Local Conditional High-Level Robot Programs (extended version)

Sebastian Sardiña

Department of Computer Science
University of Toronto
Toronto, Canada M5S 1A4
ssardina@cs.toronto.edu,
WWW home page: http://www.cs.toronto.edu ~ssardina

Abstract. When it comes to building robot controllers, high-level programming arises as a feasible alternative to planning. The task then is to verify a high-level program by finding a legal execution of it. However, interleaving offline verification with execution in the world seems to be the most practical approach for large programs and complex scenarios involving information gathering and exogenous events.
In this paper, we present a mechanism for performing local lookahead for the Golog family of high-level robot programs. The main features of such mechanism are that it takes sensing seriously by constructing conditional plans that are ready to be executed in the world, and it mixes perfectly with an account of interleaved perception, planning, and action. Also, a simple implementation is developed.

1 Motivation

In general terms, this paper is concerned with how to conveniently specify the behavior of an intelligent agent or robot living in an incompletely known dynamic world. One popular way of specifying the behavior of an agent is through planning — the generation of a sequence of actions achieving or maintaining a set of goals. To cope with incomplete knowledge, some sort of sensing behavior is usually assumed [1, 2], resulting in conditional or contingency plans [3–5], where branches are executed based on the outcome of perceptual actions or sensors. The task of a conditional planner is to find a tree-structured plan that accounts for and handles all eventualities, in advance of execution.

However this type of conditional planning is computationally difficult and impractical in many robot domains. The non-conditional planning problem is already highly intractable, and taking sensing into account only makes it worse.

High-level logic programming languages like Golog [6] and ConGolog [7] offer an interesting alternative to planning in which the user specifies not just a goal, but also constraints on how it is to be achieved, perhaps leaving small subtasks to be handled by an automatic planner. In that way, a high-level program serves as a "guide" heavily restricting the search space. By a high-level program, we mean one whose primitive instructions are domain-dependent actions of the robot, whose tests involve domain-dependent fluents affected by these actions, and whose code may contain nondeterministic choice points.

Instead of looking for a legal sequence of actions achieving some goal, the task now is to find a sequence that constitutes a legal execution of a high-level program. Originally, Golog and ConGolog programs were intended to be solved offline, that is, a complete solution was obtained before committing even to the first action. Also, sensing behavior was not considered so that the approach to uncertainty resembles more that of conformant planners [8]. While Lakemeyer [9] suggested an extension of Golog to handle sensing and contingent plans, De Giacomo and Levesque [10] provided an account of interleaved perception, planning, and action [11, 12] for ConGolog programs.

In this paper, we propose to combine both improvements by suggesting a method of executing high-level robot programs that is both conditional (in the sense of Lakemeyer) and local (in the sense of De Giacomo and Levesque.) The advantages are twofold. First, we can expect to deal with much larger programs, assuming planning is locally restricted. Second, the offline verification of subtasks will handle sensing and provide contingent solutions. Although this may seem initially a trivial intersection of the two pieces, it is not. For one, sGolog semantics is given as a macro expansion while an incremental execution is defined with a single-step semantics. Furthermore, sGolog does not handle ConGolog constructs, namely those for concurrency and reactive behavior, which we do not want to give up.

The rest of the paper is organized as follows: in the next two sections, we give brief introductions to the situation calculus, high-level programs, and their executions. Section 4 is devoted to our approach to offline verification of programs. In Section 5, we develop a simple and provably sound Prolog implementation. We draw conclusions and discuss future lines of research in Section 6.

2 Situation Calculus and Programs

In this section, we start by explaining the situation calculus dialect on which all the high-level approach is based on, and after that, we informally show what high-level programs look like.

The situation calculus is a second order language specifically designed for representing dynamically changing worlds [13, 14]. We will not go over it here except to note the following components: there is a special constant S_0 used to denote the *initial situation* where no actions have yet occurred; there is a distinguished binary function symbol do where $do(a, s)$ denotes the successor situation to s resulting from performing action a; relations whose truth values vary from situation to situations are called *fluents*, and are denoted by predicate/function symbols taking a situation term as their last argument; there is a special predicate $Poss(a, s)$ used to state that action a is executable in situation s. Depending on the type of action theory used we may have other predicates and axioms to state what are the sensing results of special sensing actions [4] or the outcomes of onboard sensors [2] at some situation. Finally, by a history σ

we mean a sequence of pairs (a, μ) where a is a primitive action and μ encodes the sensing results at that point.[1] A formula $Sensed[\sigma]$ in the language can be defined stating the sensing results of history σ. Lastly, $end[\sigma]$ stands for the situation term corresponding to history σ. Informally, while $Sensed[\sigma]$ extracts from σ all the sensing information already gathered, $end[\sigma]$ extracts the sequence of actions already performed.

On top of the situation calculus, we can define logic-based programming languages like Golog [6] and ConGolog [7], which, in addition to the primitive actions of the situation calculus, allow the definition of complex actions. Indeed, Golog offers all the control structures known from conventional programming languages (e.g., sequence, iteration, conditional, etc.) plus some nondeterministic constructs. It is due to these last control structures that programs do not stand for complete solutions, but only for sketches of them whose gaps have to be filled later, usually at execution time. ConGolog extends Golog to accommodate concurrency and interrupts. As one may expect, both Golog and ConGolog rely on an underlying situation calculus axiomatization to describe how the world changes as the result of available actions, i.e. a theory of action. For instance, *basic action theories* [15] or the more general *guarded action theories* [2] may be used for that propose.

To informally introduce the syntax and some of the common constructs of these programming languages, we show next a possible ConGolog program for a version of the well-known airport problem [4, 9, 16]. Suppose that the ultimate goal of an agent is to board its plane. For that, she first needs to get to the airport, go to the right airline terminal, and once there, she has to get to the correct gate, and finally board her plane. In addition, she probably wants to buy something to read and drink before boarding the plane. The following may be a ConGolog control program for such agent:

proc $catch\_plane1$
    $(\pi a.a)^*$; at(airport)?;
    (goto(term1) | goto(term2));
    (buy(magazine) | buy(paper));
    if $gate \geq 90$ **then** { goto(gate); buy(coffee) } **else**
                    { buy(coffee); goto(gate) }
    board_plane;
end_proc

where $\delta_1; \delta_2$ stands for sequence of programs δ_1 and δ_2; $\pi x.\delta(x)$ for nondeterministic choice of argument x; $\delta_1 | \delta_2$ for nondeterministic selection between programs δ_1 and δ_2; and δ^* for nondeterministic iteration of program δ (zero, one, or more times). Finally, action $(\phi)?$ checks that condition ϕ holds. As it is easy to observe, the above program has many gaps due to nondeterministic points that need to be resolved by an automated planner. For example, the first two complex actions $(\pi a.a)^*; at(airport)?$ require the agent to select some number of actions (pick up

[1] The outcome of a itself in basic theories, or the values of all sensors in guarded theories.

the car key, get in the car, drive to the airport, etc.) so that after their execution she would eventually be at the airport. As the reader may have noticed, that particular sub-task is very similar to classical planning.[2] Once in the airport, the agent has to decide whether to head to terminal 1 or 2 (another gap to be filled) and, after that, whether to buy a magazine or a newspaper. Finally, she would buy something to drink and board the airplane. However, in case the gate number is 90 or up, it is preferable to buy coffee at the gate, otherwise it is better to buy coffee before going to the gate.

3 Incremental Execution of Programs

Finding a legal execution of high-level programs is at the core of the whole approach. Indeed, a sequence of action standing for a program execution will be taken as the ultimate agent behavior. Originally, Golog and ConGolog programs were conceived to be executed (verified) offline. In other words, we look for a sequence of actions $[a_1, ..., a_m]$ such that $Do(\delta, s, do([a_1, ..., a_m], S_0))$[3] is entailed by the specification, where $Do(\delta, s, s')$ is intended to say that situation s' represents a legal execution of program δ from the initial situation s. Once a sequence like that is found, the agent is supposed to execute it one action at a time. Clearly, this type of execution remains infeasible for large programs and precludes both runtime sensing information and reactive behavior. To deal with these drawbacks, De Giacomo and Levesque [10] provided a formal notion of interleaved planning, sensing, and action [11,12] which we support for cognitive robotic applications. In their account, they make use of two predicates defined in [7] in order to give a single-step semantics to ConGolog programs:

- $Trans(\delta, s, \delta', s')$ is meant to say that program δ in situation s may legally execute one step, ending in situation s' with program δ' remaining;
- $Final(\delta, s)$ is meant to say that program δ may legally terminate in situation s.

Both predicates are defined inductively for each language construct. As an example, we list the axioms corresponding to the nondeterministic choice of program and sequence:[4]

$$\begin{aligned}
Trans(\delta_1|\delta_2, s, \delta', s') &\equiv Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s') \\
Trans(\delta_1;\delta_2, s, \delta', s') &\equiv Trans(\delta_1, s, \delta', s'') \wedge \delta' = (\delta''; \delta_2) \vee \\
&\qquad Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s') \\
Final(\delta_1|\delta_2, s) &\equiv Final(\delta_1, s) \vee Final(\delta_2, s) \\
Final(\delta_1;\delta_2, s) &\equiv Final(\delta_1, s) \wedge Final(\delta_2, s)
\end{aligned}$$

[2] In fact, one would prefer to avoid this kind of sub-tasks and write more detailed programs since the search space required for such sub-tasks will be huge.

[3] $do([a_1, ..., a_m], S_0))$ denotes the situation term $do(a_m, do(a_{m-1}, ..., do(a_1, S_0))...)$.

[4] From now on, we assume all free variables are universally quantified.

From now on, we use $Axioms$ to refer to the set of axioms defining the underlying theory of action, the axioms for $Trans$ and $Final$, and those needed for the encoding of programs as first-order terms (see [7].) Also, $Trans^*$ stands for the second-order definition of the transitive closure of $Trans$.

Definition 1. *An online execution of a program δ_0 starting from a history σ_0 is a sequence $(\delta_0, \sigma_0), \ldots, (\delta_n, \sigma_n)$, such that for $i = 0, .., n-1$:*

$$Axioms \cup Sensed[\sigma_i] \models Trans(\delta_i, end[\sigma_i], \delta_{i+1}, end[\sigma_{i+1}])$$

$$\sigma_{i+1} = \begin{cases} \sigma_i, & \text{if } end[\sigma_{i+1}] = end[\sigma_i] \\ \sigma_i \cdot (a, \mu), & \text{if } end[\sigma_{i+1}] = do(a, end[\sigma_i]) \\ & \text{and } \mu \text{ is the sensing outcomes after } a \end{cases}$$

Furthermore, the online execution is successful if:

$$Axioms \cup Sensed[\sigma_n] \models Final(\delta_n, end[\sigma_n])$$

Among other things, with an online (incremental) execution, it is possible to gather information after each transition. However, given that an incremental execution requires committing in the world at each step and programs may contain nondeterministic points, some lookahead mechanism is required to avoid unsuccessful (dead-end) executions. To that end, in [10] a new language construct Σ, the search operator, is provided as a local controlled form of offline verification where the amount of lookahead to be performed is under the control of the programmer. As with all the other language constructs, a single-step semantics for it can be defined such that $\Sigma\delta$ selects from all possible transitions of (δ, s) those for which there exists a sequence of further transitions leading to a final configuration (δ', s'). Formally,

$$\begin{aligned} Final(\Sigma\delta, s) &\equiv Final(\delta, s) \\ Trans(\Sigma\delta, s, \delta', s') &\equiv \exists\gamma, \gamma', s''.\delta' = \Sigma\gamma \wedge Trans(\delta, s, \gamma, s') \wedge \\ &\qquad Trans^*(\gamma, s', \gamma', s'') \wedge Final(\gamma', s'') \end{aligned}$$

Nonetheless, we recognize some important limitations of this search operator. In particular, we are concerned with its limitation to explicitly handle sensing and the fact that it does not generate solutions that are ready to be carried out by the agent. This is because search only calculates the next "safe" action the agent should commit to, even though there may be a *complete* (conditional) course of action to follow. What we propose here is a new search operator which overcomes both issues.

3.1 Offline Verification with Sensing

As already noted, one way to cope with incomplete information, especially when sensors are cheap and accurate, or effectors are costly, is by gaining new information through sensing and adopting a contingent planning strategy. Consider a revised version of the airport example in which the agent does not know the gate number, but can learn it by examining the departure screen at the right terminal.

```
proc catch_plane2
    (πa.a)*; at(airport)?;
    (goto(term1) | goto(term2));
    watch_screen;          /* Sensing Action! */
    (buy(magazine) | buy(paper));
    if gate ≥ 90 then { goto(gate); buy(coffee) } else
                      { buy(coffee); goto(gate) }
    board_plane;
end_proc
```

Conformant planning (like [8]), the development of non-conditional plans that do not rely on sensory information, cannot generally solve our example because there is no linear course of action that solves the program under any possible outcome of the sensing action *watch_screen*. It should be clear then that neither Golog nor ConGolog would find any successful offline execution for *catch_plane*2. An online execution, however, would adapt the sequence depending on the information observed on the boarding panel.

In [9], it was argued that, yet, "*there is a place for offline interpretation of programs with sensing.*" In fact, Lakemeyer suggested an extension of Golog, namely sGolog, that handles sensing actions offline by computing conditional plans instead of linear ones. These plans are represented - in the language - by *conditional action trees* (CATs) terms of the form $a \cdot c_1$ or $[\phi, c_1, c_2]$, where a is an action term, ϕ is a formula, and c_1 and c_2 are two CATs. Roughly, an sGolog solution for our airport example would look as follows:

$$\begin{aligned} c = goto(airport) \cdot goto(term2) \cdot watch\_screen \cdot buy(paper) \\ \cdot [gate \geq 90, goto(gate) \cdot buy(coffee) \cdot board\_plane, \\ buy(coffee) \cdot goto(gate) \cdot board\_plane] \end{aligned}$$

sGolog extends Golog's $Do(\delta, s, s')$ to $Do_s(\delta, s, c)$ which expands into a formula of the situation calculus augmented by a set of axioms Ax_{CAT} for dealing with CAT terms. $Do_s(\delta, s, c)$ may be read as "executing the program δ in situation s results in CAT c." It is worth noting that although sGolog is able to build conditional plans as the above one, it requires programs to use a special action $branch\_on(\phi)$ to state where to split and how. Intuitively, a $branch\_on(\phi)$ tells the planner that it should split w.r.t. the condition $\phi(s)$. In that sense, the above CAT c is not a seen as a legal solution for program *catch_plane*2, but it is a legal one for the following version of it:

```
proc catch_plane2b
    (πa.a)*; at(airport)?;
    (goto(term1) | goto(term2));
    watch_screen;                /* Sensing Action! */
    (buy(magazine) | buy(paper)); branch_on(gate ≥ 90);
```

if *gate* ≥ 90 **then** { goto(gate); buy(coffee) } **else**
{ buy(coffee); goto(gate) }
board_plane;
end_proc

From now on, we denote by δ^- to the program δ with all its "*branch_on*" actions suppressed (e.g., $catch\_plane2b^- = catch\_plane2$).

4 Conditional Lookahead

Lakemeyer argued that many programs with a moderate number of sensing actions can very well be handled with his approach. Even though we are skeptical about doing full offline execution of any (large) program, we consider his argument a much more plausible one if offline execution were restricted to local places in a program. In what follows, we define a new search construct providing a local lookahead mechanism that takes potential sensing behavior seriously and fits smoothly with the incremental execution scheme from Section 3. We begin by defining a subset of useful high-level programs.

Definition 2. *A Golog program δ is a conditional program plan (CPP) if*

- $\delta = nil$*, i.e.,* δ *is the empty program;*
- $\delta = A$*,* A *is an action term;*
- $\delta = (A; \delta_1)$*,* A *is an action term, and* δ_1 *is a CPP;*
- $\delta =$ **if** ϕ **then** δ_1 **else** δ_2*,* ϕ *is a fluent formula, and* δ_1, δ_2 *are CPPs.*

Under our approach, CPPs will play the role of conditional-plan solutions. Notice they are no more than regular *deterministic* high-level programs where only sequence of actions and conditional splitting (branching) are allowed. It is easy to state an axiom defining the relation $condPlan(\delta)$, which, informally, holds only when δ is a CPP.

Next, we introduce a two-place function *run*–our version of Lakemeyer's *cdo* function–which takes a CPP δ and a situation s, and returns a situation which is obtained from s using the actions along a path in δ.[5] Briefly, *run* follows a certain branch in the CPP depending on the truth value of the branch-conditions.

$$\begin{aligned} run(nil, s) &= s \\ run(a, s) &= do(a, s) \\ run((a; \delta), s) &= run(\delta, do(a, s)) \\ \phi(s) \supset run(\text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2, s) &= run(\delta_1, s) \\ \neg\phi(s) \supset run(\text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2, s) &= run(\delta_2, s) \end{aligned}$$

Lastly, predicate $knowHow(\delta, s)$ is intended to mean that "we know how to execute δ starting at situation s." By this we mean that at every branching

[5] A CPP can be easily seen as a tree with actions and conditional splittings as nodes.

point in the CPP δ, the branch-formula is known to be true or false. In order to enforce this restriction, programs would generally have some sensing behavior that will guarantee that each formula in a CPP will be known. A high-level description of the corresponding axioms for run is the following:

$$\begin{aligned}
knowHow(nil, s) &\equiv TRUE \\
knowHow(a, s) &\equiv TRUE \\
knowHow((a;\delta), s) &\equiv knowHow(\delta, do(a, s)) \\
knowHow(\text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2, s) &\equiv Kwhether(\phi, s) \ \wedge \\
&\quad \phi(s) \supset knowHow(\delta_1, s) \ \wedge \\
&\quad \neg\phi(s) \supset knowHow(\delta_2, s)
\end{aligned}$$

Observe that the last axiom makes use of predicate $Kwhether(\phi, s)$ defined in [17], which gives us a solution to knowledge in the situation calculus. Relation $Kwhether(\phi, s)$ is intended to say that the condition ϕ will be eventually known (true or false) in situation s.[6] Although it is possible to use more general definitions of "knowing how to execute a program" we stick to the above one for the the sake of simplicity.

We now have all the machinery needed to define our new mechanism of controlled lookahead. Namely, we introduce a conditional search operator Σ_c that, instead of only returning the next action to be performed, it computes a whole (remaining) CPP that solves the original program and is ready to be executed online. To that end, we define $Final$ and $Trans$ for the new operator. For $Final$, we have that $(\Sigma_c\delta, s)$ is a final configuration if (δ, s) itself is.

$$Final(\Sigma_c\delta, s) \equiv Final(\delta, s)$$

For $Trans$, a configuration $(\Sigma_c\delta, s)$ can evolve to (δ', s) if δ' is a CPP that the agent knows how to execute from s, and such that *every possible and complete* path through δ' represents a successful execution of the original program δ.

$$\begin{aligned}
Trans(\Sigma_c\delta, s, \delta', s') \equiv\ & s' = s \wedge condPlan(\delta') \wedge knowHow(\delta', s) \ \wedge \\
& \exists\delta''.Trans^*(\delta, s, \delta'', run(\delta', s)) \wedge Final(\delta'', run(\delta', s))
\end{aligned}$$

While the first line defines what the "form" of a legal solution is, the second one makes the connection between the CPP δ' and the original program δ. Notice we want this sentence to be true in every interpretation, and, therefore, the sequence of actions produced by $run(\delta', s)$ must always correspond to a (complete) sequence of transitions for δ. This is very important since not every CPP will be acceptable, but only the ones that are "hidden" in δ. It is important to remark that different interpretations could lead to different "runs" and transitions.

From now on, we assume the above two axioms for Σ_c, together with the axioms for run, $condPlan$ and $knowHow$, are all included into the already mentioned set of axioms $Axioms$. If, for example, we execute $\Sigma_c catch\_plane2$ we get

[6] See [17] for a complete coverage of knowledge and sensing in the situation calculus.

that

$$Axioms \cup Sensed[\sigma_0] \models Trans(\Sigma_c catch\_plane2, S_0, \delta', S_0)$$

where

$$\begin{aligned}\delta' = {} & goto(airport); goto(term2); watch\_screen; buy(paper);\\ & \quad \text{if } gate \geq 90 \text{ then } \{goto(gate); buy(coffee); board\_plane\}\\ & \quad\quad \text{else } \{buy(coffee); goto(gate); board\_plane\}\end{aligned}$$

In this case, $run(\delta', S_0)$ would have two different interpretations w.r.t. the set $Axioms \cup Sensed[\sigma_0]$. In the models where $gate \geq 90$, function $run(\delta', S_0)$ denotes the situation

$$\begin{aligned}& do([goto(airport), goto(term2), watch\_screen,\\ & \quad buy(paper), goto(gate), buy(coffee), board\_plane], S_0)\end{aligned}$$

On the contrary, in those models where $gate < 90$, function $run(\delta', S_0)$ denotes the situation term

$$\begin{aligned}& do([goto(airport), goto(term2), watch\_screen,\\ & \quad buy(paper), buy(coffee), goto(gate), board\_plane], S_0)\end{aligned}$$

The point is that, in either case, $run(\delta', S_0)$ is supported by the original program $catch\_plane2$.

By inspecting the above $Trans$ axiom for Σ_c, one can see that Σ_c performs no action step, but calculates a remaining program δ' (in particular, a CPP one) that is ready to be executed online, and that has previously considered how future sensing will be managed. This implies that the final sequence of actions will eventually depend on the future sensing outcomes; in our example, after committing to action $watch\_screen$. Furthermore, the CPP returned has already solved all nondeterministic points in the original program as well as all concurrency involved on it. In some sense, Σ_c can be visualized as an operator that transforms an arbitrary complex ConGolog program into a simple and deterministic CPP without requiring it to know in advance how future sensing will turn out.

The following are some useful properties of Σ_c.

Property 1

$$Trans((\Sigma_c\delta_1)|(\Sigma_c\delta_2), s, \delta', s') \equiv Trans(\Sigma_c(\delta_1|\delta_2), s, \delta', s')$$

i.e., search distributes over the nondeterministic choice of program. An interesting example comes up with programs $\delta_1 = (a; \phi; b)$ and $\delta_2 = (a; \neg\phi; c)$. Even though not trivial to see, the CPP $\delta' = (a; \text{if } \phi \text{ then } b \text{ else } c)$ is a solution for both $\Sigma_c(\delta_1|\delta_2)$ and $(\Sigma_c\delta_1)|(\Sigma_c\delta_2)$. The former case is easy; the latter, though, involves realizing that, in the interpretation where ϕ holds, the program $\Sigma_c\delta_1$ is the one that performs the transition and a "run" of δ' is action a followed by action b. However, in the interpretation where $\neg\phi$ holds, the program chosen for the transition is $\Sigma_c\delta_2$, and a "run" of δ' is action a followed by action c.

Property 2

$$Trans(\Sigma_c\delta, s, \delta', s') \supset Final(\Sigma\delta, s) \vee \exists\delta''.s''.Trans(\Sigma\delta, s, \delta'', s'')$$

This means that whenever there is a transition w.r.t. Σ_c, there is also a transition w.r.t. Σ. However, the converse does not apply.

Property 3

$$\begin{aligned} &Trans(\Sigma_c(\delta_1;\delta_2), s, \delta, s) \equiv \exists\delta_1'.Trans(\Sigma_c\delta_1, s, \delta_1', s) \wedge \\ &\qquad \exists\delta^*.Trans(\Sigma_c\delta_2, run(\delta_1', s), \delta^*, run(\delta_1', s)) \wedge ext_{CPP}(\delta_1', \delta, \delta^*, s) \end{aligned}$$

i.e., a solution for $\delta_1;\delta_2$ can be seen as some solution for δ_1 extended, at each leaf, with a conditional plan that solves δ_2. Relation $ext_{CPP}(\delta', \delta, \delta^*, s)$ is the analogous one to sGolog's $ext(c', c, c^*, s)$. Informally, $ext_{CPP}(\delta', \delta, \delta^*, s)$ means that CPP δ is obtained by extending the CPP δ' with the CPP δ^* after executing δ' from situation s. The axioms for such relation can be obtained by a straightforward reformulation of ext's axioms given in [9]. Technically, $ext_{CPP}(\delta', \delta, \delta^*, s)$ is defined to be logically equivalent to the conjunction of the following formulas:

$$\begin{aligned} \delta = nil &\supset \delta' = \delta^* \\ \delta = a &\supset \delta' = a;\delta^* \\ \delta = a;\delta_1 &\supset (\exists\delta_1'.\delta' = a;\delta_1' \wedge ext_{CPP}(\delta_1', \delta_1, \delta^*, do(a, s))) \\ \delta = \text{if } \phi \text{ then } \delta_2 \text{ else } \delta_3 &\supset (\exists\delta_2', \delta_3'.\delta' = \text{if } \phi \text{ then } \delta_2' \text{ else } \delta_3' \wedge \\ &\qquad \phi(s) \supset ext_{CPP}(\delta_2', \delta_2, \delta^*, s)) \wedge \\ &\qquad \neg\phi(s) \supset ext_{CPP}(\delta_3', \delta_3, \delta^*, s)) \end{aligned}$$

Property 4

$$\begin{aligned} &Trans(\Sigma_c\delta, s, \delta', s') \supset Final(\delta, s) \vee \\ &\qquad \exists\delta'', s'', \delta^*, s^*.Trans(\delta, s, \delta'', s'') \wedge Trans(\Sigma_c\delta'', s'', \delta^*, s^*) \end{aligned}$$

This property is closely related to Property 2 for Σ given in [10]. Intuitively, search can be seen as performing one single step while propagating itself to the program that remains after such step.

It is not surprising that sGolog solutions are solutions under conditional search as well. To show that, we make use of a one-place function $CATtoCPP$ that takes a CAT and returns its analogous CPP. We will refer with $Ax_{CATtoCPP}$ to the set of axioms defining such function, namely

$$\begin{aligned} CATtoCPP(\epsilon) &= nil \\ CATtoCPP(a \cdot c) &= a; CATtoCPP(c) \\ CATtoCPP([\phi, c_1, c_2]) &= \textbf{if } \phi \textbf{ then } CATtoCPP(c_1) \textbf{ else } CATtoCPP(c_2) \end{aligned}$$

Theorem 1. *Let δ be a sGolog program, and let σ be some history the agent has already committed to. Then, the set of axioms $Axioms \cup Sensed[\sigma] \cup Ax_{CAT} \cup Ax_{CATtoCPP}$ entails the following sentence:*

$$Do_s(\delta, end[\sigma], c) \supset Trans(\Sigma_c \delta^-, end[\sigma], CATtoCPP(c), end[\sigma])$$

The opposite, though, does not hold, because conditional search is more general than sGolog in that it allows for splittings *at any point*. In contrast, and as already stated, sGolog splits only at the points explicitly stated by the user via the special action $branch\_on$. As a matter of fact, the CAT c of Section 3.1 is a solution for $catch\_plane2b$, but not for $catch\_plane2$. On the other hand, program δ' above *is indeed a solution* for $(\Sigma_c catch\_plane2)$ itself, since Σ_c need not be told where to split.[7]

4.1 Restricted Conditional Search

We finish this section by noting that it is easy to slightly modify our axioms to define a restricted version of Σ_c, say Σ_{cb}, such that splittings in CPPs occurs *only* where the programmer has explicitly said so via a special action $branch\_on(\phi)$ (as done in sGolog.) The main motivation for defining Σ_{cb} is to provide a simple and clear semantics to our implementation.

We then make use of a special action $branch\_on(\phi)$, whose "effect" is to introduce a new conditional construct into the solution, i.e., into the CPP. Fortunately, we can achieve this by simply treating $branch\_on(\phi)$ as a normal primitive action that is always possible. Intuitively, a transition on a branch action is used to leave a "mark" in the situation term so as to force a conditional splitting at that point. Given that, at planning time, the branch action will be added to the situation term (as done with any other primitive action), we should guarantee that it has no effect on any of the domains fluents. In other words, every fluent in the domain should have the same (truth) value before and after a branch action.

In addition, we change the last axiom of function run to the following one:

$$\phi(s) \supset run(\text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2\text{,s}) = do(branch\_on(\phi), run(\delta_1, s))$$
$$\neg\phi(s) \supset run(\text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2\text{,s}) = do(branch\_on(\phi), run(\delta_2, s))$$

Now, a "run" of the program leaves a "mark" on the situation term, namely a $branch\_on(\phi)$ action term, to account for a conditional splitting.

It worth observing that, by using the same $Trans$ and $Final$ axioms given for Σ_c, all conditional constructs in the CPP solution are now required to perfectly coincide with the branch statements mentioned in the program. Finally, it is very important to remark that a branch action will never be mentioned in any CPP δ' obtained by search. In that sense, a $branch\_on(\phi)$ action can be viewed as a (meta-level) action whose direct effects are seen only at "planning time."

[7] However, under Σ_c, there may be strange solutions due to naive and useless splittings (e.g., splittings w.r.t. tautologies are always allowed.)

It is not difficult to prove that all four properties listed for Σ_c are properties of Σ_{cb} as well.[8] What is more important, it can be proved that Σ_{cb} and sGolog are equivalent for Golog programs. In addition, all solutions of Σ_{cb} are also solutions of Σ_c. We will refer with $Axioms'$, instead of $Axioms$, when using the modified axioms of Σ_{cb}.

Theorem 2. *Let δ_1 be an sGolog program and δ_2 a ConGolog one. Let σ be some history the agent has already committed to. Then, the set of axioms $Axioms' \cup Sensed[\sigma] \cup Ax_{CAT} \cup Ax_{CATtoCPP}$ entails the following sentence:*

$$Do_s(\delta_1, end[\sigma], c) \equiv Trans(\Sigma_{cb}\delta_1, end[\sigma], CATtoCPP(c), end[\sigma])$$

Furthermore, if $Axioms' \cup Sensed[\sigma] \models Trans(\Sigma_{cb}\delta_2, end[\sigma], \delta', s')$, then

$$Axioms \cup Sensed[\sigma] \models Trans(\Sigma_c\delta_2^-, end[\sigma], \delta', s')$$

Once again, the restricted version of search is not interesting in terms of the specification itself, as it is less general than Σ_c; but it is convenient in terms of implementation issues as we will see in the following section.

5 A Simple Implementation

In this section, we show a simple Prolog implementation of the restricted conditional search construct Σ_{cb} under two main assumptions borrowed from [9]: (i) only the truth value of relational fluents can be sensed; (ii) whenever a $branch\_on(P)$ action is reached, where P is a fluent, both truth values are conceivable for P. Assumption (ii) allows us to safely use hypothetical reasoning on the two possible truth values of P. For that, we use two auxiliary actions $assm(P)$ and $assm(neg(P))$ whose only effect is to turn P true and false respectively. We also assume the following code is already available:

1. A set of `trans/4` and `final/2` clauses constituting a correct implementation of $Trans$ and $Final$ predicates for all ConGolog constructs (see [7,18]);
2. A set of clauses implementing the underlying theory of action used. In particular, this set will include facts of the form `action(`a`)` and `fluent(`f`)` defining each action name a and each fuent name f respectively;
3. A set of `kwhether/2` clauses implementing predicate $Kwhether(P, s)$. For basic action theories, we can make a simplification by checking whether the fluent in question was sensed earlier and not changed since then [9]. For guarded theories, where inertia law may not apply, one may check that the fluent can be regressed up to a situation where a sensing axiom is applicable.

With all these assumptions, the restricted search implementation arises as a nice, but still not trivial, mixture between the implementation of sGolog and the

[8] Nonetheless we should replace $\Sigma\delta$ by $\Sigma\delta^-$ in Property 2; for, branch actions make no sense in the scope of Σ.

one for ConGolog. The reader will quickly notice that the code below reuses the clauses for $Trans$ and $Final$ of all the other constructs. Besides, it is independent of the background theory used, in particular independent on how sensing is modeled, as long as the above requirements are met.[9]

```
trans(searchcr(E),S,CPP,S):- build_cpp(E,S,CPP).
trans(branch_on(P),S,[],[branch_on(P)|S]).

build_cpp(E,S,[])      :- final(E,S).
build_cpp([E1|E2],S,C):- E2\=[], !, build_cpp(E1,S,C1),
                         ext_cpp(E2,S,C1,C).
build_cpp(branch_on(P),S,if(P,[],[])):- !, kwhether(P,S).
build_cpp(E,S,C)       :- trans(E,S,E1,[branch_on(P)|S]),
                         build_cpp([branch_on(P)|E1],S,C).
build_cpp(E,S,C)       :- trans(E,S,E1,S), build_cpp(E1,S,C).
build_cpp(E,S,[A|C])   :- trans(E,S,E1,[A|S]), fluent(P),
                         A\=branch_on(P), build_cpp(E1,[A|S],C).

/* ext_cpp(E,S,C,C1) recursively descends  the CPP C. On a      */
/* leaf, build_cpp/3 is used to extend the branch wrt program E.*/
ext_cpp(E,S,[A|C],[A|C2]):- action(A), ext_cpp(E,[A|S],C,C2).
ext_cpp(E,S,if(P,C1,C2),if(P,C3,C4)):-
 ext_cpp(E,[assm(P)|S],C1,C3), ext_cpp(E,[assm(neg(P))|S],C2,C4).
ext_cpp(E,S,[],C):- build_cpp(E,S,C). /* leaf of CPP */
```

Roughly speaking, build_cpp(δ, s, C) builds a CPP C for program δ at situation term s by calling `trans/4` to obtain a single step, and `ext_cpp/4` to extend intermediate already-computed CPPs. Relying on the correctness of `trans/4`, `final/2`, and `kwhether/2`, it is possible to show that the above program, which we will refer as P, is occur-check and floundering free [19].

Lemma 1. *Let δ be a ground ConGolog program term, and let s be a ground situation term. Then, the goal $G =$*build_cpp(δ, s, C) *is occur-check and floundering free w.r.t. program P, assuming a correct implementation of* `trans/4`, `final/2`, `action/1`, `fluent/1`, *and* `kwhether/2`. [10]

Finally, we show that whenever the above implementation succeeds, a conditional program plan supported by the specification as a legal solution of both Σ_{cb} and Σ_c is returned (by binding variable P below.) In contrast, whenever the implementation finitely fails, we can only guarantee that the specification of Σ_{cb} supports no solution at all.

[9] For legibility, we keep the translation between the theory and Prolog implicit.

[10] In reality, the program used will be P union the code for `trans/4`, `final/2`, `kwhether/2`, and the one implementing the underlying theory of action.

Theorem 3. *Let δ be a ground program term without mentioning search, and let σ be a history. Let G be the goal* `trans(`$searchcr(\delta), end[\sigma], P, S$`)`*. If G succeeds with computed answer $P = \delta', S = s'$, then δ' is a CPP, $s' = end[\sigma]$, and*

$$Axioms' \cup Sensed[\sigma] \models Trans(\Sigma_{cb}\delta, end[\sigma], \delta', s')$$
$$Axioms \cup Sensed[\sigma] \models Trans(\Sigma_{c}\delta^{-}, end[\sigma], \delta', s')$$

On the other hand, whenever G finitely fails, then

$$Axioms' \cup Sensed[\sigma] \models \forall \delta', s'. \neg Trans(\Sigma_{cb}\delta, end[\sigma], \delta', s')$$

It is worth noting that our results rely heavily on the implementation of `trans/4`, `final/2`, and `kwhether/2`. In particular, in order to assure correctness for the first two predicates, we may need to impose extra conditions on both programs and histories (e.g., see just-in-time histories and programs in [10, 18].)

Finally, we conjecture that it is possible to develop a better, and yet implementable, splitting strategy that does not rely on the user, and hence, does not use any special branching action. A plausible approach may be to split whenever the interpreter finds a condition ϕ that is not known at planning time. Clearly, this means that at least one fluent mentioned in ϕ is unknown; if the fluent will be known due to future sensing, we should branch w.r.t. to it. Observe that we should not only consider the conditions mentioned in the program, but all the formulas required to evaluate a transition (such as the actions' preconditions.) One point in favor of this strategy is that it is always sound w.r.t. Σ_c, due to the fact that Σ_c allows for any branching at any point, even for naive and unnecessary ones. Put differently, any solution reported by Prolog will be supported by the specification. On the other hand, it is not totally clear whether we can capture the branching power of Σ_c completely. Furthermore, this strategy will require considerable more computational effort during the search. Despite this difficulties, we think these ideas deserve future attention in pursuit of a more flexible and practical implementation.

6 Conclusions and Further Research

In this article, we have developed a new local lookahead construct for the Golog family of robot programs. The new construct provides local offline verification with sensing of ConGolog programs, produces complete conditional plans, and moreover, it mixes well with an interleaved account of execution. In some sense, the work here shows how easily one can extend Golog and ConGolog, together with their implementations, to handle *local contingent planning*.

Many problems remain open. First, it would be interesting to investigate some principled way of interleaving search in high-level programs since that determines how realistic, practical, and complete our programs are. Second, there is much to say regarding the relation between our search and the original one in [10]. For instance, neither subsumes completely the other. Nonetheless, it can be shown that, in some interesting cases, the original search Σ would actually execute an "implicit" CPP which Σ_c would support as a solution. Third, as already said, we would like to investigate some principled way of branching that does not rely on the user and still be implementable. Last, but not least, our approach may suggest the construction of more general (robot) plans than CPPs (in the sense of [4, 21, 22].) Indeed, solutions where the length of a branch is finite, but not bounded, cannot be captured with our conditional construct, but would be captured with a more general framework using loops (e.g., the cracking eggs example in [4].) There seems to be, however, a natural tradeoff between the expressivity in the theory and its corresponding computational complexity.

Acknowledgements
I am grateful to Hector Levesque for many helpful discussions and comments. Thanks also to Gerhard Lakemeyer for an early discussion on the subject of this paper, and to the anonymous referees for their valuable suggestions.

References

1. Baral, C., Son, T.C.: Approximate reasoning about actions in presence of sensing and incomplete information. In Maluszynski, J., ed.: International Logic Programming Symposium (ILSP' 97), Port Je erson, NY, MIT Press (1997) 387–401
2. De Giacomo, G., Levesque, H.: Projection using regression and sensors. In: Proceedingsof the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI), Stockholm, Sweden (1999) 160–165
3. Etzioni, O., Hanks, S., Weld, D.: An approach to planning with incomplete information. In: Proceedings of 3rd International Conference on Knowledge Representation and Reasoning. (1992)
4. Levesque, H.: What is planning in the presence of sensing? In: The Proceedings of the Thirteenth National Conference on Artificial Intelligence, AAAI-96, Portland, Oregon, American Association for Artificial Intelligence (1996) 1139–1146
5. Peot, M.A., Smith, D.E.: Conditional nonlinear planning. In: Proceedings of the First International Conference on AI Planning Systems, College Park, Maryland (1992) 189–197
6. Levesque, H., Reiter, R., Lesperance, Y., Lin, F., Scherl, R.: GOLOG: A logic programming language for dynamic domains. Journal of Logic Programming **31** (1997) 59–84
7. De Giacomo, G., Lespérance, Y., Levesque, H.: ConGolog, a concurrent programming language based on the situation calculus. Artificial Intelligence **121** (2000) 109–169
8. Smith, D., Weld, D.: Conformat graphplan. In: Proceedings of AAAI-98. (1998)
9. Lakemeyer, G.: On sensing and off-line interpreting in Golog. In: Logical Foundations for Cognitive Agents, Contributions in Honor of Ray Reiter. Springer, Berlin (1999) 173–187
10. De Giacomo, G., Levesque, H.: An incremental interpreter for high-level programs with sensing. In Levesque, H.J., Pirri, F., eds.: Logical Foundation for Cognitive Agents: Contributions in Honor of Ray Reiter. Springer, Berlin (1999) 86–102
11. Kowalski, R.A.: Using meta-logic to reconcile reactive with rational agents. In Apt, K.R., Turini, F., eds.: Meta-Logics and Logic Programming. MIT Press (1995) 227–242

12. Shanahan, M.: What sort of computation mediates best between perception and action? In Levesque, H., Pirri, F., eds.: Logical Fundations for Cognitive Agents: Contributions in Honor of Ray Reiter. Springer-Verlag (1999) 352–368
13. McCarthy, J., Hayes, P.J.: Some philosophical problems from the standpoint of artificial intelligence. Machine Intelligence **4** (1969) 463–502
14. Reiter, R.: Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems. MIT Press (2001)
15. Reiter, R.: The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Lifschitz, V., ed.: Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy. Academic Press, San Diego, CA (1991) 359–380
16. Lifschitz, V., McCain, N., Remolina, E. Tacchella, A.: Getting to the airport: The oldest planning problem in AI. Logic-Based Artificial Intelligence (2000) 147–165
17. Scherl, R., Levesque, H.: The frame problem and knowledge-producing actions. In: Proceedings of AAAI-93. (1993) 689–695
18. De Giacomo, G., Levesque, H.J., Sardiña, S.: Incremental execution of guarded theories. ACM Transactions on Computational Logic (TOCL) **2** (2001) To appear.
19. Apt, K.R., Pellegrini, A.: On the occur-check free prolog program. ACM Toplas **16** (1994) 687–726
20. Sardiña, S.: Local conditional high-level robot program (extended version). http://www.cs.toronto.edu/∼ssardina/papers/lchlrp-ext.ps (2001)
21. Smith, D.E., Williamson, M.: Representation and evaluation of plans with loops. In: Working Notes of the AAAI Spring Symposium on Extended Theories of Actions. Formal Theory and Practical Applications., Stanford, CA (1995)
22. Lin, S.H., Dean, T.: Generating optimal policies for high-level plan. In Ghallab, M., Milani, A., eds.: New Directions in AI Planning. IOS Press (1996) 187–200

A Proofs

A.1 Proofs of Properties

Proof (Property 1).

By definition of Σ_c, $Trans((\Sigma_c\delta_1)|(\Sigma_c\delta_2), s, \delta', s')$ is equivalent to

$$[\exists\delta''.Trans^*(\delta_1, s, \delta'', run(\delta', s)) \wedge Final(\delta'', run(\delta', s))\wedge$$
$$condPlan(\delta') \wedge knowHow(\delta', s)]$$
$$\vee$$
$$[\exists\delta''.Trans^*(\delta_2, s, \delta'', run(\delta', s)) \wedge Final(\delta'', run(\delta', s))$$
$$condPlan(\delta') \wedge knowHow(\delta', s)]$$

With that, the definition of $|$ for $Trans$ and $Final$, and the fact that $Trans^*$ is the transitive closure of $Trans$, we obtain the equivalent sentence:

$$\exists\delta''.Trans^*(\delta_1|\delta_2, s, \delta'', run(\delta', s)) \wedge Final(\delta'', run(\delta', s))\wedge$$
$$condPlan(\delta') \wedge knowHow(\delta', s)$$

which is, in fact, equivalent to $Trans(\Sigma_c(\delta_1;\delta_2), s, \delta', s')$. $\square$

Proof (Property 2). Assume that $Trans(\Sigma_c\delta, s, \delta', s')$ holds. Then, by definition of Σ_c,

$$\exists\delta''.Trans^*(\delta, s, \delta'', run(\delta', s)) \wedge Final(\delta'', run(\delta', s))$$

holds as well. If $\neg Final(\Sigma\delta, s)$ is the case, then $\neg Final(\delta, s)$ applies, and there must be at least one transition of δ. Formally,

$$\exists\delta'', \delta^*, s^*.Trans(\delta, s, \delta^*, s^*) \wedge Trans^*(\delta^*, s^*, \delta'', run(\delta', s)) \wedge Final(\delta'', run(\delta', s))$$

holds, and $\exists\delta'', s''.Trans(\Sigma\delta, s, \delta'', s'')$ follows easily. $\square$

Proof (Property 3). ($\Rightarrow$) Suppose that $Trans(\Sigma_c(\delta_1;\delta_2), s, \delta, s)$ holds. By definition of Σ_c,

$$\exists\delta''.Trans^*((\delta_1;\delta_2), s, \delta'', run(\delta, s)) \wedge Final(\delta'', run(\delta, s))$$

is entailed. Now, given that $Trans^*$ is no more than the transitive closure of $Trans$, we can truncate the CPP δ in each model M so as to obtain a new δ_1' that accounts *only* for the execution of δ_1 in M.

Consider then a particular model M. In M, there is a finite sequence of $Trans$' followed by a $Final$ for program $\delta_1;\delta_2$. By definition of $Trans$ for sequence, this implies a sequence of $Trans$' ended with a $Final$ for δ_1, followed by a sequence of $Trans$' and a $Final$ for δ_2. Clearly, this whole sequence is represented by a complete branch b in the CPP δ, since δ is a solution for $\Sigma_c(\delta_1;\delta_2)$. Roughly speaking, the new truncated CPP δ_1' is constructed by cutting branch

b as soon as an action on it correspond to a transition of δ_2 in M. Formally, $cut(\delta, M) = \delta'$ is defined inductively as

$$
\begin{aligned}
cut(nil, M) &= nil \\
cut((A;\delta), M) &= A; cut(\delta, M), \text{ if } A \text{ is due to a } \delta_1 \text{ transition in } M \\
cut((A;\delta), M) &= nil, \text{ if } A \text{ is due to a } \delta_2 \text{ transition in } M \\
cut(\text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2, M) &= \text{if } \phi \text{ then } cut(\delta_1, M) \text{ else } \delta_2, \text{ if } M[\phi] = true \\
cut(\text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2, M) &= \text{if } \phi \text{ then } \delta_1 \text{ else } cut(\delta_2, M), \text{ if } M[\phi] = false
\end{aligned}
$$

Notice that the truncation is performed in the third rule, as soon as we get to an action corresponding to a δ_2's step. Also, observe that the branches of δ not considered by M remain the same. It is not hard to see that there exists δ_1'' such that

$$Trans^*(\delta_1, s, \delta_1'', run(\delta_1', s)) \wedge Final(\delta_1'', run(\delta_1', s))$$

holds in M since $run(\delta_1', s)$ encodes the complete execution of δ_1 in M. Moreover, there is a CPP δ^*, namely the one cut from δ, that extends δ_1' and accounts for a complete execution of δ_2 starting at $run(\delta_1', s)$. Formally, there is a δ^* (the CPP removed from δ), and a program δ_2'' such that

$$Trans^*(\delta_2, run(\delta_1', s), \delta_2'', run(\delta^*, run(\delta_1', s))) \wedge Final(\delta_2'', run(\delta^*, run(\delta_1', s)))$$

Clearly, since δ is a CPP, so are δ_1' and δ^*. In addition, because $knowHow(\delta, s)$ holds in M, both $knowHow(\delta_1', s)$ and $knowHow(\delta^*, run(\delta_1', s))$ hold in M as well. For, the complete execution of δ in s is exactly the same as the one obtained by executing δ_1' frist followed by δ^*. Putting all these together, we get that

$$Trans(\Sigma_c\delta_1, s, \delta_1', s) \wedge Trans(\Sigma_c\delta_2, run(\delta_1', s), \delta^*, run(\delta_1', s))$$

holds in M. Finally, extending the CPP δ_1' with the CPP δ^* shields the CPP δ. For, δ_1' was obtained by removing δ^* from δ. As a consequence, $ext_{CPP}(\delta_1', \delta, \delta^*, s)$ is true in M.

Given that all this applies for every model M satisfying $Trans(\Sigma(\delta_1; \delta_2), s, \delta, s)$, the property follows.

($\Leftarrow$) This way is similar to the right-hand side. The point is that, in each model, it is possible to perform a sequence of transitions for δ_1 corresponding to a complete path in the CPP δ_1'. From that, it is possible to follow a path in the CPP δ^*, which extends δ_1', corresponding to transitions for program δ_2. The important thing is that the CPP δ will account for all the necessary extensions at the leafs of the CPP δ_1' w.r.t. program δ_2.

In that sense, following the CPP δ is the same as following first δ_1' as a solution to δ_1, and then following δ^* as a solution for δ_2. $\square$

Proof (Property 4).

Suppose that $Trans(\Sigma_c\delta, s, \delta', s')$ holds. By definition of Σ_c,

$$\exists\delta''.Trans^*(\delta_1, s, \delta'', run(\delta', s)) \wedge Final(\delta'', run(\delta', s))$$

is entailed. Take a model M and suppose that $Final(\delta, s)$ is not true. Thus, there has to be at least one $Trans$ step of δ_1, i.e., either

$$\exists \delta'', \delta^*. Trans(\delta, s, \delta^*, s) \wedge Trans^*(\delta^*, s, \delta'', run(\delta', s)) \wedge Final(\delta'', run(\delta', s))$$

is true in M (due to a test transition), or

$$\begin{aligned}&\exists \delta'', \delta''', \delta^*, a. Trans(\delta, s, \delta^*, do(a, s)) \wedge \\ &\quad Trans^*(\delta^*, do(a, s), \delta'', run(\delta''', do(a, s))) \wedge Final(\delta'', run(\delta''', do(a, s)))\end{aligned}$$

is true in M (due to an action transition.) Notice that δ''' is the CPP that remains after performing one single step of δ' w.r.t. the model. In either case,

$$\exists \delta''. \delta''', \delta^*. Trans(\delta, s, \delta^*, s'') \wedge Trans(\Sigma_c \delta^*, s'', \delta''', s'')$$

Because this applies for any model M, the property follows. $\square$

A.2 Proofs of Theorems

Proof (Lemma 1).

We will need the following terminology: A *mode* for an n-ary predicate symbol p is a function $m_p : \{1, ..., n\} \rightarrow \{+, -\}$. Positions mapped to $'+'$ are called *input* positions of p, and positions mapped to $'-'$ are called *output* positions of p. Intuitively, queries formed by predicate p will be expected to have input positions occupied by ground terms. We write m_p in the form $p(m_p(1), ..., m_p(n))$. A family of terms is *linear* if every variable occurs at most once on it. A clause is (input) output linear if the family of terms occurring in all (input) output positions of its body is linear.

An *input-output specification* for a program P is a set of modes, one for each predicate symbol in P. A clause (goal) is *well-moded* if every variable occurring in an input position of a body goal occurs either in an input position of the head, or in an output position of an earlier body goal; and every variable occurring in an output position of the head occurs in an input position of the head, or in an output position of a body goal. A goal can be viewed as a clause with no head and we will be interested only in goals with one atom, i.e. $G = \leftarrow A$. A program is called *well-moded* w.r.t. its input-output specification if all its clauses are. The definition of well-moded program constrains "the flow of data" through the clauses of the program. Lastly, a clause (goal) is *strictly moded* if it is well-moded and output linear, and a program is *strictly moded* if every rule of it is.

It was proved in Apt and Pellegrini [[19], Corollary 4.5] that well-moded and output linear programs (for some input-output specification) are occur-check free w.r.t. well-moded goals. It was also proven there (Corollary 6.5) that a program P is occur-check free w.r.t. a goal G if both P and G are strictly moded. Finally, Theorem 8.5 in [19] says that if P_D and G are well moded and all predicate symbols occurring under *not* in P_D and G are moded completely input, then $P_D \cup \{G\}$ does not flounder.

Let M be the follwing mode for program P:

```
trans(+,+,-,-), final(+,+), =(+,+), prim_action(+), kwhether(+,+),
                    build_cpp(+,+,-), ext_cpp(+,+,+,-)
```

In the mode M, both P and G are well-moded. Moreover, every clause of P is output linear. Then, by Corollary 4.5 in [19], $P \cup \{G\}$ is occur-check free.

Also, the only relation that appears in negative literals of P, namely `=/2`, is moded completely input. By Theorem 8.5 in [19], we conclude that $P \cup \{G\}$ does not flounder. □

Proof (Theorem 1). By induction on the structure of the program δ. For simplicity, let us refer with S to the situation term $end[\sigma]$.

Base Case: consider the case $\delta = A$, where A is an action. Given that $Do_s(A, S, c)$ is entailed, it follows that $Axioms \cup Sensed[\sigma] \cup Ax_{CAT} \models Poss(A, S) \wedge c = A$. First, it is trivial to check that $Axioms \models condPlan(A) \wedge knowHow(A, S)$. Second, from the definitions of $Trans$ and $Final$, it follows that $Axioms \cup Sensed[\sigma] \models Trans(A, S, nil, do(A, S)) \wedge Final(nil, do(A, S))$. From the fact that $CATtoCPP(A) = A$, we get that $Axioms \cup Sensed[\sigma] \models Trans(\Sigma_c A, S, A, S)$.

The cases for $\delta = ?(\phi)$ and $\delta = nil$ are similar.

Induction Step: we will only show the cases of nondeterministic choice of program and sequence. For the latter one, assume $\delta = \delta_1 | \delta_2$. Since $Do_s(\delta_1 | \delta_2, S, c)$ holds, $Axioms \cup Sensed[\sigma] \cup Ax_{CAT} \models Do_s(\delta_1, S, c) \vee Do_s(\delta_2, S, c)$. It is not hard to see that we can safely apply the induction hypothesis to get that $Axioms \cup Sensed[S] \cup Ax_{CAT} \cup Ax_{CATtoCPP}$ entails

$$Trans(\Sigma_c \delta_1, S, CATtoCPP(c), S) \vee Trans(\Sigma_c \delta_2, S, CATtoCPP(c), S)$$

Using Property 1, $Trans(\Sigma_c \delta_1, S, CATtoCPP(c), S)$ follows.

Finally, consider the case $\delta = \delta_1; \delta_2$. From the definition of sGolog, the set of axioms $Axioms \cup Sensed[\sigma] \cup Ax_{CAT}$ entails

$$\exists c'.Do(\delta_1, S, c') \wedge \exists c^*.Do(\delta_2, cdo(c', S), c^*) \wedge ext(c', c, c^*, S)$$

By induction, we get that $Axioms \cup Sensed[\sigma] \cup Ax_{CAT} \cup Ax_{CATtoCPP}$ entails

$$\begin{aligned}&\exists c'.Trans(\Sigma \delta_1, s, CATtoCPP(c'), S) \wedge \\ &\exists c^*.Trans(\Sigma \delta_2, cdo(c', s), CATtoCPP(c^*), S) \wedge ext(c', c, c^*, s)\end{aligned}$$

Using the definition of Σ_c, the following is entailed:

$$\begin{aligned}\exists c', \delta_1''.&Trans^*(\delta_1, S, \delta_1'', run(CATtoCPP(c'), S)) \wedge \\ &\qquad Final(\delta_1'', run(CATtoCPP(c'), S)) \wedge \\ &\exists c^*, \delta_2''.Trans^*(\delta_2, cdo(c', S), \delta_2'', run(CATtoCPP(c^*), cdo(c', S))) \wedge \\ &Final(\delta_2'', run(CATtoCPP(c^*), cdo(c', S))) \wedge ext(c', c, c^*, S)\end{aligned}$$

Next, from the fact that $cdo(c, s) = run(CATtoCPP(c), s)$, the following is entailed as well:

$$\exists c', c^*, \delta_2''.Trans^*((\delta_1;\delta_2), S, \delta_2'', run(CATtoCPP(c^*), cdo(c', S))) \wedge$$
$$Final(\delta_2'', run(CATtoCPP(c^*), cdo(c', S))) \wedge ext(c', c, c^*, S)$$

Finally, since c is an extension of c' by means of each c^* from each model, we know that, in each of such models, it is the case that

$$run(CATtoCPP(c^*), cdo(c', S)) = cdo(c, S) = run(CATtoCPP(c), S)$$

Hence, $Axioms \cup Sensed[\sigma] \cup Ax_{CAT} \cup Ax_{CATtoCPP}$ entails

$$\exists \delta_2''.Trans^*((\delta_1;\delta_2), S, \delta_2'', run(CATtoCPP(c), S)) \wedge$$
$$Final(\delta_2'', run(CATtoCPP(c), S))$$

Putting this together with the fact that $CATtoCPP(c)$ is clearly a CPP, and that $knowHow(CATtoCPP(c), S)$ is entailed as well, we conclude that

$$Axioms \cup Sensed[\sigma] \cup Ax_{CAT} \cup Ax_{CATtoCPP} \models$$
$$Trans(\Sigma(\delta_1;\delta_2), S, CATtoCPP(c), S)$$

□

Proof (Theorem 2 (first part)).

($\Rightarrow$) the proof is the same as the one for Theorem 2 with the following modifications: (i) $Axioms$ is replaced by $Axioms'$; (ii) Σ_c is replaced by Σ_{cb}; and (iii) program δ is replaced by program δ^-.

($\Leftarrow$) By induction on the structure of the program δ. For simplicity, let us refer with S to the situation term $end[\sigma]$.

Base Case: consider the case $\delta = A$, where A is a domain action. Given that $Trans(\Sigma_{cb} A, S, CATtoCPP(c), S)$ is entailed, it should be the case that $CATtoCPP(c) = A$, and hence, $c = A$. Thus, $Trans(A, S, nil, do(A, S))$ holds, which implies that $Poss(A, S)$ holds. As a result, $Do_s(A, S, A)$ applies.

Consider next the case $\delta = branch\_on(\phi)$. Given the fact that the set of axioms entails $Trans(\Sigma_{cb} branch\_on(\phi), S, CATtoCPP(c), S)$, it should be the case that $CATtoCPP(c) =$ (if ϕ then nil else nil), and hence, $c = [\phi, \epsilon, \epsilon]$.

Also, since $knowHow(CATtoCPP(c)), s$ holds, we know that $kwhether(\phi, S)$ is true. By the macro expansion definition of Do_s, $Do_s(branch\_on(\phi), S, [\phi, \epsilon, \epsilon])$ is entailed.

The cases for $\delta = ?(\phi)$ and $\delta = nil$ are similar.

Induction Step: we will show the case for sequence. Assume then that $\delta = \delta_1;\delta_2$, and that $Trans(\Sigma_{cb}(\delta_1;\delta_2), S, CATtoCPP(c), S)$ is entailed from the axioms. By Property 3,

$$\exists \delta_1'.Trans(\Sigma_{cb}\delta_1, S, \delta_1', S) \wedge \exists \delta^*.Trans(\Sigma_{cb}\delta_2, run(\delta_1', S), \delta^*, run(\delta_1', S)) \wedge$$
$$ext_{CPP}(\delta_1', CATtoCPP(c), \delta^*, S)$$

holds. Next, let c_1' and c^* be two CATs such that $CATtoCPP(c_1') = \delta_1'$ and $CATtoCPP(c^*) = \delta^*$ respectively. Using the induction hypothesis,

$$\exists c_1'.Do_s(\delta_1, S, c_1') \wedge \exists c^*.Do_s(\delta_2, run(CATtoCPP(c_1'), S), c^*) \wedge$$
$$ext_{CPP}(CATtoCPP(c_1'), CATtoCPP(c), CATtoCPP(c^*), S)$$

is entailed by $Axioms' \cup Sensed[\sigma] \cup Ax_{CATtoCPP} \cup Ax_{CAT}$. The final step is simple and based on the fact that $run(CATtoCPP(c), s) = cdo(c, s)$, and that

$$Axioms' \cup Sensed[\sigma] \cup Ax_{CATtoCPP} \cup Ax_{CAT} \models$$
$$ext_{CPP}(CATtoCPP(c_1), CATtoCPP(c_2), CATtoCPP(c_3), s) \equiv ext(c_1, c_2, c_3, s)$$

for every situation s and CATs c, c_1, c_2, c_3. This is saying that run and ext_{CPP} are the analogous of sGolog cdo and ext. As a result,

$$Axioms' \cup Sensed[\sigma] \cup Ax_{CATtoCPP} \cup Ax_{CAT} \models$$
$$\exists c_1'.Do_s(\delta_1, S, c_1') \wedge \exists c^*.Do_s(\delta_2, cdo(c_1', S), c^*) \wedge ext(c_1', c, c^*, S)$$

□

Proof (Theorem 2 (second part)). The second part of Theorem 2 is not hard and follows easily by inspecting the differences between Σ_c and its restricted version Σ_{cb}.

If δ' is a solution for $\Sigma_{cb}\delta_2$, then it should also be a solution for $\Sigma_c\delta_2^-$ given that the only difference between them is that Σ_c allows splittings *at any point*.

1. Clearly, $Axioms \cup Sensed[\sigma] \models condPlan(\delta') \wedge knowHow(\delta', end[\sigma])$ as they hold for $Axioms'$;
2. Every path in δ' is represented by a successful sequence of $Trans$' followed by a $Final$ w.r.t. $Axioms \cup Sensed[\sigma]$ and program δ_2^-. This follows trivially from the fact that δ_2^- is δ_2 with all its $branch\_on$ actions suppressed, and the fact that the run function does not introduce any branch term action into the situation term. Hence, the sequence of $Trans$' followed by the $Final$ for each path on δ' is the same sequence of that for δ_2 except that all transitions of $branch$ actions are discarded.

Putting both 1 and 2 together, we conclude

$$Axioms \cup Sensed[\sigma] \models Trans(\Sigma_c\delta_2^-, end[\sigma], \delta, end[\sigma])$$

□

Proof (Theorem 3).

First part: This is proved by induction on the number of calls to `build_cpp/3` and relying on the soundness of `trans/3` and `final/2`. We will refer with s to the corresponding sequence of actions representing the situation term $end[\sigma]$.

The base case is when only one call to `build_cpp/3` is performed, namely the call in the `trans/4` rule for $searchcb(E)$. In that case, goal succeeds with the first rule of `build_cpp/3`, and soundness is obtained trivially from the soundness of `final/2`.

For the induction step, suppose the goal `build_cpp`(δ, s, C) succeeds with $n > 1$ calls to `do/4`. Then, one of the following cases applies:

1. Case 1: $\delta = \delta_1; \delta_2$ and goal $\texttt{build\_cpp}(\delta_1; \delta_2, s, C)$ succeeds with $C = \delta'$. First, $\texttt{build\_cpp}(\delta_1, s, C1)$ succeeds with $C1 = \delta'_1$. By induction hypothesis, $Axioms \cup Sensed[\sigma] \models Trans(\Sigma_{cb}\delta_1, s, \delta'_1, s)$. Also, $\texttt{ext\_cpp}(\delta_2, s, \delta'_1, C)$ succeeds with $C = \delta'$ itself. Now, by inspecting the three rules for `do/4`, it is not hard to see that C is obtained by extending each possible path of the CPP δ' with a new conditional program plan. This is done by reasoning by cases at each branch point of δ' (legal due to assumption (ii) at the beginning of Section 5.) By induction, every path extension of δ' is sound, and `do/4` bounds C to a conditional plan that completely extends (i.e., extends every path) the CPP returned as a δ_1 solution, i.e., the CPP δ'_1, with new and sound CPPs at every leaf. Formally,

$$\exists \delta^*. Trans(\Sigma_{cb}\delta_2, run(\delta'_1, s), \delta^*, run(\delta'_1, s)) \land ext_{CPP}(\delta'_1, \delta, \delta^*, s)$$

   is entailed by $Axioms \cup Sensed[\sigma]$. Hence, by Property 2 for Σ_{cb},

$$Axioms \cup Sensed[\sigma] \models Trans(\Sigma_{cb}(\delta_1; \delta_2), end[\sigma], \delta', end[\sigma])$$

2. Case 2: $\delta = branch\_on(f)$ and goal $\texttt{build\_cpp}(branch\_on(f), s, C)$ succeeds with $C = \delta' = if(f, nil, nil)$, where f is a domain fluent. Thus, the soundness of search follows directly from the soundness of call $\texttt{kwhether}(f, s)$.
3. Case 3: the fourth rule for `build_cpp/3` succeeds. In such a case, the sub-goal $\texttt{trans}(\delta, s, E1, [branch\_on(P)|s])$ succeeds with $P = f$ and $E1 = \delta^*$ for some fluent f and program δ^*. Moreover, $\texttt{build\_cpp}([branch\_on(f)|\delta^*], s, C)$ succeeds with $C = \delta'$. The key point is the fact that if δ makes a $branch\_on(f)$ step with a remaining program δ^*, then a solution for $\Sigma_{cb}(branch\_on(f); \delta^*)$ is also a solution for $\Sigma_{cb}\delta$ itself. Intuitively, any potential legal next action can be moved in front of the whole program safely, i.e., the solutions of the transformed program will also be solutions of the original one (note the converse is not true.) Knowing that, we simply have to use the induction hypothesis on the new call $\texttt{build\_cpp}([branch\_on(f)|\delta^*], s, C)$ to reconstruct the soundness of the original call to `build_cpp/3`.
4. Case 4 and 5: either the fifth or the sixth rule succeeds. Those clauses correspond to transitions of non-sequence programs where the transition involves a test condition $?(\phi)$ or the execution of a domain action A. Using the induction hypothesis on the calls to `build_cpp/3`, together with the soundness of `trans/4`, we get the soundness of the original call to `build_cpp/3`.

It follows then that $Axioms' \cup Sensed[\sigma] \models Trans(\Sigma_{cb}\delta, end[\sigma], \delta', end[\sigma])$. Also, $Axioms \cup Sensed[\sigma] \models Trans(\Sigma\delta^-, end[\sigma], \delta', end[\sigma])$ follows directly using Theorem 2.

Second part: Here, we prove that the top-level call to `trans/4` finitely fails, then the specification supports no solution w.r.t. Σ_{cb}. Notice we cannot guarantee that for Σ_c since Σ_c is more general and may find solutions by splitting arbitrarily.

The proof is, again, by induction on the number of calls to `build_cpp/3` in the finitely failed SLDNF-tree. The base case is when only one call to `build_cpp/3` is needed, namely, the call in the `trans/4` rule. In that case, either $\delta = A$, and A is not possible; $\delta = ?(\phi)$, and ϕ does not hold; or $\delta = branch\_on(f)$ for some fluent f that is unknown at s. All these cases are straightforward in that we only need to refer to the (assumed) soundness of `trans/4`, `final/2`, and `kwhether/2`.

For the induction step, suppose the goal `build_cpp(`δ, s, C`)` finitely fails with $n > 1$ calls to `build_cpp/3`. Then, one of the following cases applies:

1. Case 1: $\delta = \delta_1;\delta_2$. The only eligible `build_cpp/3` rule is the second one, and either (i) the sub-goal `build_cpp(`δ_1, s, C`)` finitely fails; or (ii) the sub-goal `build_cpp(`$\delta_1;\delta_2, s, C1$`)` succeeds with computer answer $C1 = \delta'$, but the sub-goal `ext_cpp(`$\delta_2, s, \delta_1', C$`)` finitely fails.
   In the first case, we apply the induction hypothesis to get that $Axioms' \cup Sensed[\sigma] \models \forall \delta', s'.\neg Trans(\Sigma_{cb}\delta_1, s, \delta', s')$.
   The second case deserves a little more attention. Since `ext_cpp(`$\delta_2, s, \delta_1', C$`)` finitely fails, it has to be the case that some complete path of the CPP δ_1' from situation s cannot be extended with a valid CPP for δ_2. In other words, after traversing a complete path of δ_1', the third rule of `ext_cpp/4` finitely fails, because its body call to `build_cpp/3` finitely fails when trying to extend the path by using program δ_2. Hence,

   $$Axioms' \cup Sensed[\sigma] \models \neg\exists\delta^*.Trans(\Sigma_{cb}\delta_2, run(\delta_1', s), \delta^*, run(\delta_1', s))$$

   since there is at least one complete path of δ_1 for which there is no extension w.r.t. δ_2.
   In either case (i) or (ii), by using Property 3, we conclude that

   $$Axioms' \cup Sensed[\sigma] \models \forall\delta', s'.\neg Trans(\Sigma_{cb}(\delta_1;\delta_2), end[\sigma], \delta', s')$$

2. Case 2: In this case, δ is not a sequence program, and after all possible transitions via `trans/4` in the third, fourth, and fifth rules of `build_cpp/3`, the corresponding sub-goal call to `build_cpp/3` finitely fails. Given that we assumed a correct `trans/4` implementation, by the induction hypothesis on each sub-goal call to `build_cpp/3`, we conclude that

   $$\forall\delta'', s''.Trans(\delta, s, \delta'', s'') \supset \forall\delta^*, s^*.\neg Trans(\Sigma_{cb}\delta'', s'', \delta^*, s^*)$$

   is entailed by the specification. Notice that the failure of the third clause for `build_cpp/3` is a bit complicated; by moving the branch action to the front of the program, the interpreter will try to build two separated CPPs, one for each truth value of the branch-fluent. As a consequence of that, the finitely failure of the sub-goal call to `build_cpp/3` (in the third clause of `build_cpp/3`) means that, for some truth value of the fluent in question, there is no legal CPP extension. By Property 4,

   $$Axioms' \cup Sensed[\sigma] \models \forall\delta', s'.\neg Trans(\Sigma_{cb}\delta, end[\sigma], \delta', s')$$

$\square$