is uniform in the situation argument, which, in turn, can be regressed to a formula that only talks about the initial situation.*

In particular, we devise a procedure that extracts a first-order formula that is true on the initial database iff the target program can be simulated by the available programs. Since such a procedure reduces checking a second-order formula to checking a first-order one, it may not terminate in general.

The procedure is based on three ingredients, namely, fixpoint approximates, regression, and programs' characteristic graphs. We detail each of these ingredients below.

***Sim* approximates.** Approximates $\mathit{Sim}_k(\delta_t, \delta_1, \ldots, \delta_n, s)$, for $k \geq 0$ say that program δ_t may be simulated by programs $\delta_1, \ldots, \delta_n$ in situation s for k steps. Such approximates can be formally defined by induction as follows:

$$\begin{aligned}&\mathit{Sim}_0(\delta_t, \delta_1, \ldots, \delta_n, s) \equiv\\ &\quad (\mathit{Final}(\delta_t, s) \rightarrow \textstyle\bigwedge_{i=1}^{n} \mathit{Final}(\delta_i, s)).\\ &\mathit{Sim}_{k+1}(\delta_t, \delta_1, \ldots, \delta_n, s) \equiv\\ &\quad \mathit{Sim}_k(\delta_t, \delta_1, \ldots, \delta_n, s) \ \wedge\\ &\quad (\forall \delta'_t, s'.\mathit{Trans}(\delta_t, s, \delta'_t, s') \rightarrow\\ &\qquad \textstyle\bigvee_{i=1}^{n}[\exists \delta'_i.\mathit{Trans}(\delta_i, s, \delta'_i, s') \wedge\\ &\qquad\qquad \mathit{Sim}_k(\delta'_t, \delta_1, \ldots, \delta_{i-1}, \delta'_i, \delta_{i+1}, \ldots, \delta_n, s')]).\end{aligned}$$

Informally, Sim_0 merely requires that whenever program δ_t is final, so are all programs $\delta_1, \ldots, \delta_n$; while Sim_{k+1} requires that δ_t is in k-simulation with $\delta_1, \ldots, \delta_n$ and that every transition of δ_t can be *mimicked* by some available program δ_i and be in k-simulation in the resulting next step.

By Knaster&Tarski classical result on fixpoints approximates [Tarski, 1955], we get:

Proposition 3. *If there exists a $k \geq 0$ such that*

$$\mathit{Sim}_k(\delta_t, \delta_1, \ldots, \delta_n, s) \equiv \mathit{Sim}_{k+1}(\delta_t, \delta_1, \ldots, \delta_n, s),$$

then

$$\mathit{Sim}_k(\delta_t, \delta_1, \ldots, \delta_n, s) \equiv \mathit{Sim}(\delta_t, \delta_1, \ldots, \delta_n, s).$$

We observe that while there is no guarantee that a finite k exists for the antecedent of this proposition to hold, if it does exist, then one can use approximates to compute *Sim*.

Regression. One of the most important features of basic action theories is the existence of a sound and complete *regression mechanism* for answering queries about situations resulting from performing a sequence of actions [Pirri and Reiter, 1999; Reiter, 2001]. In a nutshell, the so-called regression operator $\mathcal{R}^*$ reduces a formula ϕ about some future situation to an equivalent formula $\mathcal{R}^*[\phi]$ about the initial situation S_0, by basically substituting fluent relations with the right-hand side formula of their successor state axioms.

In this paper, we shall use a simple *one-step only* variant $\mathcal{R}$ of the standard regression operator $\mathcal{R}^*$ for basic action theories. Let ϕ be a situation-suppressed formula and α be a non-variable action term. Then $\mathcal{R}[\phi[do(\alpha, s)]]$ stands for the *one-step* regression of ϕ through action α, which is in itself a formula uniform in s. It is straightforward to adapt Pirri and Reiter [1999]'s regression theorem to get the following result:

Theorem 4. *Let $\mathcal{D}$ be a basic action theory, ϕ a situation-suppressed formula, and α a non-variable action term. Then,*

$$Axioms \models \phi[do(\alpha, s)] \equiv \mathcal{R}[\phi[do(\alpha, s)]].$$

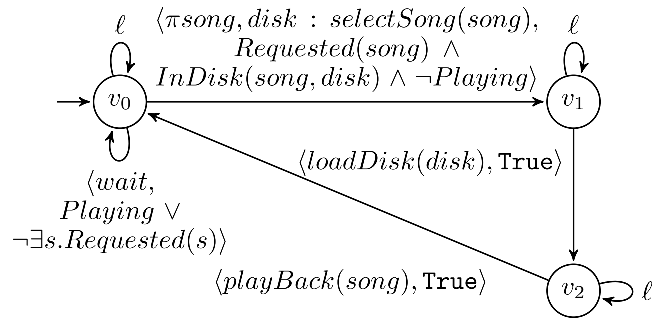
Furthermore, if $\alpha_1, \ldots, \alpha_n$ are ground non-variable action terms, then

$$Axioms \models \phi[do(\alpha_n, \ldots, do(\alpha_1, S_0)\ldots)] \textit{ iff } \mathcal{D}_{S_0} \cup \mathcal{D}_{una} \models \mathcal{R}^*[\phi[do(\alpha_n, \ldots, do(\alpha_1, S_0)\ldots)]].$$

That is, operator $\mathcal{R}$ reduces a formula about situation $do(\alpha, s)$ to a formula about situation s. By several applications of $\mathcal{R}$, hence, one can reduce formulas about future situations of S_0 to formulas about the initial situation only, which can then be verified/answered by first-order entailment/evaluation using only the initial database $\mathcal{D}_{S_0}$ and $\mathcal{D}_{una}$, a much simpler task.

Characteristic graphs. In order to compactly represent all possible configurations a program δ^0 may be in during its execution, we shall use the so-called *characteristic graph* $\mathcal{G}_{\delta^0}$ of δ^0 introduced by Claßen and Lakemeyer [2008]. The nodes V in a characteristic graph $\mathcal{G}_{\delta^0}$ are tuples of the form $\langle \delta, \chi \rangle$, stating that δ is a possible remaining program during δ^0's execution and χ characterizes the conditions under which δ may terminate (i.e., it is final). The initial node is $v_0 = \langle \delta^0, \chi^0 \rangle$. Edges in $\mathcal{G}_{\delta^0}$ stand for single transitions between program configurations and are labeled with tuples of the form $\langle \pi\vec{x} : \alpha, \psi_t \rangle$, where α is a non-variable action term and variables $\vec{x}$ may appear free in α and ψ_t. Roughly speaking, an edge states that a program may evolve into another remaining program when one chooses instantiations for $\vec{x}$ and performs action α in a situation where ψ_t holds.

Figure 1 depicts the characteristic graph for the music system example of Section 2. The edge from v_0 to v_1, for instance, represents those transitions in which a requested song is selected for playback. For that to happen, the program requires to pick a song and a compact disk such that the song is currently being requested, it is in the chosen disk, and the matchbox is currently not playing. Under such circumstances, the program may select the song in question, provided the precondition of such action holds, and evolve to node v_1. The program may next evolve to node v_2 by loading the chosen disk, provided the precondition of the loading action is true. When program is node v_0, it executes action $wait$ whenever there is no song requested for playback or the matchbox is currently playing a song. Finally, we observe that in each node, the system program may execute an exogenous actions (e.g., $requestSong(song)$) and stay in the same node, that is, in the same program configuration.


ℓ
$\langle \pi song, disk : selectSong(song), Requested(song) \wedge InDisk(song, disk) \wedge \neg Playing \rangle$
ℓ
v_0
v_1
$\langle loadDisk(disk), \texttt{True} \rangle$
$\langle wait, Playing \vee \neg\exists s.Requested(s) \rangle$
$\langle playBack(song), \texttt{True} \rangle$
v_2
ℓ

$v_0 = \langle \delta_{music} \| \; \delta_{EXO}, \; \texttt{False} \rangle$

$v_1 = \langle (loadDisk(disk); playBack(song)) \| \; \delta_{EXO}, \; \texttt{False} \rangle$

$v_2 = \langle playBack(song) \| \; \delta_{EXO}, \; \texttt{False} \rangle$

$\ell = \langle \pi a : a, \; Exo(a) \rangle$

Figure 1: Characteristic graph for program δ_{music}.

Next propositions can be shown by induction on the program structure under the assumption that the variable x from each construct $\pi x.\delta$ used in δ occurs in an action term of δ.

Proposition 5. *Given a node (δ, χ), we have that*

$$Axioms \models \chi[s] \equiv Final(\delta, s).$$

Proposition 6. *If $(\delta, \chi) \overset{\langle \pi\vec{x}:\alpha,\psi \rangle}{\longrightarrow} (\delta', \chi')$ is an edge, then*

$$Axioms \models \psi[s] \wedge Poss(\alpha, s) \equiv Trans(\delta, s, \delta', do(\alpha, s)).$$

5 The Procedure

Given the target program δ_t^0 and the available programs $\delta_1^0, \ldots, \delta_n^0$ (wlog we assume that such programs do not share variables) we compute a relation whose tuples have the form $\langle v_t, v_1, \ldots, v_n, \varphi \rangle$ where $v_t, v_1, \ldots, v_n$ are nodes in the characteristic graphs $\mathcal{G}_{\delta_t^0}$,$\mathcal{G}_{\delta_1^0}$,…,$\mathcal{G}_{\delta_n^0}$ respectively, and φ is a first-order formula. Such tuples intuitively mean that the target program in v_t is simulated by the available programs in $\langle v_1, \ldots, v_n \rangle$ "iff φ holds".

Procedure 2 $\textsc{SymSim}(\delta_t^0, \delta_1^0, \ldots, \delta_n^0)$

```
Compute characteristic graphs $\mathcal{G}_{\delta_t^0}$,$\mathcal{G}_{\delta_1^0}$,…,$\mathcal{G}_{\delta_n^0}$
$X := \{\langle (\delta_t, \chi_t), (\delta_1, \chi_1), \ldots, (\delta_n, \chi_n), \chi_t \rightarrow \bigwedge_{i=1}^n \chi_i \rangle \mid (\delta_j, \chi_j) \text{ in } \mathcal{G}_{\delta_j^0}, \; j \in \{t, 1, \ldots, n\}\}$
$X_{old} := \emptyset$
while $X \neq X_{old}$ do
    $X_{old} := X$
    $X := \textsc{Next}[X]$
end while
return $X$
```

Specifically we compute such a relation through the fixpoint procedure in Procedure 2, where the operator $\textsc{Next}[X]$ represents a "one step refinement" of the simulation: it updates the formulas φ in the tuples $\langle v_t, v_1, \ldots, v_n, \varphi \rangle$ through one step of regression while maintaining the simulation *Sim*. In other words, X represents the approximates of the simulation, which are refined at each iteration of the procedure. Formally:

$$\textsc{Next}[X] = \{\langle v_t, v_1, \ldots, v_n, \varphi_{old} \wedge \varphi_{new}\rangle \mid \\ \langle v_t, v_1, \ldots, v_n, \varphi_{old}\rangle \in X\},$$

where φ_{new} stands for the following formula:

$$\bigwedge_{v_t \overset{\pi\vec{x}\ \alpha_t/\psi_t}{\longrightarrow} v'_t \in E_t} \Big(\forall \vec{x}.\psi_t[s] \wedge \mathit{Poss}(\alpha_t, s) \rightarrow \\ \bigvee_{i=1}^{n} \bigvee_{v_i \overset{\pi\vec{y}.\alpha_i/\psi_i}{\longrightarrow} v'_i \in E_i \wedge \langle v'_t, v_1, \ldots, v'_i, \ldots, v_n, \varphi_i\rangle \in X} \\ \exists \vec{y}.\alpha_t = \alpha_i \wedge \psi_i[s] \wedge \mathcal{R}[\varphi_i(do(\alpha_i, s))]\Big).$$

As usual, we assume that $\bigwedge$ involving zero conjuncts is equal to `True` and $\bigvee$ involving zero disjuncts is equal to `False`. Note that in each iteration there will be at most one tuple $\langle v_t, v_1, \ldots, v_n, \varphi\rangle \in X$ for each $\langle v_t, v_1, \ldots, v_n\rangle$. Note also that as soon as we recognize $\varphi \equiv$ `false` in a tuple $\langle v_t, v_1, \ldots, v_n, \varphi\rangle \in X$ we can stop processing it.

We are now ready to state our core result:

Theorem 7. *Let δ_t^0, δ_1^0, …, δ_n^0 be ConGolog programs, and assume that the procedure* $\textsc{SymSim}(\delta_t^0, \delta_1^0, \ldots, \delta_n^0)$ *terminates returning the set X. Then,*

$$\mathit{Axioms} \models \mathit{Sim}(\delta_t, \delta_1, \ldots, \delta_n, s) \equiv \varphi[s],$$

where $\langle(\delta_t, \chi_t), (\delta_1, \chi_1), \ldots, (\delta_n, \chi_n), \varphi\rangle \in X$.

Proof (Sketch). We show by induction on ℓ, and exploiting Proposition 5 and 6, that in the ℓ-th *while* iteration performed by procedure SymSim, the following holds:

$$\mathit{Axioms} \models \mathit{Sim}_\ell(\delta_t, \delta_1, \ldots, \delta_n, s) \equiv \varphi_\ell[s],$$

where $\langle(\delta_t, \chi_t), (\delta_1, \chi_1), \ldots, (\delta_n, \chi_n), \varphi_\ell\rangle \in X_\ell$. We then apply Proposition 3. ☐

By considering Theorem 7 together with the soundness and completeness of regression (cf. Theorem 4) we get:

Theorem 8. *Let δ_t^0, δ_1^0, …, δ_n^0 be ConGolog programs. Assume that the procedure* $\textsc{SymSim}(\delta_t^0, \delta_1^0, \ldots, \delta_n^0)$ *terminates returning the set X. Then, for every ground situation term S:*

$$\mathit{Axioms} \models \mathit{Sim}(\delta_t, \delta_1, \ldots, \delta_n, S) \textit{ iff } \mathcal{D}_{S_0} \cup \mathcal{D}_{una} \models \mathcal{R}^*[\varphi[S]],$$

where $\langle(\delta_t, \chi_t), (\delta_1, \chi_1), \ldots, (\delta_n, \chi_n), \varphi\rangle \in X$.

Based on these results, we can define a version of the delegator in Procedure 1 that works by evaluating first-order formulas only, as shown in Procedure 3. Roughly speaking the new delegator maintains, throughout the execution, the current node (in their corresponding characteristic graph) of each available program (δ_i, χ_i) and the target (δ_t, χ_t), together with their current bindings θ_i and θ_t, respectively. At every iteration, the delegator first checks whether the target program may be entitled for termination, by checking the formula in the current target node (line 11). Notice that if this is the case, then the available programs are also entitled for termination, since *Sim* holds. If termination is not an option or is not requested, then the next step is asked to the target module. This amounts to asking for a transition in the target graph, and an action α_t and a binding θ'_t (line 17). In step 18, the delegator finds an available program δ_i such that its current node can legally transition to a new node by matching the action α_t that has been requested and guaranteeing the simulation (we know there is at least one such i, since *Sim* holds). Finally, the selected program is executed one step accordingly (line 19), the current situation is updated to include the just satisfied action α_t, and the current nodes and bindings for the target and program i are updated (line 21-22).

Procedure 3 FOLDelegator$(\delta_t, \delta_1, \ldots, \delta_n)$

```
 1: Compute $X = \textsc{SymSim}(\delta_t^0, \delta_1^0, \ldots, \delta_n^0)$
 2: Let $\langle(\delta_t^0, \chi_t^0), (\delta_1^0, \chi_1^0), \ldots, (\delta_n^0, \chi_n^0), \varphi^0\rangle \in X$
 3: **if** $\mathcal{D}_{S_0} \cup \mathcal{D}_{UNA} \not\models \varphi^0[S_0]$ **then**
 4:     *fail*
 5: **end if**
 6: **for all** $i \in \{t, 1, \ldots, n\}$ **do**
 7:     $\delta_i := \delta_i^0$; $\chi_i := \chi_i^0$; $\theta_t := \emptyset$        // initialization
 8: **end for**
 9: $S := S_0$;                   // init current situation
10: **loop**
11:     **if** $\mathcal{D}_{S_0} \cup \mathcal{D}_{UNA} \models \mathcal{R}^*[\chi_t\theta_t[S]]$ **then**
12:         **Ask** whether to *stop*        // target may stop
13:         **if** *stop* **then**
14:             *exit*
15:         **end if**
16:     **end if**
17:     **Ask** $\delta'_t, \chi'_t, \theta'_t$ s.t. $(\delta_t, \chi_t) \overset{\pi\vec{x}\ \alpha_t/\psi_t}{\longrightarrow} (\delta'_t, \chi'_t) \in E_t$ and
        $\mathcal{D}_{S_0} \cup \mathcal{D}_{UNA} \models \mathcal{R}^*[\psi_t\theta_t\theta'_t[S] \wedge Poss(\alpha_t\theta_t\theta'_t, S)]$
18:     **Choose** $i, \delta'_i, \theta'_i$ s.t. *(a)* $(\delta_i, \chi_i) \overset{\pi\vec{y}\ \alpha_i/\psi_i}{\longrightarrow} (\delta'_i, \chi'_i) \in E_i$;
        *(b)* $\langle(\delta'_t, \chi'_t), (\delta_1, \chi_1), \ldots, (\delta'_i, \chi'_i) \ldots, (\delta_n, \chi_n), \varphi\rangle \in$
        $X$; *(c)* $\alpha_t\theta_t\theta'_t = \alpha_i\theta_i\theta'_i$; *(d)* $\mathcal{D}_{S_0} \cup \mathcal{D}_{UNA} \models$
        $\mathcal{R}^*[\psi_i\theta_i\theta'_i[S]]$; and *(e)* $\mathcal{D}_{S_0} \cup \mathcal{D}_{UNA} \models$
        $\mathcal{R}^*[\varphi[do(\alpha_t\theta_t\theta'_t, S)]]$
19:     **Execute** transition from configuration $(\delta_i\theta_i, S)$ to con-
        figuration $(\delta'_i\theta_i\theta'_i, do(\alpha_t\theta_t\theta'_t, S))$
20:     $S := do(\alpha_t\theta_t\theta'_t, S)$   // new current situation
21:     $\delta_t := \delta'_t$; $\chi_t := \chi'_t$; $\theta_t := \theta_t\theta'_t$;     // update target
22:     $\delta_i := \delta'_i$; $\chi_i := \chi'_i$; $\theta_i := \theta_i\theta'_i$;  // update program $i$
23: **end loop**
```

As a direct consequence of Theorems 5, 6, 7, and 4, we obtain:

Theorem 9. *If* $\textsc{SymSim}(\delta_t^0, \delta_1^0, \ldots, \delta_n^0)$ *terminates, then* FOLDelegator$(\delta_t, \delta_1, \ldots, \delta_n)$ *is a sound and complete implementation of* Delegator$(\delta_t, \delta_1, \ldots, \delta_n)$.

In other words, for every initial database, FOLDelegator will produce exactly the same input/output behavior, at each point in time, of procedure Delegator.

6 Conclusion

In this paper, we looked at the problem of composing a desired high-level program by suitably executing concurrently a set of available programs at hand. Our technique is able to handle programs that may not terminate and that run over domains that may go through an infinite number of states. We observe here that if the initial database is finite, then the delegator in Procedure 3, which only requires to evaluate first-order formulas, can be readily implemented using standard relational database technology.

As mentioned, having complete information on the initial situation at runtime allowed us to side-step the issue of offline vs. online executions of high-level programs [Sardina *et al.*, 2004]. Indeed, under complete information of the initial situation the two kinds of execution styles coincide. To extend our approach to deal with incomplete information on the initial situation, more work has to be done. In particular, while the delegators in Procedure 1 and Procedure 3 remain formally well-defined (using entailment instead of formula evaluation), they may not be able to carry out the composition, as they may get stuck by not being able to decide the truth value of a formula. This is a subtle issue, which has been investigated in the context of *epistemic feasibility* of plans, see e.g., in [Sardina *et al.*, 2004], and which becomes crucial also in our composition context.

The main limitation of the approach proposed here is the lack of (sufficient) conditions guaranteeing termination of procedure SymSim. Indeed, finding cases for which we have guarantees of termination of a procedure that eliminates fixpoints (as in this paper, or [Claßen and Lakemeyer, 2008] and [Kelly and Pearce, 2007]) is an interesting research direction for future work.

Acknowledgments

We thank the reviewers for their interesting comments. Sebastian Sardina was supported by Agent Oriented Software and the Australian Research Council (under grant LP0882234), and the National Science and Engineering Research Council of Canada (under a PDF fellowship).

References

[Berardi *et al.*, 2003] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Massimo Mecella. Automatic composition of e-Services that export their behavior. In *Proceedings of the International Joint Conference on Service Oriented Computing (ICSOC)*, pages 43–58, 2003.

[Berardi *et al.*, 2005] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Rick Hull, and Massimo Mecella. Automatic composition of transition-based semantic web services with messaging. In *Proceedings of the International Confernece on Very Large Databases (VLDB)*, pages 613–624, 2005.

[Claßen and Lakemeyer, 2008] Jens Claßen and Gerhard Lakemeyer. A logic for non-terminating Golog programs. In *Proceedings of Principles of Knowledge Representation and Reasoning (KR)*, pages 589–599, 2008.

[De Giacomo and Sardina, 2007] Giuseppe De Giacomo and Sebastian Sardina. Automatic synthesis of new behaviors from a library of available behaviors. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1866–1871, 2007.

[De Giacomo *et al.*, 2000] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence Journal*, 121(1–2):109–169, 2000.

[Kelly and Pearce, 2007] Ryan F. Kelly and Adrian R. Pearce. Property persistence in the situation calculus. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1948–1953, 2007.

[Levesque *et al.*, 1997] Hector J. Levesque, Ray Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.

[Lustig and Vardi, 2009] Yoad Lustig and Moshe Y. Vardi. Synthesis from component libraries. In *Proceedings of the International Confernece on Foundations of Software Science and Computational Structures (FOSSACS)*, volume 5504 of *LNCS*, pages 395–409. Springer, 2009.

[McIlraith and Son, 2002] Sheila A. McIlraith and Tran Cao Son. Adapting Golog for composition of semantic web service. In *Proceedings of Principles of Knowledge Representation and Reasoning (KR)*, pages 482–493, 2002.

[Milner, 1971] Robin Milner. An algebraic definition of simulation between programs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 481–489, 1971.

[Pirri and Reiter, 1999] Fiora Pirri and Ray Reiter. Some contributions to the metatheory of the situation calculus. *Journal of the ACM*, 46(3):261–325, 1999.

[Reiter, 2001] Ray Reiter. *Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, 2001.

[Sardina *et al.*, 2004] Sebastian Sardina, Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. On the semantics of deliberation in IndiGolog – From theory to implementation. *Annals of Mathematics and Artificial Intelligence*, 41(2–4):259–299, August 2004.

[Sardina *et al.*, 2008] Sebastian Sardina, Fabio Patrizi, and Giuseppe De Giacomo. Behavior composition in the presence of failure. In *Proceedings of Principles of Knowledge Representation and Reasoning (KR)*, pages 640–650, 2008.

[Tarski, 1955] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Traverso and Pistore, 2004] Paolo Traverso and Marco Pistore. Automated composition of semantic web services into executable processes. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 380–394, 2004.