ConGolog, Sin Trans: Compiling ConGolog into Basic Action Theories for Planning and Beyond (extended version) [*]

Christian Fritz Jorge A. Baier Sheila A. McIlraith
Department of Computer Science,
University of Toronto,
Toronto, ON M5S 3G4, Canada

Abstract

ConGolog is a logical programming language for agents that is defined in the situation calculus. ConGolog agent control programs were originally proposed as an alternative to planning, but have also more recently been proposed as a means of providing domain control knowledge for planning. In this paper, we present a compiler that takes a ConGolog program and produces a new basic action theory of the situation calculus whose executable situations are all and only those that are permitted by the program. The size of the resulting theory is quadratic in the size of the original program – even in the face of unbounded loops, recursion, and concurrency. The compilation is of both theoretical and practical interest. From a theoretical perspective, proving properties of ConGolog programs is simplified because reification of programs is no longer required, and the compiled theory contains fewer second-order axioms. Further, in some cases, properties can be proven by regressing the program to the initial situation, eliminating the need for a higher order theorem prover. From a practical perspective, the compilation provides the mathematical foundation for compiling ConGolog programs into classical planning problems, including, with minor restrictions, into the Plan Domain Definition Language (PDDL), which is used as the input language for most state-of-the-art planners. Moreover, Hierarchical Task Networks (HTNs), a popular planning paradigm for industrial applications can be represented as ConGolog programs and can thus now also be compiled to a classical planning problem. Such compilations are significant because they allow the best state-of-the-art planners to exploit ConGolog and HTN search control, without the need for special-purpose machinery.

[*]Technical Report CSRG-576, University of Toronto

1 Introduction

ConGolog [De Giacomo *et al.*, 2000] is a logical programming language for specifying high-level agent control, that is defined in the situation calculus. It extends the agent programming language Golog [Levesque *et al.*, 1997] by concurrent program execution. Golog's Algol-inspired programming constructs allow a user to program an agent's behavior while leaving parts of the program under-constrained, or "open", through the use of non-deterministic constructs. These under-constrained regions of the program are later filled in by a planner. Such integration of planning and programming has proved useful in a variety of diverse applications including soccer playing robots [Ferrein *et al.*, 2004], museum tour-guide robots [Burgard *et al.*, 1999], and Web service composition [McIlraith and Son, 2002].

By way of illustration, consider a simple delivery problem in which we have an (infinite capacity) truck and the task is to deliver packages from point A to point B. A classical planning problem would simply specify the initial state and the goal state. Using ConGolog, we can provide the following program that constrains the space of possible plans, while still leaving some work to the planner:

> *If not at point A, drive the truck to point A; while there are packages at point A, pick a package and load it onto the truck; drive to point B; while there are packages on the truck, pick a package and unload it from the truck.*

A basic action theory of the situation calculus induces a tree of possible action sequences or situations. A ConGolog program further constrains the tree to those that adhere to the program. However, in order to reason about the satisfaction of these constraints, a system requires special-purpose machinery – it needs to interpret the ConGolog program. This for instance applies to planning, but also to other problems that involve reasoning about feasible trajectories, including the problem of monitoring the continued validity of an executing plan (see, e.g., [Fritz and McIlraith, 2007]).

1.1 Contributions

We propose an algorithm for *compiling* ConGolog programs into basic action theories of the situation calculus whose tree of executable situations corresponds exactly to the one described by the program. We prove the correctness of the compilation and show that its output is of size as most quadratic in the size of the original program.

The compiled theory allows us to reason about the executions of programs using regression. Given an action sequence, we can "regress a program" over this sequence, producing a necessary and sufficient condition for the sequence to be a legal execution of the program.

The compilation is significant for a number of practical and theoretical reasons. From a practical perspective, the compilation provides the mathematical foundation for compiling ConGolog control knowledge into the Planning Domain Definition Language (PDDL) [McDermott, 1998], a *de facto* standard planning problem specification language. This in turn enables state-of-the-art planners to exploit powerful control knowledge without the need for special-purpose machinery within their planners. We have recently shown how this can be done for a subset of the language without concurrency and procedures [Baier *et al.*, 2007]. The experimental results showed that state-of-the-art planners can gain significant speed-ups from that. The current compilation of ConGolog (including concurrency and procedures) can be seen as an extension of this work – though some restrictions apply when compiling into PDDL.

ConGolog has been used for a variety of purposes, all of which can now benefit from this newly built connection to modern planners. For instance, Hierarchical Task Networks (HTN) have been translated to ConGolog [Gabaldon, 2002]. In combination, this translation and our compiler provide the means for compiling HTN control knowledge into a classical planning problem. We anticipate this contribution to be of significant interest to the planning community.

From a theoretical perspective, the compilation eliminates the need for ConGolog's tedious reification of programs, as well as the second-order axioms necessitated by its transition semantics. This facilitates proving properties of programs (e.g., reachability, invariants, termination). Further, since programs themselves can now be regressed, some proofs can be reduced to first-order theorem proving through the use of regression.

In this chapter we focus on the high-level idea of the compilation. The actual pseudo-code can be found in Appendix 6, and experimental evidence in support of our basic approach can be found in [Baier *et al.*, 2007].

2 Background

2.1 The Situation Calculus

The situation calculus is a family of many-sorted logical languages for specifying and reasoning about dynamical systems. It was first proposed by McCarthy [1963] and later significantly extended by Reiter [2001], most importantly by providing a solution to the frame problem (see below). In this thesis we use Reiter's situation calculus.

Its basic elements are situations, primitive actions (sort $\mathcal{A}$), and fluents (sort $\mathcal{F}$). A situation is a *history* of the primitive actions performed from a distinguished initial situation S_0. The function $do(a, s)$ denotes the situation resulting from performing action a in situation s, inducing a tree of situations rooted in S_0. Fluents are relations and functions that take a situation as their last argument (e.g., $F(\vec{x}, s)$), and are used to define the state of the world.

For readability, action and fluent arguments are generally suppressed. Also, $do(a_n, do(a_{n-1}, \ldots do(a_1, s)))$ is abbreviated to $do([a_1, \ldots, a_n], s)$ or $do(\vec{a}, s)$ and we define: $do([\,], s) \overset{\text{def}}{=} s$.

2.1.1 Basic Action Theories

A *basic action theory* in the situation calculus, $\mathcal{D}$, is comprised of the following sets of axioms [Levesque *et al.*, 1998]:

- Σ the set of domain independent foundational axioms of the situation calculus, including one second-order induction axiom required to properly define the tree of situations. These axioms are as follows:

$$\begin{aligned}
&do(a_1, s_1) = do(a_2, s_2) \supset a_1 = a_2 \wedge s_1 = s_2,\\
&(\forall P).P(S_0) \wedge (\forall a, s)[P(s) \supset P(do(a, s))] \supset (\forall s)P(s),\\
&\neg s \sqsubset S_0,\\
&s \sqsubset do(a, s') \equiv s \sqsubseteq s'.
\end{aligned}$$

  Here the relation $\sqsubset$ provides an ordering on situations, and $s \sqsubseteq s'$ abbreviates $s = s' \vee s \sqsubset s'$.

- $\mathcal{D}_{ss}$, *successor state axioms*, provide a parsimonious representation of frame and effect axioms under an assumption of the completeness of the axiomatization. There is one successor state axiom for each fluent, $F \in \mathcal{F}$, of the form $F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$, where $\Phi_F(\vec{x}, a, s)$ is a formula with free variables among $\vec{x}, a, s$. $\Phi_F(\vec{x}, a, s)$ characterizes the truth value of the fluent $F(\vec{x})$ in the situation $do(a, s)$ in terms of what is true in situation s. These axioms can be automatically generated from effect axioms, as described below.

- $\mathcal{D}_{ap}$, *action precondition axioms*, first-order axioms that specify the conditions under which actions are possible. There is one axiom for each action $a \in \mathcal{A}$ of the form $Poss(a(\vec{x}), s) \equiv \Pi_a(\vec{x}, s)$ where $\Pi_a(\vec{x}, s)$ is a formula with free variables among $\vec{x}, s$.

- $\mathcal{D}_{una}$, a set of *unique name axioms* for actions;

- $\mathcal{D}_{S_0}$ a set of sentences relativized to situation S_0, specifying what is true in the initial state.

Although any situation calculus action theory is second-order, many reasoning tasks can be reduced to first-order theorem proving by using regression [Reiter, 2001]. Properties that hold in all executable situations can be shown by induction over situations [Reiter, 1993].

2.1.2 The Frame Problem and A Solution for Deterministic Actions

To the user, specifying the effects of actions can be more natural when using *effect axioms*. For a relational fluent $F(\vec{x}, s)$, for example, positive and negative effect axioms can define the conditions under which the fluent becomes true ($\phi^+(\vec{x}, s)$), respectively false ($\phi^-(\vec{x}, s)$), after performing action a:

$$\begin{aligned}
&\phi^+(\vec{x}, s) \supset F(\vec{x}, do(a, s)),\\
&\phi^-(\vec{x}, s) \supset \neg F(\vec{x}, do(a, s)).
\end{aligned}$$

These axioms describe the effects on the considered fluents, but they do not describe the *non-effects* on all other fluents. Axioms describing the latter are called *frame axioms*. The *frame problem* states the impossibility of stating and reasoning with all frame axioms explicitly, due to their cardinality: Even apparently nonsensical assertions, like "drinking water does not change one's hair color" would have to be captured by a frame axiom:

$$haircolor(do(drinkwater, s)) = y \leftarrow haircolor(s) = y.$$

Ray Reiter [Reiter, 1991] proposed a solution to the frame problem based on a completeness assumption, namely that the provided effect axioms specify *all* possible ways by which a fluent may change. In Reiter's solution, the set of all effect axioms, is hence syntactically transformed into the set of successors state axioms (one for each fluent).

In the following we describe how Reiter's solution applies to *functional fluents*, for relational fluents the computations are similar and can be found in [Reiter, 1991]. The effect axiom for a functional fluent f and action α has the form:

$$\phi_f(\vec{t}, y, s) \supset f(\vec{t}, do(\alpha, s)) = y$$

where $\vec{t}$ are terms not mentioning situation terms. Note that, unlike relation fluents, for functional fluents there are no positive and negative effect axioms but only one axiom explicitly stating the new value (y) of the fluent. Above formula can be rewritten to:

$$\underbrace{a = \alpha \wedge \vec{x} = \vec{t} \wedge \phi_f(\vec{x}, y, s)}_{\Phi_f} \supset f(\vec{x}, do(a, s)) = y$$

and this can be done for all n effect axioms for fluent f. These axioms can then be merged into a single *normal form* for this fluent:

$$\begin{array}{rcll} \Phi_f^{(1)} \vee \cdots \vee \Phi_f^{(n)} & \supset & f(\vec{x}, do(a, s)) = y, & \text{or} \\ \gamma_f(\vec{x}, y, a, s) & \supset & f(\vec{x}, do(a, s)) = y & \end{array} \tag{1}$$

The completeness assumption expresses that if fluent f changes its value from situation s to situation $do(a, s)$, then $\phi_f(\vec{x}, y, a, s)$ must be true:

$$f(\vec{x}, s) = y' \wedge f(\vec{x}, do(a, s)) = y \wedge y \neq y' \quad \supset \quad \gamma_f(\vec{x}, y, a, s) \tag{2}$$

Together with the assumption

$$\neg(\exists \vec{x}, y, y', a, s).\gamma_f(\vec{x}, y, a, s) \wedge \gamma_f(\vec{x}, y', a, s) \wedge y \neq y'$$

Reiter shows that (1) and (2) are logically equivalent to:

$$\begin{array}{rcl} f(\vec{x}, do(a, s)) = y & \equiv & \gamma_f(\vec{x}, y, a, s) \vee \\ & & f(\vec{x}, s) = y \ \wedge (\nexists y').\gamma_f(\vec{x}, y', a, s) \wedge y \neq y' \end{array} \tag{3}$$

which is the successor state axiom for functional fluent f.

2.1.3 Notation and Definitions

Lower case letters denote variables in the theory of the situation calculus, upper case letters denote constants. We use α to denote arbitrary but explicit actions and S to denote arbitrary but explicit situations, that is $S = do(\vec{\alpha}, S_0)$ for some explicit action sequence $\vec{\alpha}$. Variables that appear free are implicitly universally quantified unless stated otherwise. By $\psi[x/y]$ we denote the formula resulting from substituting all occurrences of x in ψ with y. Further, $\vec{a} \cdot a$ denotes the result of appending action a to the sequence $\vec{a}$.

For two situations s, s', such that $s \sqsubseteq s'$, we say that s is a *sub-history* of s', or s' is a *continuation* of s.

We say that a situation s is *executable*, denoted as $executable(s)$, if all actions in the history of s have their preconditions satisfied in the situation where they are performed, formally:

$$executable(s) \stackrel{\text{def}}{=} (\forall a, s').do(a, s') \sqsubseteq s \supset Poss(a, s').$$

2.2 Golog and ConGolog

2.2.1 Golog

Golog is a programming language defined in the situation calculus. It allows a user to specify programs whose set of legal executions specifies a sub-tree of the tree of situations of a basic action theory. From a planning point of view, it can be used to provide an effective way of pruning the search by specifying the skeleton of a plan. Golog has an Algol-inspired syntax extended with flexible non-deterministic constructs. Its constructs are shown below.

a	primitive action
$\phi?$	test condition ϕ
$(\delta_1; \delta_2)$	sequence
if ϕ **then** δ_1 **else** δ_2	conditional
while ϕ **do** δ'	loops
$(\delta_1 \vert \delta_2)$	non-deterministic choice
$\pi v.\delta$	non-deterministic choice of argument
δ^*	non-deterministic iteration
$\{P_1(\vec{t}_1, \delta_1); \ldots; P_n(\vec{t}_n, \delta_n); \delta\}$	procedures

The semantics of a Golog program δ is defined in terms of macro expansion into formulae of the situation calculus. $Do(\delta, s, s')$ is understood to denote a formula expressing that executing δ in situation s is possible and may result in a situation s'. This is defined inductively over the program constructs. For instance for a primitive action a: $Do(a, s, s') \stackrel{\text{def}}{=} Poss(a[s], s) \wedge s' = do(a[s], s)$, where $a[s]$ denotes the action a with all its arguments instantiated in situation s, and for non-determinism: $Do(\delta_1|\delta_2, s, s') \stackrel{\text{def}}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$. While deterministic constructs enforce the occurrence of particular actions, non-deterministic constructs define "open parts" that are completed using planning.

In particular, the non-deterministic choice of argument $\pi v.\delta$ introduces a *program variable* v that may occur in δ. In this chapter, we restrict program variables to only appear as action parameters or in the place of objects in conditions. For instance **while** $(\exists b).OnTable(b)$ **do** $\pi v.\ OnTable(v)?; Remove(v)$ could be a program that removes all blocks, one-by-one from a table.

2.2.2 ConGolog

ConGolog adds concurrency to Golog, allowing the following additional constructs:

$(\delta_1 \parallel \delta_2)$	concurrent execution
$(\delta_1 \rangle\!\rangle \delta_2)$	prioritized concurrency
$\delta^{\parallel}$	concurrent iteration

Concurrency is defined as action interleaving. For example, the program $(a \parallel (b; c))$ admits three executions: *abc*, *bac*, and *bca*.

ConGolog introduced a so-called *transition semantics* for programs. The semantics of a program δ is given through two predicates $Trans(\delta, s, \delta', s')$ and $Final(\delta, s)$. The former states that in situation s program δ can perform a step, resulting in a remaining program δ' and new situation s'. The latter states that the program δ can legally terminate in s. De Giacomo *et al.* [2000] provide the complete axioms for the semantics; we show some of them below.

For a primitive action we have

$$Trans(a, s, \delta', s') \equiv Poss(a[s], s) \wedge \delta' = nil \wedge s' = do(a[s], s)$$

and $Final(a, s) \equiv false$. One important role of $Final$ is with sequences:

$$\begin{aligned}
&Trans(\delta_1; \delta_2, s, \delta', s') \equiv \\
&\quad (\exists\gamma).\delta' = (\gamma; \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \\
&\quad \vee Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s').
\end{aligned}$$

For concurrency constructs we have:

$$\begin{aligned}
&Trans(\delta_1 \parallel \delta_2, s, \delta', s') \equiv \\
&\quad (\exists\gamma).\delta' = (\gamma \parallel \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \\
&\quad \vee \delta' = (\delta_1 \parallel \gamma) \wedge Trans(\delta_2, s, \gamma, s') \\
&Trans(\delta_1 \rangle\!\rangle \delta_2, s, \delta', s') \equiv \\
&\quad (\exists\gamma).\delta' = (\gamma \rangle\!\rangle \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee \delta' = (\delta_1 \rangle\!\rangle \gamma) \\
&\quad \wedge Trans(\delta_2, s, \gamma, s') \wedge (\nexists\zeta, s'').Trans(\delta_1, s, \zeta, s'') \\
&Trans(\delta^{\parallel}, s, \delta', s') \equiv \\
&\quad (\exists\gamma).\delta' = (\gamma \parallel \delta^{\parallel}) \wedge Trans(\delta, s, \gamma, s')
\end{aligned}$$

The first two programs are only "final" when both subprograms are, while the third can be terminated at will:

$$\begin{aligned}
&\mathit{Final}(\delta_1 \,\|\, \delta_2, s) \equiv \mathit{Final}(\delta_1, s) \wedge \mathit{Final}(\delta_2, s)\\
&\mathit{Final}(\delta_1 \rangle\!\rangle\, \delta_2, s) \equiv \mathit{Final}(\delta_1, s) \wedge \mathit{Final}(\delta_2, s)\\
&\mathit{Final}(\delta^{\|}, s) \equiv \mathit{true}
\end{aligned}$$

A transition semantics facilitates the interleaving of program interpretation (planning) and execution, and reasoning about sensing actions. The downside of this semantics is its requirement to reify programs: programs are represented as terms, in order to quantify over them. The other shortcoming is the requirement of an additional second-order axiom for defining the transitive closure of *Trans*, denoted *Trans**. This axiom is needed to define a new Do_2 predicate that defines the situations that result from executing a (ConGolog) program:

$$Do_2(\delta, s, s') \overset{\text{def}}{=} (\exists \delta').\, \mathit{Trans}^*(\delta, s, \delta', s') \wedge \mathit{Final}(\delta', s').$$

We refer to the axioms defining the transition semantics as Σ_{ConGolog}. This includes the mentioned second-order axioms and axioms required for reification of programs.

3 Compiling ConGolog into Basic Action Theories

In this section we describe an algorithm for compiling a given ConGolog program and a given basic action theory into a new basic action theory. For readability, we focus our description on the intuitions behind the algorithm. The actual pseudo code of the algorithm can be found in Appendix 6.

Our algorithm accepts as input a basic action theory $\mathcal{D}$ and a ConGolog program $\mathcal{P} = \{P_1(\vec{t_1}, \delta_{P_1}); \ldots; P_n(\vec{t_n}, \delta_{P_n})\}; \delta_{main}$ containing n procedure definitions with formal arguments $\vec{t_i}$ and procedure body δ_{P_i}, and a main program δ_{main}. It outputs a new basic action theory $\mathcal{D}_{\mathcal{P}}$ whose tree of executable situations corresponds to the sub-tree of situations in $\mathcal{D}$ that are executions of $\mathcal{P}$ in $\mathcal{D}$.

The intuition behind our compilation is to model the dynamics of a ConGolog program as a Petri net with an infinite stack, and then represent this Petri net and the stack as a basic action theory in the situation calculus. Roughly, a Petri net is a finite state automaton that can be in more than one state at the same time. To reflect that, in Petri net terminology, states are called *places* and active places are marked by *tokens* which move from place to place using transitions. The total number of tokens can change during execution, for instance to model concurrency. To model the dynamics of ConGolog programs, we use a so-called *colored* Petri net, where tokens have unique identifiers. We do not define the Petri net induced by a program formally, but only use it for illustration. Intuitively, places in the Petri net represent the current position in the execution of the program (i.e., a sort of program counter), while (labeled) transitions specify which actions are legal at each stage during the execution. Each token represents one of possibly several concurrently executing threads. Given a program $\mathcal{P}$, our algorithm generates the axioms required to model the underlying Petri net as a basic action theory. To this end, we create (1) special bookkeeping predicates, to represent the Petri net and the stack, and (2) additional actions, to represent some of the transitions in the Petri net.

It is important to note that our algorithm operates only syntactically on the given inputs. In particular, it does not perform any type of reasoning within the provided basic action theory, which makes it easy to show that our algorithm has modest complexity (see below).

The compilation proceeds in six steps.

Step 1

For each procedure $P_j(t_{j_1}, \ldots, t_{j_{k_j}}, \delta_{P_j})$ in $\mathcal{P}$ we compute

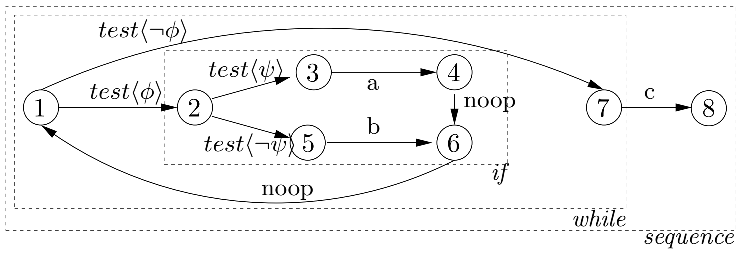
$$(\mathtt{ax}_j, \mathtt{i}_j) = \mathbf{comp}(\delta_{P_j}, \mathtt{0}, \{t_{j_1}, \ldots, t_{j_{k_j}}\}, P_j)$$

where $\{t_{j_1}, \ldots, t_{j_{k_j}}\}$ are the formal parameters of the procedure, and δ_{P_j} is the body of P_j.[1] The function **comp**, defined in Appendix A, takes as input a ConGolog program, an integer used as a program counter, a set of program variables, and a procedure name, used to distinguish different contexts. It outputs a set of sentences $\mathtt{ax}$, and an integer $\mathtt{i}$, intuitively denoting the value of the program counter after the program terminates. The set of sentences is later processed further to generate the axioms of $\mathcal{D}_{\mathcal{P}}$, but before we get to this, we first consider the function **comp** in more detail.

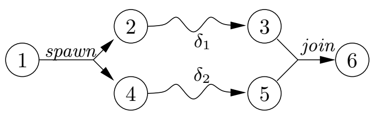
comp is defined recursively over the structure of programs. Starting from an initial place labeled $(0, \mathit{main})$, **comp** incrementally constructs the Petri net, generating new network places as it recurses over the structure of the program. Assume **comp** is currently at a place labeled with (i, p), where i is the program counter and p a procedure name, and that it encounters a primitive action α in the program. Then, it adds a new place to the Petri net labeled with $(i + 1, p)$ and a transition from the current place to this new place, labeled with α. **comp** generates and returns several sentences which will later be included as axioms of $\mathcal{D}_{\mathcal{P}}$. First, it generates a sentence about the preconditions of α. In the described case it generates $\mathit{Poss}(\alpha(\mathit{th}), s) \leftarrow \mathit{Thread}(\mathit{th}, s) \wedge \mathit{state}(\mathit{th}, s) = (i, p)$ which states that we can execute α in thread th if th denotes an active thread and its token is in (i, p). (Note that we give an extra argument to each action, denoting the thread it is being performed in.) It further generates an appropriate effect, stating that when α is performed in (i, p), the token moves to $(i + 1, p)$. The sentence generated in this case is $\mathit{state}(\mathit{th}, \mathit{do}(\alpha(\mathit{th}), s)) = (i + 1, p) \leftarrow \mathit{state}(\mathit{th}, s) = (i, p)$.

Example 1. Consider the program of Figure 1(a), where special *test* actions are used to transition to a sub-net conditioned on a formula, and *noop* allows

[1] For simplicity of presentation we assume that procedures do not contain additional procedure definitions.



(a) Petri net for **while** ϕ **do** (**if** ψ **then** a **else** b); c.



(b) Petri net for $\delta_1 \parallel \delta_2$

Figure 1: Two example Petri nets.

unconditional transitions.[2] To keep the presentation simple, we only show the sentences produced by the algorithm for the transitions from state $1 \rightarrow 2$ and $7 \rightarrow 8$.

For transition $1 \rightarrow 2$, if ϕ does not mention program variables, the algorithm generates the following sentences:

$$\begin{aligned} Poss(test(th, 1, 2, main), s) \leftarrow (&Thread(th, s) \wedge \phi(s) \wedge \\ &state(th, s) = (1, main)), \end{aligned} \tag{4}$$

$$state(th, do(test(th, 1, 2, main), s)) = (2, main). \tag{5}$$

And for the transition $7 \rightarrow 8$ we get:

$$Poss(c(th), s) \leftarrow (Thread(th, s) \wedge state(th, s) = (7, main)), \tag{6}$$

$$state(th, do(c(th), s)) = (8, main) \leftarrow state(th, s) = (7, main). \tag{7}$$

In the remaining steps of the compilation (see below), the successor state axiom for the *state* fluent is formed and precondition axioms are put into normal form. If in $\mathcal{D}$ the precondition axiom for c was $Poss(c, s) \equiv \Pi_c(s)$, then the new precondition axiom in $\mathcal{D}_\mathcal{P}$ is $Poss(c(th), s) \equiv \Pi_c(s) \wedge \varphi$, where φ stands for the right-hand side of Equation 6. §

So far, the Petri net is equivalent to a simple automaton, since we have only been concerned with a single token. This changes when one considers concurrency. Concurrency is modeled using threads, where each thread is represented by an identifiable token in the net. For instance, the basic concurrency construct $\delta_1 \parallel \delta_2$ puts the current token in the initial state of the sub-Petri net recursively generated for δ_1, and creates a new token which it puts into the initial state of

[2]Names used for test actions in this example are simplified for clarity. Refer to the pseudocode for more details.

δ_2. These tokens are joined back together when both programs have finished executing (Figure 1(b)).

The greatest challenges we faced while devising **comp**, were caused by the interaction of various advanced programming constructs, in particular program variables, procedures, and iterative concurrency. We elaborate briefly on some of these challenges.

Procedure calls are realized using two new actions *call* and *return*. The former moves the token of the current thread to the initial place of the called procedure, while *return* returns it to the next state of the current program, once the token has reached the final state of that procedure. Since the compilation of the procedures themselves needs to be independent from the context from which they are called, we do not know the return state during compile time, but need to store it during run-time instead. Since procedures can be recursive, we require a stack, containing all (recursive) return states. The stack is realized using two functional fluents $\mathit{stack}(\mathit{th}, v, s)$ and $\mathit{sp}(\mathit{th}, s)$, where the former denotes the content of the stack entries, and the latter is a stack-pointer, always pointing to the next free position on top of the stack.

Concurrency is realized by using explicit thread names. Each action is given an additional parameter *th*, denoting the thread it is executed in. This is necessary since there may be situations where two threads intend to execute the same action next. Once that action executes, we need the thread name to disambiguate which thread actually proceeded. Thread names are also required for other purposes, like program variables, described below. The active threads are denoted by the relative fluent $\mathit{Thread}(\mathit{th}, s)$, and initially only one thread, [0], is active. A new thread is created by the *spawn* action, which also sets up some new data structures (fluents) for the new thread, for instance its own procedure call stack. Two threads are joined back by the action *join*.

For thread names, we use lists of numbers. The main thread is [0], and its direct children are called $[N, 0]$ where N is the number of the child. The k-th child of the n-th child of the main thread is called $[k, n, 0]$. This is more complicated than increasing a single thread counter, which would have been an alternative, but has the advantage that thread names can be reused after threads terminate. With numbers, for instance, an infinitely running program with concurrency would require infinite numbers. This would also more severely limit the ability to compile into PDDL.

Prioritized concurrency is governed by a new fluent $\mathit{Prio}(\mathit{th}_1, \mathit{th}_2, s)$ which indicates that thread th_1 has priority over thread th_2. A thread can only proceed when no prioritized thread can perform an action.

Program variables as created by π constructs, are realized using the functional fluent $\mathit{map}(x, s)$, to denote their value. The parameter x is a tuple (th, y, v) where *th* denotes the thread this variable was created in, y the stack position, and v is the name as mentioned in the program (e.g., $\pi\, v.\delta$). Thread names are required to disambiguate in cases like $(\pi\, v.\delta)^{\|}$ where in each thread a new variable of the same name is created. Similarly stack positions are required when program variables are used in recursive procedures.

To compile the main procedure we call

$$(\mathtt{ax}_{main}, \mathtt{i}_{main}) = \mathbf{comp}(\delta_{main}, \mathtt{0}, \emptyset, \mathit{main}),$$

which yields the final program counter $\mathtt{i}_{main}$, which corresponds to a particular "final" place of the Petri net. This will be used as a goal: if there is a token in $(\mathtt{i}_{main}, \mathit{main})$, the program has executed successfully. This roughly corresponds to the *Final* predicate in ConGolog.

Step 2

Thus far we have generated program-specific sentences, describing the dynamics of the Petri net. There is also a number of program-independent sentences that we require, which intuitively state the default dynamics of the involved bookkeeping actions (see Appendix B for details). We denote these as $\mathtt{ax}_{\text{common}}$ and define the set $\mathtt{AX}$ as $\mathtt{ax}_{main} \cup \bigcup_j \mathtt{ax}_j \cup \mathtt{ax}_{\text{common}}$.

The remaining steps of the compilation aggregate the sentences in $\mathtt{AX}$ to produce $\mathcal{D}_{\mathcal{P}}$, producing all the precondition axioms, successor state axioms, initial state axioms, and unique names axioms.

Step 3

Recall that procedure calls require two new actions *call* and *return*. The effect axioms for both are domain independent and thus in $\mathtt{ax}_{\text{common}}$, and the precondition axioms for *call* are generated by **comp**. In Step 3 we need to create the precondition axioms for *return*, which is possible in all final states, i.e., for each procedure P_j compiled in Step 1, we enable *return* when $\mathit{state}(\mathit{th}, s) = (\mathtt{i}_j, P_j)$.

Step 4

For each place of each Petri net, all conditions under which any action can execute in this place and context are recorded. We generate axioms for a new fluent $\mathit{CanTrans}(\mathit{th}, s)$, which indicates whether in situation s a given thread *th* can perform an action. This definition is only required in conjunction with concurrency, and can be skipped if this language feature is not used.

Step 5

For each primitive action α (including bookkeeping actions), Step 5 removes all sentences $\mathit{Poss}(\alpha, s) \leftarrow \phi$ from $\mathtt{AX}$ and combines them into a new precondition axiom for α, by:

1. disjoining all ϕ's,
2. conjoining the resulting formula with any preexisting preconditions for α, and
3. conjoining the result with the following additional condition that governs priority among threads and allows forced execution of a selected thread:

   $(\nexists t). \mathit{Thread}(t) \wedge t \neq \mathit{th} \wedge (\mathit{Forced}(t) \wedge \neg \mathit{Parent}(t, \mathit{th}) \vee \mathit{Prio}(t, \mathit{th}) \wedge \mathit{CanTrans}(t))$

   where the new relational fluent $\mathit{Parent}(\mathit{th}_1, \mathit{th}_2)$ expresses that thread th_2 was spawn from thread th_1, directly or indirectly.

The latter is used to enable prioritized concurrency, explicitly prohibiting threads from executing for which there is a thread with higher priority that can execute its next action. This condition is also used to ensure so-called *synchronized* while's and if's. Roughly, the latter means that testing the conditions of these constructs is not a transition by itself, but needs to be immediately followed by a transition on its body, or otherwise one needs to backtrack to a place before the test.

Step 6

Since all the *Poss* sentences have been removed, `AX` now only contains sentences describing effects of actions. On these, Step 6 applies Reiter's solution to the frame problem, to produce successor state axioms (see Section 2.1.2).

The result is a set of precondition and successor state axioms, describing the dynamics of all procedures' Petri nets. We also add the axiom $\mathit{state}([0], S_0) = (0, \mathit{main})$, stating that initially the main thread, denoted [0], is in the initial place of the Petri net of the *main* procedure.

While our compilation makes several second-order axioms, specific to ConGolog's transition semantics, unnecessary, it does require second-order to define natural numbers and lists. The former is used to address the elements of the stack, the later to give names to threads. We assume standard definitions for these. These can be avoided when both recursion, and the number of concurrent threads is bound by a constant. This restriction is also required for further compilation to PDDL (see below).

Let $\mathcal{D}_\mathcal{P}$ be the new basic action theory resulting from compiling $\mathcal{P}$ into the given action theory $\mathcal{D}$. We can show the following theorems which state that the compilation is both correct and succinct. All the proofs can be found in Appendix 6.

Theorem 1. Let S' be any ground situation term of $\mathcal{D}$. Then there is a ground situation term $S'_\mathcal{P}$ in $\mathcal{D}_\mathcal{P}$ such that $S' = \mathit{filter}(S'_\mathcal{P}, \mathcal{D})$ and

$$\mathcal{D}_\mathcal{P} \models \mathit{executable}(S'_\mathcal{P}) \wedge \mathit{state}([0], S'_\mathcal{P}) = (\mathtt{i}_{\mathit{main}}, \mathit{main})$$

iff $\mathcal{D} \models \mathit{Do}_2(\mathcal{P}, S_0, S')$.

Here *filter*$(S'_{\mathcal{P}}, \mathcal{D})$ is a function that removes from the situation term $S'_{\mathcal{P}}$ any actions not defined in $\mathcal{D}$, and also removes the additional thread argument from the remaining actions. This removes all bookkeeping actions from $S'_{\mathcal{P}}$, in order to compare the sequence of contained domain actions with S'.

For the next theorem we define the size of a program as the number of program constructs it contains plus the number of logical connectives mentioned in conditions. Similarly, the size of an axiom is measured by the number of logical connectives it contains.

Theorem 2. If the size of $\mathcal{P}$ is n and $\mathcal{D}$ contains m axioms each of size $\leq k$, then $\mathcal{D}_{\mathcal{P}}$ contains $O(n) + m$ axioms each of size $O(k + n)$.

Theorem 3. If the size of $\mathcal{P}$ is n, then the time required to compute the compilation is $O(n^2)$.

Intuitively, recursive procedure calls, while–loops, concurrency and other seemingly problematic constructs do not incur a significant increase in the size of the output, because of the syntactic nature of the compilation and the careful use of bookkeeping fluents and actions to model the desired behaviors. Similarly, the requirement for second-order logic to define loops is cast into the induction axiom included in the foundational axioms of the situation calculus, through the use of bookkeeping fluents and actions.

4 Analysis

4.1 Theoretical Merits

To prove properties of a ConGolog program $\mathcal{P}$, we now have two alternatives. We can reason using the original transition semantics of ConGolog, represented as a fixed set of axioms Σ_{ConGolog}, or we can use the new basic action theory $\mathcal{D}_{\mathcal{P}}$ resulting from applying our compilation, extended with natural numbers and lists. At first glance, using Σ_{ConGolog} may look simpler since the axioms in Σ_{ConGolog} are independent of the program. However, we argue that reasoning itself is actually simplified when using $\mathcal{D}_{\mathcal{P}}$.

One advantage of $\mathcal{D}_{\mathcal{P}}$ is that it defines the dynamics of a program in terms of fluents. For example, any executable situation S for which $\mathcal{D}_{\mathcal{P}} \models \mathit{state}([0], S) = (\mathtt{i}_{main}, \mathit{main})$, with $\mathtt{i}_{main}$ as defined above, is a legal execution of the program. Regressing the condition $\mathit{state}([0], S) = (\mathtt{i}_{main}, \mathit{main})$ over the actions comprising S, together with all involved action preconditions, results in a formula over the initial situation S_0. Following Theorem 1 and Reiter's Regression Theorem, this formula is equivalent to the question of whether the actions comprising S are a legal execution of the program. More generally, using regression we can determine sufficient conditions under which a given sequence of actions (whose parameters do not need to be ground) will satisfy a given formula while executing the program. These queries could not be answered using regression in the transition semantics since neither the semantics of Golog nor ConGolog were in terms of regressable formulae[3].

Another advantage of reasoning in $\mathcal{D}_{\mathcal{P}}$ is that the compilation eliminates the need for ConGolog's tedious (second-order) reification of programs, as well as the second-order axioms found in $\Sigma_{\mathrm{ConGolog}}$ for defining the *Trans* and the *Trans** predicates. As such, proving properties of programs in $\mathcal{D}_{\mathcal{P}}$ is not much different from proving properties in the standard situation calculus. In some cases (e.g., when proving a property of a particular execution trace) we can apply regression. In more general cases (e.g., when proving invariants), we can simply use induction over situations [Reiter, 1993]. In fact, we have proved properties of simple Golog programs by representing $\mathcal{D}_{\mathcal{P}}$ in the higher-order theorem prover PVS [Owre *et al.*, 1992]. In PVS, situations, natural numbers, and lists, can be easily defined as recursive data-types. We found the lack of reification in $\mathcal{D}_{\mathcal{P}}$ together with the limited number of second-order axioms made theorem proving less laborious and more intuitive than previous attempts to prove properties of Golog programs in PVS [Shapiro *et al.*, 2002].

In our translated domain it is particularly simple to prove a property about a specific point during the program's execution. The main reason for this is that in our compiled theories we can refer to points in the program's execution by referring to the states of the Petri net that represent those points. For example, proving a property about the situations that result from executing the program to termination reduces to proving that a certain formula is true for every situation in which we are at the Petri net place that corresponds to the end of the program. When proving these types of properties using the second-order axioms of the original ConGolog semantics, as was done by [Shapiro *et al.*, 2002], one is forced to effectively simulate an execution of the program by incrementally evaluating the transitive closure of the *Trans* predicate. On the other hand, in case we want to prove a property that holds during the whole execution of a program using our compiled theory, we have to resort to induction over situations. The course of the proof in this case is very similar to the one that would be obtained in the framework of [Shapiro *et al.*, 2002].

To demonstrate the feasibility of proving properties of programs using automated theorem provers, we modeled one of the Golog example programs in the blocks world used by [Liu, 2002]. This program consists of a while loop that non-deterministically moves blocks until there is only one block on the table. The task is to prove that there is a single tower in the final situation. This could be proved automatically by PVS in fractions of a second. [Liu, 2002] also obtained a very simple proof but appealing to a Hoare-style proof system on top of ConGolog's semantics.

4.2 Practical Merits

ConGolog to PDDL

A practical consequence of the compilation is the possibility of further com-

[3][Reiter, 2001, p. 62] defines regressable formulae.

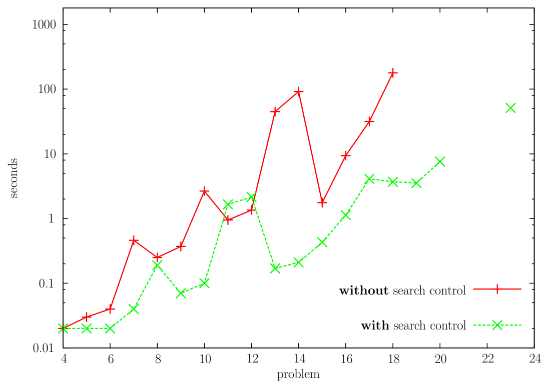


Figure 2: Run-time comparison of a heuristic search based planner solving instances of the `storage` domain of the International Planning Competition, with and without Golog search controlled compiled into the PDDL domain definition [Baier *et al.*, 2007].

piling the resulting action theory into other action languages, like PDDL. The advantage of this approach is the possibility of using the fastest state-of-the-art planners to accomplish the planning needed while interpreting ConGolog programs. This is not only of interest to the agent programming community but also for the planning community, since ConGolog can be used to express domain control knowledge.

In previous work we have shown that it is possible to compile Golog programs without procedures into PDDL [Baier *et al.*, 2007], and shown that Golog domain control knowledge can speed up search of standard planning benchmarks. Figure 2 shows an example of the obtainable speed-up for the `storage` domain of the International Planning Competition.

In the compilation proposed in this paper we are considering the richer variant ConGolog, which allows programs with various forms of concurrency, and we also enabled the use of possibly recursive procedures. Unfortunately, these additions all together cannot be compiled directly into current versions of PDDL. The main reason is that PDDL does not provide the functionality for defining unbounded data structures, which we need, for example, for representing the stack for procedure calls.

Recent versions of PDDL support natural numbers, but these cannot be used as arguments to predicates, since numbers are not considered objects of the domain. The pragmatic reason for this restriction is to avoid the possibility of infinite branching factors [Fox and Long, 2003, p. 68] since actions could take numerical arguments. Since our compilation does not introduce infinite branching factors, we believe that PDDL could be extended accordingly to allow the full expressiveness of ConGolog and HTNs. We hope that our work may convince the planning community that such an extension would lead to a significant increase in the expressiveness of PDDL.

It is still possible to translate ConGolog into PDDL if we are willing to either disallow recursion and iterative concurrency or limit the depth of recursion and the number of concurrently executing threads. The second option is probably the most interesting one, since in practical applications in which finite plans are needed, we will not require the power of infinite recursion. The main challenge in this case, is to generate a theory in which the stack and the lists which are used to represent thread names are bounded. The following are the main aspects that are needed to translate to PDDL.

1. All fluents that represent counters (e.g., the stack pointer fluent) are now represented by *relational* fluents, an argument of which corresponds to the value of the counter. The value of the counter is represented by a PDDL *object*. We generate finitely many objects for counters, and a static predicate to indicate the successor for each counter object.

2. All other functional fluents (e.g., *map* and *state*) are represented in PDDL as relational fluents. In particular, the relational fluent for *state* contains one argument for each element of the $(\mathtt{i}, \mathtt{c})$ pair.

3. Threads, which in our basic action theories are represented as lists, and which are employed as arguments to actions are represented in PDDL as (bounded) lists of size equal to a parameter k. Moreover, actions, instead of having a single thread parameter, are now represented as having k additional parameters, where the i-th parameter of the action corresponds to the i-th parameter of the thread list. We emulate lists with fewer than k elements by using a special constant *nao* (not-an-object) to represent a position of the list that is not occupied by any object. Finally, effects of the actions *spawn*, and *join*, which modify the current thread, can be straightforwardly modified to use this new representation.

4. The precondition of the *call* and *spawn* actions are modified such that they will not be possible if the capacity of the stack/thread list is already at its maximum.

Our PDDL translation is defined for ConGolog programs, that are assumed to operate over a preexisting PDDL domain and problem specification. Thus, we assume that, instead of receiving a basic action theory as input, the algorithm receives a PDDL domain and problem definition describing preconditions and effects of actions, and the initial and goal state of the planning problem. The steps of the compilation procedure that integrate the basic action theory with the output of the program compilation are trivially modified for the PDDL case. Thus, new bookkeeping actions are added, and existing domain actions receive additional parameters, preconditions and effects as necessary. More details on the general setup of this compilation can be found in [Baier *et al.*, 2007].

HTN to PDDL

Hierarchical Task Networks (HTNs) [Erol *et al.*, 1994; Ghallab *et al.*, 2004] are a popular planning formalism used to provide domain control knowledge to a planner by representing planning solutions in a hierarchical fashion. They have broad applications, including classical planning [Nau *et al.*, 2003] and web service composition [Kuter *et al.*, 2004]. The HTN formalism has been in a sense divorced from classical planning since state-of-the-art planners do not handle HTNs. Our approach enables the compilation of HTNs into basic action theories and – when bounding recursion – to PDDL. Compiling HTNs to PDDL is beneficial, as it provides the means of combining their expressiveness with modern planning techniques.

Several HTN variants have been proposed in the literature, and one particular one has been previously translated to ConGolog [Gabaldon, 2002]. Here we consider the HTN formalism described by [Ghallab *et al.*, 2004], using a compelling subset of the language for constraints allowed by the SHOP2 planner [Nau *et al.*, 2003], which obtained a second place in the 2002 International Planning Competition. The translation of this flavor of HTN to ConGolog is almost trivial.

In the variant of HTN planning that we consider, we distinguish three entities, which are specified by the user: *tasks, operators*, and *methods.* Tasks represent parametrized activities to perform. They can be *primitive* or *compound.* Primitive tasks are realized by operators, actions that can be physically executed in the domain. Compound tasks need to be decomposed using one of possibly several applicable methods. A method m is of the form `(:method` $\mathtt{\textit{head}}(m)$ $p_1(\vec{v})$ $t_1(\vec{v}) \ldots p_n(\vec{v})$ $t_n(\vec{v})$`)` where the head specifies the task with formal arguments $\vec{v}$ to which this method is applicable, $p_i(\vec{v})$ are preconditions and each $t_i(\vec{v})$ is a list of sub-tasks. As in SHOP2, we give an if–then–else semantics to methods: if $p_1(\vec{v})$ holds, then the task is decomposed into the sub-tasks $t_1(\vec{v})$. Otherwise, $p_2(\vec{v})$ is tested and so on. For a method to be applicable to a given task instance, the task's actual parameters have to unify with the method's formal parameters, and at least one p_i has to be satisfied. Each list of sub-tasks $t_i(\vec{v})$ can be a nesting of `:ordered` and `:unordered` lists, stating restrictions on the order in which these tasks can be carried out.

A detailed description of the formal algorithm for translating these HTNs to ConGolog is beyond the scope of this chapter, but roughly the construction proceeds as follows: For each method m, we create a new procedure

$$m\big(\vec{v}, \textbf{ if } p_1(\vec{v}) \textbf{ then } \delta_1 \textbf{ else if} \ldots \textbf{else } (p_n(\vec{v})?; \delta_n)\big)$$

where δ_i is the following program representing sub-task t_i: Recursively, if t_i is an `:ordered` set of tasks, then δ_i is simply the sequence of these tasks. Otherwise, if t_i is an `:unordered` set, then δ_i is the concurrent execution of all of these. For instance, `(:unordered` a `(:ordered` b_1 b_2`)` `(:ordered` c_1 c_2 c_2`))`, would be translated to: $(a \parallel (b_1; b_2) \parallel (c_1; c_2; c_3))$. Since there may be more than one method applicable to a given task, we translate each task into a non-deterministic choice over all of its applicable procedures: $(m_1|m_2|\ldots|m_n)$.

An HTN represented in such a way as a ConGolog program can thus be compiled into a basic action theory just as easily, and by limiting the recursion depth of methods, we can again compile the resulting theory further into PDDL.

Execution Monitoring

A third practical merit of our compilation is that it allows us to lift existing execution monitoring techniques used in planning for monitoring the execution of ConGolog programs.

Monitoring the execution of a plan amounts to tracking the state of the world, recognizing discrepancies between the expected state of the world according to the model assumptions made during planning and the actual state of the world, and determining whether a recognized discrepancy warrants plan modification. One promising strategy is to annotate the plan in each step with a sufficient and necessary condition for its validity with respect to reaching the goal. Implicitly or explicitly, many execution monitoring approaches in the literature apply this technique and derive these conditions by regressing the goal over the remaining actions of the plan. We have shown that this can be generalized to the case where not only the validity of a plan, but also its optimality must be monitored [Fritz and McIlraith, 2007].

When an agent is controlled by a Golog or ConGolog program, we need to monitor more than just the stated final state goal. Also, the constraints on the agent's course of action imposed by the program must be satisfied. Those tasks can be accomplished using regression on our compiled theory. Hence, we can adapt existing regression-based techniques to the problem of monitoring the execution of ConGolog programs without any extra machinery.

5 Related Work

There are several pieces of related work. In previous work we provided a compilation of Golog programs without procedures into PDDL [Baier *et al.*, 2007], showing that notable speed-ups can be obtained in planning benchmarks. Our current work significantly extends the aforementioned compilation by showing how ConGolog programs (with procedures and extended with useful features like concurrency) can also be translated into classical planning, under certain restrictions. While our previous work exploited automata in the translation, the added expressivity of ConGolog necessitated the use of Petri nets.

Funge [1998] provided a compilation of Golog programs into Prolog, to make program interpretation more efficient. His approach is similar to ours in the sense that the output can be viewed as representing a finite-state automaton. However, the output is not a logical theory, the approach cannot handle concurrency, and there are no immediate applications like planning.

There is also related work on the compilation of HTNs into ConGolog and PDDL. As previously noted, Gabaldon [2002] presented a means of translating the general HTN formalism of Erol *et al.* [1994] into ConGolog. We showed how the HTN formalism [Ghallab *et al.*, 2004] with the popular SHOP2 [Nau *et al.*, 2003] language for constraints could be translated into ConGolog and in turn compiled into PDDL. We limited ourselves to SHOP2 constraints because of its practical interest; this syntax also eliminated the need for additional predicates. Nevertheless, we could have just as easily used Gabaldon's more involved translation to ConGolog to compile general HTNs with bounded recursion into PDDL. Of further note, recently Lekavý and Návrat [2007] provided a linear translation of a restricted acyclic subset of HTN into STRIPS. Their translation generates a Turing machine with a finite tape represented in STRIPS.

Finally, there is related work on proving properties of Golog/ConGolog programs. Shapiro *et al.* [2002] used PVS to prove properties of ConGolog programs appealing to a direct representation of the $Trans^*$ second-order axiom, and by reifying programs. As a result it is possible to use induction to prove properties that hold during the execution of programs, but it is not straightforward to prove properties that hold at particular points in the execution (e.g., at the end of the program). As we have demonstrated, in our case proving any of these properties is done as with any property of the situation calculus. Also of note, Liu [2002] introduced a Hoare-style proof system for proving properties of Golog programs (without concurrency). The motivation for this approach was similar to ours: to minimize second-order reasoning. As a consequence, proving properties is facilitated in this formalism, too. Recently, Claßen and Lakemeyer [2008] proposed an interesting algorithm for proving properties of non-terminating Golog programs expressed in a logic that resembles CTL*. To prove such properties, they construct a *characteristic graph*, which resembles our Petri nets. With our compiled domains and by using known translations of LTL into planning goals (e.g., [Baier and McIlraith, 2006]) we could prove similar properties, but restricted to only finite executions.

6 Discussion

We proposed an algorithm for compiling arbitrary ConGolog programs into basic action theories in the situation calculus. The size of the resulting theory is quadratic in the size of the compiled program, and contains a simpler set of axioms, in the sense that it avoids the need for program reification and reduces the number of second-order axioms. The compilation presents a significant contribution for at least two reasons. First, it provides the mathematical foundations for compiling powerful ConGolog and HTN search control into basic action theories of the situation calculus. These can in turn be translated into other action formalisms including, with minor restrictions, PDDL. Such a translation enables most state-of-the-art planners to exploit powerful domain control knowledge without the need to construct special-purpose machinery within their planner. Second, in eliminating the need for reification, the translated theory facilitates automated proof of program properties in systems such as PVS as well as, in some cases, enabling properties to be proved by regression of ConGolog programs followed by (first-order) theorem proving in the initial situation.

References

[Baier and McIlraith, 2006] Jorge A. Baier and Sheila A. McIlraith. Planning with first-order temporally extended goals using heuristic search. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI)*, pages 788–795, Boston, MA, 2006.

[Baier *et al.*, 2007] Jorge A. Baier, Christian Fritz, and Sheila A. McIlraith. Exploiting procedural domain control knowledge in state-of-the-art planners. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS)*, pages 26–33, Providence, Rhode Island, USA, September 22 - 26 2007.

[Burgard *et al.*, 1999] Wolfram Burgard, Armin B. Cremers, Dieter Fox, Dirk Hähnel, Gerhard Lakemeyer, Dirk Schulz, Walter Steiner, and Sebastian Thrun. Experiences with an interactive museum tour-guide robot. *Artificial Intelligence*, 114(1-2):3–55, 1999.

[Claßen and Lakemeyer, 2008] Jens Claßen and Gerhard Lakemeyer. A logic for non-terminating Golog programs. In *Proceedings of the 11th International Conference on Knowledge Representation and Reasoning (KR)*, Sydney, Australia, 2008.

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

[Erol *et al.*, 1994] Kutluhan Erol, James A. Hendler, and Dana S. Nau. HTN planning: Complexity and expressivity. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI)*, pages 1123–1128, 1994.

[Ferrein *et al.*, 2004] A. Ferrein, C. Fritz, and G. Lakemeyer. On-line decision-theoretic Golog for unpredictable domains. In *Proceedings of 27th German Conference on AI (KI)*, pages 322–336, Ulm, Germany, September 20-24 2004. Also appeared in Proceedings of The 4th International Cognitive Robotics Workshop, at ECAI-2004, Valencia, Spain.

[Fox and Long, 2003] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[Fritz and McIlraith, 2007] Christian Fritz and Sheila A. McIlraith. Monitoring plan optimality during execution. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS)*, pages 144–151, Providence, Rhode Island, USA, September 22 - 26 2007.

[Funge, 1998] John Funge. *Making Them Behave: Cognitive Models for Computer Animation.* PhD thesis, University of Toronto, Toronto, Canada, 1998.

[Gabaldon, 2002] Alfredo Gabaldon. Programming hierarchical task networks in the situation calculus. In *AIPS'02 Workshop on On-line Planning and Scheduling*, Toulouse, France, April 2002.

[Ghallab *et al.*, 2004] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[Kuter *et al.*, 2004] Ugur Kuter, Evren Sirin, Dana S. Nau, Bijan Parsia, and James A. Hendler. Information gathering during planning for web service composition. In *Proceedings of the 3rd International Semantic Web Conferece (ISWC)*, pages 335–349, 2004.

[Lekavý and Návrat, 2007] Marián Lekavý and Pavol Návrat. Expressivity of STRIPS-like and HTN-like planning. In *Proceedings of Agent and Multi-Agent Systems: Technologies and Applications, First KES International Symposium (KES-AMSTA)*, pages 121–130, Wroclaw, Poland, May 31–June 1 2007.

[Levesque *et al.*, 1997] Hector J. Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[Levesque *et al.*, 1998] H. Levesque, F. Pirri, and R. Reiter. Foundations for the situation calculus. *Linkoping Electronic Articles in Computer and Information Science*, 3(018), 1998.

[Liu, 2002] Yongmei Liu. A hoare-style proof system for robot programs. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI)*, pages 74–79, 2002.

[McCarthy, 1963] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963.

[McDermott, 1998] Drew V. McDermott. PDDL — The Planning Domain Definition Language. Technical Report TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[McIlraith and Son, 2002] S. McIlraith and T. Son. Adapting golog for composition of semantic web services. In *Proceedings of the 8th International Conference on Knowledge Representation and Reasoning (KR)*, pages 482–493, Toulouse, France, April 22-25 2002.

[Nau *et al.*, 2003] Dana S. Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.

[Owre *et al.*, 1992] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction (CADE)*, pages 748–752, Saratoga Springs, NY, 1992.

[Reiter, 1991] Ray Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, San Diego, CA, 1991.

[Reiter, 1993] Raymond Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, 64(2):337–351, 1993.

[Reiter, 2001] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems.* MIT Press, Cambridge, MA, 2001.

[Shapiro *et al.*, 2002] Steven Shapiro, Yves Lespérance, and Hector J. Levesque. The cognitive agents specification language and verification environment for multiagent systems. In *Proceedings of the 1st International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, pages 19–26, Bologna, Italy, 2002.

A Definition of comp

We here provide the pseudo-code for the **comp** function of Step 1 of our compilation. It takes four inputs: a program δ, an integer `i` used as a program counter, a set `e` of program variables (introduced using the π-construct, see below), and the name of the procedure this program belongs to, `c`. It outputs a set of first-order sentences and a new integer. We will further process the sentences in the subsequent steps of the compilation, eventually producing the axioms of the new theory. The integer represents the value of the program counter at the end of the program. The definition of **comp** is given on pages 25 and 26. The following glossary is to provide some intuition about the used bookkeeping fluents and actions. For parsimony we omit situation arguments:
Fluents:

Thread(th)*:* Thread th exists/is active.

$state(th) = y$*:* Thread th is in state y, where $y = (\mathtt{i}, \mathtt{c})$ for some integer `i` (program counter), and some procedure name `c` (“context”).

$stack(th, p) = y$*:* A stack for storing procedure call return addresses: p is the stack positions, and y the content.

$sp(th) = y$*:* The stack pointer for thread th, pointing to the top of the stack.

$map(th, p, v) = y$*:* The program variable v has value y in thread th on stack position p. Note that thread names and stack positions are required, since the same program variable name (v) may be used simultaneous in different threads, and in recursive procedures.

$childp(th) = y$: For modeling concurrency: the number of children thread *th* has.

$Parent(th_1, th_2)$: Thread th_1 is an ancestor of thread th_2.

$Forced(th)$: Thread *th* and its descendants, have exclusive execution rights.

$Prio(th_1, th_2)$: Thread th_1 has priority over thread th_2. Note that it may still temporarily be the case that $Forced(th_2)$ is true, in which case th_2 is still forced to execute before th_1.

$final(th)$: Thread *th* has executed completely.

$blockeds(th, p, \mathtt{i}, \mathtt{c})$: Thread *th* on stack position p may not move to state $(\mathtt{i}, \mathtt{c})$, for instance, because that branch has been tried before and a backtrack action (see below) has been executed.

$blocked(th, p, \mathtt{i}, \mathtt{c}, x)$: Similarly, a π construct that transitions into $(\mathtt{i}, \mathtt{c})$ in thread *th* on stack position p may not chose value x.

$backtp = y$: A pointer to positions in a backtracking stack. The stack itself contains so-called "shadowed" versions of all relevant bookkeeping fluents X, named s_X, see below.

Actions:

$test(th, \mathtt{i}, \mathtt{c}, \mathtt{i}')$: Under a certain condition, compiled into the preconditions of this action, thread *th* may move from state $\mathtt{i}$ to $\mathtt{i}'$, in context $\mathtt{c}$.

$rtest(th, \mathtt{i}, \mathtt{c}, \mathtt{i}')$: The same as above, but corresponding to an actual transition in the program. Used for the ϕ? constructs only.

$noop(th, \mathtt{i}, \mathtt{c}, \mathtt{i}')$: Just like *test*, but unconditional.

$rnoop(th, \mathtt{i}, \mathtt{c}, \mathtt{i}')$: An unconditional transition, just like the previous, but only for cases where no backtracking information needs to be recorded.

$spawn(th, \mathtt{c}, \mathtt{i}', \mathtt{i}_1, \mathtt{i}_2)$: Creates two new threads (tokens in the Petri net) and sets the first thread to state $\mathtt{i}_1$, the second to $\mathtt{i}_2$, and the current thread to $\mathtt{i}'$.

$join(th, \mathtt{c}, \mathtt{i})$: Joins the child threads back into their parent.

$\pi(th, v, x, \mathtt{c}, \mathtt{i})$: Chose object x for program variable v.

$call(th, p, \mathtt{c}, \mathtt{i})$: Call procedure p.

$return(th)$: Return from a procedure call: look up return address on the stack, and move to the stated program position.

$backtrack(th)$: Backtrack to the last backtracking point.

Function $\texttt{comp}(\delta, \texttt{i}, \texttt{e}, \texttt{c})$ – Part 1 of 2

```
Output: a tuple $(\texttt{ax}, \texttt{i}')$ with $\texttt{ax}$ a set of sentences, $\texttt{i}'$ an integer
switch $\delta$ do
    case $nil$
        return $(\emptyset, \texttt{i})$
    case $A(t_1, \ldots, t_n)$ (where $A(x_1, \ldots, x_n)$ is an action)
        $\texttt{ax} = \{Poss(A(th, x_1, \ldots, x_n), s) \leftarrow Thread(th, s) \wedge state(th, s) = (\texttt{i}, \texttt{c}) \wedge$
                $\bigwedge_{\texttt{j s.t. } t_\texttt{j} \notin \texttt{e}} x_\texttt{j} = t_\texttt{j} \wedge \bigwedge_{\texttt{j s.t. } t_\texttt{j} \in \texttt{e}} map(th, sp(th, s), t_\texttt{j}, s) = x_\texttt{i},$
            $state(th, do(A(th, \vec{x}), s)) = (\texttt{i}{+}1, \texttt{c}) \leftarrow state(th) = (\texttt{i}, \texttt{c})\};$
        return $(\texttt{ax}, \texttt{i}{+}1)$
    case $(\phi?)$
        return $(\textbf{rtest}(\phi, \texttt{i}, \texttt{i}{+}1, \texttt{e}, \texttt{c}), \texttt{i}{+}1)$
    case $(\delta_1; \delta_2)$
        $(\texttt{ax}_1, \texttt{i}_1) = \textbf{comp}(\delta_1, \texttt{i}, \texttt{e}, \texttt{c});$
        $(\texttt{ax}_2, \texttt{i}_2) = \textbf{comp}(\delta_2, \texttt{i}_1, \texttt{e}, \texttt{c});$
        return $(\texttt{ax}_1 \cup \texttt{ax}_2, \texttt{i}_2)$
    case $(\delta_1 | \delta_2)$
        $(\texttt{ax}_1, \texttt{i}_1) = \textbf{comp}(\delta_1, \texttt{i}{+}1, \texttt{e}, \texttt{c});$
        $(\texttt{ax}_2, \texttt{i}_2) = \textbf{comp}(\delta_2, \texttt{i}_1{+}1, \texttt{e}, \texttt{c});$
        $\texttt{ax} =$
        $\{\ \textbf{noop}(\texttt{i}, \texttt{i}{+}1, \texttt{c}), \textbf{noop}(\texttt{i}, \texttt{i}_1{+}1, \texttt{c}), \textbf{noop}(\texttt{i}_1, \texttt{i}_2{+}1, \texttt{c}), \textbf{noop}(\texttt{i}_2, \texttt{i}_2{+}1, \texttt{c})\};$
        return $(\texttt{ax} \cup \texttt{ax}_1 \cup \texttt{ax}_2, \texttt{i}_2{+}1)$
    case (if $\phi$ then $\delta_1$ else $\delta_2$)
        $(\texttt{ax}_1, \texttt{i}_1) = \textbf{comp}(\delta_1, \texttt{i}{+}1, \texttt{e}, \texttt{c});$
        $(\texttt{ax}_2, \texttt{i}_2) = \textbf{comp}(\delta_2, \texttt{i}_1{+}1, \texttt{e}, \texttt{c});$
        $\texttt{ax} = \{\textbf{test}(\phi, \texttt{i}, \texttt{i}{+}1, \texttt{e}, \texttt{c}), \textbf{test}(\neg\phi, \texttt{i}, \texttt{i}_1{+}1, \texttt{e}, \texttt{c}), \textbf{noop}(\texttt{i}_1, \texttt{i}_2, \texttt{c})\};$
        return $(\texttt{ax}_1 \cup \texttt{ax}_2 \cup \texttt{ax}, \texttt{i}_2)$
    case (while $\phi$ do $\delta'$)
        $(\texttt{ax}, \texttt{i}_1) = \textbf{comp}(\delta', \texttt{i}{+}2, \texttt{e}, \texttt{c});$
        return $(\{\textbf{test}(\neg\phi \vee blockeds(th, sp(th), \texttt{i}{+}2, \texttt{c}), \texttt{i}{+}1, \texttt{i}_1{+}1, \texttt{e}, \texttt{c}),$
                $\textbf{rnoop}(\texttt{i}, \texttt{i}{+}1, \texttt{c}), \textbf{test}(\phi, \texttt{i}{+}1, \texttt{i}{+}2, \texttt{e}, \texttt{c}), \textbf{rnoop}(\texttt{i}_1, \texttt{i}{+}1, \texttt{c})\} \cup$
        $\texttt{ax}, \texttt{i}_1{+}1)$
    case $(\delta'^*)$
        $(\texttt{ax}, \texttt{i}_1) = \textbf{comp}(\delta', \texttt{i}{+}1, \texttt{e}, \texttt{c});$
        return
        $(\texttt{ax} \cup \{\textbf{noop}(\texttt{i}, \texttt{i}{+}1, \texttt{c}), \textbf{noop}(\texttt{i}{+}1, \texttt{i}_1{+}1, \texttt{c}), \textbf{noop}(\texttt{i}_1, \texttt{i}{+}1, \texttt{c})\}, \texttt{i}_1{+}1)$
    case $(\pi(v, \delta))$
        $(\texttt{ax}_1, \texttt{i}_1) = \textbf{comp}(\delta, \texttt{i}{+}1, \texttt{e} \cup \{v\}, \texttt{c});$
        $\texttt{ax} = \{Poss(pi(th, v, x, \texttt{c}, \texttt{i}{+}1), s) \leftarrow Thread(th, s) \wedge state(th, s) = (\texttt{i}, \texttt{c})\};$
        return $(\texttt{ax} \cup \texttt{ax}_1, \texttt{i}_1)$
    case $P(t_1, \ldots, t_n)$ (where $P(x_1, \ldots, x_n)$ is a procedure)
        $\texttt{ax} = \{Poss(call(th, P, \texttt{i}{+}1, \texttt{c}), s) \leftarrow Thread(th, s) \wedge state(th, s) = (\texttt{i}, \texttt{c})\}$
            $\cup\ \textbf{bindProc}(\texttt{e}, [t_1, \ldots, t_n], [x_1, \ldots, x_n], call(th, P, \texttt{i}{+}1, \texttt{c}));$
        return $(\texttt{ax}, \texttt{i}{+}1)$
    otherwise see Part 2 on next page
```

Function $\mathtt{comp}(\delta, \mathtt{i}, \mathtt{e}, \mathtt{c})$ – Part 2 of 2

```
switch $\delta$ do
    case $(\delta_1 \parallel \delta_2)$
        $(\mathtt{ax}_1, \mathtt{i}_1) = \mathbf{comp}(\delta_1, \mathtt{i}{+}1, \mathtt{e}, \mathtt{c})$;
        $(\mathtt{ax}_2, \mathtt{i}_2) = \mathbf{comp}(\delta_2, \mathtt{i}_1{+}1, \mathtt{e}, \mathtt{c})$;
        $\mathtt{ax} = \{Poss(spawn(th, \mathtt{c}, \mathtt{i}_2{+}1, \mathtt{i}{+}1, \mathtt{i}_1{+}1), s) \leftarrow$
                $Thread(th, s) \wedge state(th, s) = (\mathtt{i}, \mathtt{c})$,
            $Poss(join(th, \mathtt{c}, \mathtt{i}_2{+}1), s) \leftarrow Thread(th, s) \wedge state(th, s) =$
        $(\mathtt{i}_2{+}1, \mathtt{c}) \wedge$
                $final([childp(th, s){-}1|th], s) \wedge final([childp(th, s){-}2|th], s)$,
            $state(th, do(join(th, \mathtt{c}, \mathtt{i}_2{+}1), s)) = (\mathtt{i}_2{+}2, \mathtt{c})$,
            $Poss(finalize(th), s) \leftarrow Thread(th) \wedge \neg final(th) \wedge$
                $state(th, s) = (\mathtt{i}_1, \mathtt{c}) \vee state(th, s) = (\mathtt{i}_2, \mathtt{c})\}$;
        return $(\mathtt{ax}_1 \cup \mathtt{ax}_2 \cup \mathtt{ax}, \mathtt{i}_2 {+}2)$
    case $(\delta^{\parallel})$
        $(\mathtt{ax}_1, \mathtt{i}_1) = \mathbf{comp}(\delta, \mathtt{i}{+}1, \mathtt{e}, \mathtt{c})$;
        $\mathtt{ax} = \{\mathbf{noop}(\mathtt{i}, \mathtt{i}_1{+}1, \mathtt{c})$,
            $Poss(spawn(th, \mathtt{c}, \mathtt{i}_1{+}1, \mathtt{i}, \mathtt{i}{+}1), s) \leftarrow$
        $Thread(th, s) \wedge state(th, s) = (\mathtt{i}, \mathtt{c})$,
            $Poss(join(th, \mathtt{c}, \mathtt{i}_1{+}1), s) \leftarrow Thread(th, s) \wedge state(th, s) =$
        $(\mathtt{i}_1{+}1, \mathtt{c}) \wedge$
                $final([childp(th, s){-}1|th], s) \wedge final([childp(th, s){-}2|th], s)$,
            $state(th, do(join(th, \mathtt{c}, \mathtt{i}_2{+}1), s)) = (\mathtt{i}_2{+}2, \mathtt{c})$,
            $Poss(finalize(th), s) \leftarrow Thread(th) \wedge \neg final(th) \wedge$
                $state(th, s) = (\mathtt{i}_1, \mathtt{c}) \vee state(th, s) = (\mathtt{i}_2, \mathtt{c})\}$;
        return $(\mathtt{ax}_1 \cup \mathtt{ax}, \mathtt{i}_1 {+}2)$
    case $(\delta_1 \rangle\!\rangle\, \delta_2)$
        $(\mathtt{ax}_1, \mathtt{i}_1) = \mathbf{comp}(\delta_1, \mathtt{i}{+}1, \mathtt{e}, \mathtt{c})$;
        $(\mathtt{ax}_2, \mathtt{i}_2) = \mathbf{comp}(\delta_2, \mathtt{i}_1{+}1, \mathtt{e}, \mathtt{c})$;
        $\mathtt{ax} = \{Poss(spawn(th, \mathtt{c}, \mathtt{i}_2{+}1, \mathtt{i}{+}1, \mathtt{i}_1{+}1), s) \leftarrow$
                $Thread(th, s) \wedge state(th, s) = (\mathtt{i}, \mathtt{c})$,
            $Poss(join(th, \mathtt{c}, \mathtt{i}_2{+}1), s) \leftarrow Thread(th, s) \wedge state(th, s) =$
        $(\mathtt{i}_2{+}1, \mathtt{c}) \wedge$
                $final([childp(th, s){-}1|th], s) \wedge final([childp(th, s){-}2|th], s)$,
            $state(th, do(join(th, \mathtt{c}, \mathtt{i}_2{+}1), s)) = (\mathtt{i}_2{+}2, \mathtt{c})$,
            $Prio(th_1, th_2, do(spawn(th, \mathtt{c}, \mathtt{i}_2{+}1, \mathtt{i}{+}1, \mathtt{i}_1{+}1), s)) \leftarrow$
                $th_1 = [childp(th, s) + 1|th] \wedge th_2 = [childp(th, s) + 2|th]$,
            $\neg Prio(th_1, th_2, do(join(th, \mathtt{c}, \mathtt{i}_2{+}1), s)) \leftarrow$
                $th_1 = [childp(th, s) + 1|th] \wedge th_2 = [childp(th, s) + 2|th]$,
            $Poss(finalize(th), s) \leftarrow Thread(th) \wedge \neg final(th) \wedge$
                $state(th, s) = (\mathtt{i}_1, \mathtt{c}) \vee state(th, s) = (\mathtt{i}_2, \mathtt{c})\}$;
        return $(\mathtt{ax}_1 \cup \mathtt{ax}_2 \cup \mathtt{ax}, \mathtt{i}_2 {+}2)$
```

finalize(*th*)*:* Mark thread *th* as final.

In the algorithm we use the auxiliary functions **test**, **rtest**, **noop**, **rnoop** to create additional transitions in the generated Petri net, which may be conditional (*test*) or unconditional (*noop*). In these algorithms $\phi(s)|_{\mathtt{V}}$ denotes the formula resulting from substituting each occurrence of v by x_v for every pair $(v, x_v) \in \mathtt{V}$.

Function $\mathtt{test}(\phi, \mathtt{i}_1, \mathtt{i}_2, \mathtt{e}, \mathtt{c})$

```
1 $\mathtt{V} = \{(v, x_v) \mid v \in \mathtt{e} \wedge \phi$ mentions $v\}$ ;                // $x_v$ a new var.
2 return $\{Forced(th, do(test(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s)),$
3     $\neg Forced(th', do(test(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s)) \leftarrow Thread(th', s) \wedge th' \neq th,$
4     $state(th, do(test(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s)) = (\mathtt{i}_2, \mathtt{c}),$
5     $Poss(test(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s) \leftarrow Thread(th, s) \wedge state(th, s) = (\mathtt{i}_1, \mathtt{c}) \wedge$
6             $(\bigwedge_{(v,x_v)\in\mathtt{V}} map(th, sp(th, s), v, s) = x_v) \wedge \phi(s)|_{\mathtt{V}}$
7             $\wedge \neg blockeds(th, sp(th, s), \mathtt{i}_2, \mathtt{c})\} \cup$
8     shadow$(test(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), \neg Forced(Th, s))$
```

Function $\mathtt{rtest}(\phi, \mathtt{i}_1, \mathtt{i}_2, \mathtt{e}, \mathtt{c})$

```
1 $\mathtt{V} = \{(v, x_v) \mid v \in \mathtt{e} \wedge \phi$ mentions $v\}$ ;                // $x_v$ a new var.
2 return $\{\neg Forced(th', do(rtest(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s)) \leftarrow Thread(th', s),$
3         $state(th, do(rtest(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s)) = (\mathtt{i}_2, \mathtt{c}),$
4         $Poss(rtest(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s) \leftarrow Thread(th, s) \wedge state(th, s) = (\mathtt{i}_1, \mathtt{c}) \wedge$
5             $(\bigwedge_{(v,x_v)\in\mathtt{V}} map(th, sp(th, s), v, s) = x_v) \wedge \phi(s)|_{\mathtt{V}}\}$
```

Function $\mathtt{noop}(\mathtt{i}_1, \mathtt{i}_2, \mathtt{c})$

```
1 return $\{Poss(noop(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s) \leftarrow$
2             $Thread(th, s) \wedge state(th, s) = (\mathtt{i}_1, \mathtt{c}) \wedge \neg blockeds(th, sp(th, s), \mathtt{i}_1, \mathtt{c}),$
3         $Forced(th, do(noop(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s)),$
4         $state(th, do(noop(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s)) = (\mathtt{i}_2, \mathtt{c})\}$
5         $\cup$ shadow$(noop(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), true)$
```

The following condition, **bindProc**, is required for handling program variables in the positions of procedure calls. The variables $\mathtt{t_i}$ serve as actual parameters and $\mathtt{x_i}$ as formal parameters. Also note that we apply *call-by-value* by evaluating all actual parameters which are not program variables before passing

```
Function rnoop(i_1, i_2, c)
1 return {$Poss(rnoop(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s) \leftarrow Thread(th, s) \wedge state(th, s) = (\mathtt{i}_1, \mathtt{c})$,
2         $state(th, do(rnoop(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s)) = (\mathtt{i}_2, \mathtt{c})$}
```

them to the procedure.

$$\mathbf{bindProc}(\mathtt{e}, [\mathtt{t}_1, \dots, \mathtt{t}_n], [\mathtt{x}_1, \dots, \mathtt{x}_n], \mathtt{a}) \overset{\text{def}}{=} \bigcup_{\mathtt{j\ s.t.\ t_j \in e}} \{map(th, sp(th, s){+}1, \mathtt{x}_\mathtt{j}, do(\mathtt{a}, s)) = map(th, sp(th, s), \mathtt{t}_\mathtt{j}, s)\} \cup \bigcup_{\mathtt{j\ s.t.\ t_j \notin e}} \{map(th, sp(th, s){+}1, \mathtt{x}_\mathtt{j}, do(\mathtt{a}, s)) = \mathtt{t}_\mathtt{j}(s)\}$$

The following function (**shadow**) creates axioms which are required for a limited form of backtracking. This is occasionally required to realize synchronized while's and if's. Consider the program (**if** ϕ **then** a **else** b) $\|$ c, and imagine that initially ϕ is false, but that c makes it true. Then still, the sequence $[c, a]$ is not permitted: testing ϕ and executing the next action has to be atomic and may not be interrupted by other threads. In our compilation we realize this through the use of a particular $Forced(th, s)$ fluent, stating that only thread th may execute next. Usually this fluent is false, but bookkeeping-actions make it true. This way, after performing the test action for ϕ, only a may execute next. However, imagine a is not executable. We need to lift the force, and allow other threads to execute, but in order to implement synchronized if's correctly, we need to *backtrack* the thread to a state before the test first. In the example above, this is because after executing c, the else-case of the conditional must be executed. Backtracking is realized by keeping a stack of previous configurations of bookkeeping fluents, to which the system can revert to when necessary. The following axioms implement the pushing of backtracking information onto the stack. The $backtrack(th)$ action, see below, implements the restoration, i.e. popping from the stack.

$\mathbf{shadow}(\mathtt{a}, \psi) \overset{\text{def}}{=} \{$

$S\_Thread(b, x, do(\mathtt{a}, s)) \leftarrow b = backtp(s) \wedge Thread(x, s) \wedge \psi,$
$\neg S\_Thread(b, x, do(\mathtt{a}, s)) \leftarrow b = backtp(s) \wedge \neg Thread(x, s) \wedge \psi,$
$s\_state(b, x, do(\mathtt{a}, s)) = v \leftarrow b = backtp(s) \wedge state(x, s) = v \wedge \psi,$
$s\_stack(b, x, y, do(\mathtt{a}, s)) = v \leftarrow b = backtp(s) \wedge stack(x, y, s) = v \wedge \psi,$
$s\_sp(b, x, do(\mathtt{a}, s)) = v \leftarrow b = backtp(s) \wedge sp(x, s) = v \wedge \psi,$
$s\_map(b, t, p, x, do(\mathtt{a}, s)) = v \leftarrow b{=}backtp(s) \wedge map(t, p, x, s){=}v \wedge \psi,$
$s\_childp(b, x, do(\mathtt{a}, s)) = v \leftarrow b = backtp(s) \wedge childp(x, s) = v \wedge \psi,$
$S\_Prio(b, x, y, do(\mathtt{a}, s)) \leftarrow b = backtp(s) \wedge Prio(x, y, s) \wedge \psi,$
$\neg S\_Prio(b, x, y, do(\mathtt{a}, s)) \leftarrow b = backtp(s) \wedge \neg Prio(x, y, s) \wedge \psi,$
$S\_Forced(b, x, do(\mathtt{a}, s)) \leftarrow b = backtp(s) \wedge Forced(x, s) \wedge \psi,$
$\neg S\_Forced(b, x, do(\mathtt{a}, s)) \leftarrow b = backtp(s) \wedge \neg Forced(x, s) \wedge \psi,$
$S\_blockeds(b, x, p, y, c, do(\mathtt{a}, s)) \leftarrow b = backtp(s) \wedge blockeds(x, p, y, c, s) \wedge \psi,$
$S\_blocked(b, x, p, y, c, x, do(\mathtt{a}, s)) \leftarrow b = backtp(s) \wedge blocked(x, p, y, c, x, s) \wedge \psi,$

$backtp(do(\mathtt{a}, s)) = v \leftarrow v = backtp(s) + 1 \wedge \psi \quad \}$

B Program-Independent Axioms

The default dynamics of the involved bookkeeping actions, which are program independent, are described by the following axioms (cf. Step 2 of the compilation described in Section 3).
For procedure calls:

$$\begin{aligned}
\mathtt{ax}_{\text{procs}} \overset{\text{def}}{=} \{ \quad & sp(th, do(call(th, x_1, x_2, x_3), s)) = y \leftarrow y = sp(th, s){+}1,\\
& state(th, do(call(th, P, x_1, x_2), s)) = y \leftarrow y = (0, P),\\
& stack(th, v, do(call(th, x_1, i, \mathtt{c}), s)) {=} y \leftarrow y {=} (i, \mathtt{c}) \wedge v = sp(th, s){+}1,\\
& Forced(th, do(call(th, x_1, x_2, x_3), s)) = y \leftarrow true,\\
& state(th, do(return(th), s) = y \leftarrow y = stack(th, sp(th, s), s),\\
& sp(th, do(return(th), s)) = y \leftarrow y = sp(th, s){-}1 \quad \}
\end{aligned}$$

For concurrency:

$$\begin{aligned}
&\mathtt{ax}_{\text{conc}} \overset{\text{def}}{=} \{\\
&\quad Thread(th', do(spawn(th, \mathtt{c}, x_1, x_2, x_3), s)) \leftarrow\\
&\qquad th' = [childp(th, s)|th] \vee th' = [childp(th, s){+}1|th],\\
&\quad childp(th, do(spawn(th, \mathtt{c}, x_1, x_2, x_3), s)) {=} y {\leftarrow} y {=} childp(th, s){+}2,\\
&\quad sp(th', do(spawn(th, \mathtt{c}, x_1, x_2, x_3), s)) = y \leftarrow\\
&\qquad y = 0 \wedge (th' = [childp(th, s)|th] \vee th' = [childp(th, s) + 1|th]),\\
&\quad childp(th', do(spawn(th, \mathtt{c}, x_1, x_2, x_3), s)) = y \leftarrow\\
&\qquad y = 0 \wedge (th' = [childp(th, s)|th] \vee th' = [childp(th, s) + 1|th]),\\
&\quad Parent(\bar{th}, th', do(spawn(th, \mathtt{c}, x_1, x_2, x_3), s)) \leftarrow\\
&\qquad (th' = [childp(th, s)|th] \vee th' = [childp(th, s) + 1|th]) \wedge (\bar{th} = th \vee Parent(\bar{th}, th)),\\
&\quad state(th, do(spawn(th, \mathtt{c}, i, x_1, x_2), s)) = y \leftarrow y = (i, \mathtt{c}),\\
&\quad state(th', do(spawn(th, \mathtt{c}, x_1, i, x_2), s)) = y \leftarrow y = (i, \mathtt{c}) \wedge th' = [childp(th, s)|th],\\
&\quad state(th', do(spawn(th, \mathtt{c}, x_1, x_2, i), s)) = y \leftarrow y = (i, \mathtt{c}) \wedge th' = [childp(th, s) + 1|th],\\
&\quad map(th', p, v, do(spawn(th, \mathtt{c}, x_1, x_2, i), s)) = x \leftarrow\\
&\qquad map(th, p, v, do(spawn(th, \mathtt{c}, x_1, x_2, i), s)) = x \wedge th' = [childp(th, s)|th],\\
&\quad Prio(th', x, do(spawn(th, \mathtt{c}, x_1, x_2, x_3), s)) \leftarrow\\
&\qquad Prio(th, x) \wedge (th' = [childp(th, s)|th] \vee th' = [childp(th, s) + 1|th]),\\
&\quad Prio(x, th', do(spawn(th, \mathtt{c}, x_1, x_2, x_3), s)) \leftarrow\\
&\qquad Prio(x, th) \wedge (th' = [childp(th, s)|th] \vee th' = [childp(th, s) + 1|th]),\\
&\quad backtp(th', do(spawn(th, \mathtt{c}, x_1, x_2, x_3), s)) = 0 \leftarrow\\
&\qquad (th' = [childp(th, s)|th] \vee th' = [childp(th, s) + 1|th]),\\
&\quad \neg final(th', do(spawn(th, \mathtt{c}, x, y, z), s)) \leftarrow\\
&\qquad (th' = [childp(th, s)|th] \vee th' = [childp(th, s){+}1|th]),\\
&\quad \neg\, Thread(th', do(join(th, x_1, x_2), s)) \leftarrow\\
&\qquad childp(th, s) > 1 \wedge (th' = [childp(th, s){-}1|th] \vee th' = [childp(th, s){-}2|th]),\\
&\quad childp(th, do(join(th, x_1, x_2), s)) = y \leftarrow childp(th, s) > 1 \wedge y = childp(th, s){-}2,
\end{aligned}$$

$Forced(th, do(join(th, x_1, x_2), s)) \leftarrow Forced([childp(th, s)|th],$
$final(th, do(finalize(th), s)) \leftarrow true\}$

For program variables:

$$\texttt{ax}_\pi \overset{\text{def}}{=} \{ \quad state(th, do(pi(th, v, x, \texttt{c}, i), s)) = y \leftarrow y = (i, \texttt{c}),$$
$$Forced(th, do(pi(th, v, x, \texttt{c}, i), s)) = y \leftarrow true,$$
$$map(th, p, v, do(pi(th, v, x, \texttt{c}, i), s)) = x \leftarrow p = sp(th, s) \quad \}$$

The following axioms realize the backtracking described earlier, i.e. the restoration of an earlier configuration of the bookkeeping fluents. This is only required when concurrency is used.

$\texttt{ax}_{backtrack} \overset{\text{def}}{=} \{$
$\neg Thread(x, do(backtrack(th), s)) \leftarrow \neg S\_Thread(backtp\text{–}1, x, s),$
$Thread(x, do(backtrack(th), s)) \leftarrow S\_Thread(backtp\text{–}1, x, s),$
$state(x, do(backtrack(th), s)) = v \leftarrow s\_state(backtp\text{–}1, x, s) = v,$
$stack(x, y, do(backtrack(th), s)) = v \leftarrow s\_stack(backtp\text{–}1, x, y, s) = v,$
$sp(x, do(backtrack(th), s)) = v \leftarrow s\_sp(backtp\text{–}1, x, s) = v,$
$map(t, p, x, do(backtrack(th), s)) = v \leftarrow s\_map(backtp\text{–}1, t, p, x, s) = v,$
$childp(x, do(backtrack(th), s)) = v \leftarrow s\_childp(backtp\text{–}1, x) = v,$
$Prio(x, y, do(backtrack(th), s)) \leftarrow S\_Prio(backtp\text{–}1, x, y, s),$
$\neg Prio(x, y, do(backtrack(th), s)) \leftarrow \neg S\_Prio(backtp\text{–}1, x, y, s),$
$Forced(x, do(backtrack(th), s)) \leftarrow S\_Forced(backtp\text{–}1, x, s),$
$\neg Forced(x, do(backtrack(th), s)) \leftarrow \neg S\_Forced(backtp\text{–}1, x, s),$
$blockeds(th, p, i, c, do(backtrack(th), s)) \leftarrow S\_blockeds(backtp\text{–}1, th, p, i, c, s),$
$\neg blockeds(th, p, i, c, do(backtrack(th), s)) \leftarrow \neg S\_blockeds(backtp\text{–}1, th, p, i, c, s),$
$blocked(th, p, i, c, x, do(backtrack(th), s)) \leftarrow S\_blocked(backtp\text{–}1, th, p, i, c, x, s),$
$\neg blocked(th, p, i, c, x, do(backtrack(th), s)) \leftarrow \neg S\_blocked(backtp\text{–}1, th, p, i, c, x, s),$
$backtp(do(backtrack(th), s) = v \leftarrow v = backtp\text{–}1,$
$blockeds(th, p, i, c, do(noop(th, c, i), s)) \leftarrow p = sp(th, s),$
$S\_blockeds(b, th, p, i, c, do(noop(th, c, i), s)) \leftarrow p = sp(th, s) \wedge b = backtp,$
$blockeds(th, p, i, c, do(test(th, i', c, i), s)) \leftarrow p = sp(th, s),$
$S\_blockeds(b, th, p, i, c, do(test(th, i', c, i), s)) \leftarrow p = sp(th, s) \wedge b = backtp,$
$blocked(th, p, i, c, x, do(pi(th, i', x, c, i), s)) \leftarrow p = sp(th, s),$
$S\_blocked(b, th, p, i, c, x, do(pi(th, i', x, c, i), s)) \leftarrow p = sp(th, s) \wedge b = backtp,$
$blockeds(th, p, i, c, do(join(th, c, i), s)) \leftarrow p = sp(th, s),$
$S\_blockeds(b, th, p, i, c, do(join(th, c, i), s)) \leftarrow p = sp(th, s) \wedge b = backtp,$
$blockeds(th, p, i, c, do(call(th, i', c, i), s)) \leftarrow p = sp(th, s),$
$S\_blockeds(b, th, p, i, c, do(call(th, i', c, i), s)) \leftarrow p = sp(th, s) \wedge b = backtp\}$
$\cup\, \mathbf{shadow}(do(call(th, i', c, i), s), \neg Forced(th))$
$\cup\, \mathbf{shadow}(do(join(th, c, i), s), \neg Forced(th))$
$\cup\, \mathbf{shadow}(do(\pi(th, i', x, c, i), true)$
$\cup\, \{Poss(backtrack(th), s) \leftarrow backtp(s) > 0 \wedge Thread(th) \wedge$
$\quad Forced(th) \wedge \neg CanTrans(th) \wedge (\nexists t).Parent(th, t) \wedge Thread(t) \wedge CanTrans(t)\}$

Intuitively, above blocking effects ensure that after backtracking not the same pseudo actions are performed, and thus creating a cycle. The unblocking effects of real actions (see below), imply that a bookkeeping action can be repeated after a real action has taken place – and thus, the state of the world may have changed. Backtracking is only possible, when a forced thread cannot execute any other action, and none of its descendant threads can either (cf. Step 4).

The following axioms describe some of the values of above used bookkeeping fluents in the initial situation S_0:

$$\begin{aligned}\mathtt{ax}_{S_0} \overset{\text{def}}{=} \{ \quad & \mathit{Thread}([0], S_0) \leftarrow \mathit{true},\\ & \mathit{state}([0], S_0) = (0, \text{'}\mathit{main}\text{'}) \leftarrow \mathit{true},\\ & \mathit{sp}([0], S_0) = 0 \leftarrow \mathit{true},\\ & \mathit{stack}([0], 0, S_0) = \text{'}\mathit{final}\text{'} \leftarrow \mathit{true},\\ & \mathit{childp}([0], S_0) = 0 \leftarrow \mathit{true} \quad \}\end{aligned}$$

We further have the following additional bookkeeping effects for *real actions*, where $\mathcal{A}$ denotes the set of all *domain actions*, i.e. primitive actions of the original theory $\mathcal{D}$:

$$\begin{aligned}\mathtt{ax}_{real} \overset{\text{def}}{=} \{ \quad & \neg \mathit{blockeds}(th, p, i, c, do(\alpha, s)),\\ & \neg \mathit{blocked}(th, p, i, c, x, do(\alpha, s)),\\ & \neg \mathit{final}(th, do(\alpha, s)),\\ & \neg \mathit{Forced}(th, do(\alpha, s)) \quad \}_{\alpha \in \mathcal{A} \cup \{rtest\}}\end{aligned}$$

The union of these sets forms the set of common, program independent axioms:

$$\mathtt{ax}_{\text{common}} \overset{\text{def}}{=} \quad \mathtt{ax}_{\text{procs}} \cup \mathtt{ax}_{\text{conc}} \cup \mathtt{ax}_{\pi} \cup \mathtt{ax}_{backtrack} \cup \mathtt{ax}_{S_0} \cup \mathtt{ax}_{real}$$

C Proof of Theorem 1

C.1 Definitions

We use the following auxiliary definitions for the proofs.

- If $\mathcal{D}$ is a basic action theory, then we denote by $\mathcal{D}_{\delta,\mathtt{i},\mathtt{e},\mathtt{c},\mathtt{i}'}$ the basic action theory resulting from computing $(\mathtt{ax}, \mathtt{i}') = \mathbf{comp}(\delta, \mathtt{i}, \mathtt{e}, \mathtt{c})$ and performing steps 2 to 6 of the compilation on $\mathtt{ax}$. When $\mathtt{i}$ is omitted, we mean $\mathtt{i} = 0$. Similarly, if $\mathtt{e}$ is omitted, we assume $\mathtt{e} = \emptyset$.

- As mentioned earlier, in action theories resulting from a compilation, all actions take a thread name as their first argument, i.e. domain actions (defined in the original theory $\mathcal{D}$) receive an additional argument in the compiled theories. In the proof below we sometimes refer to situation terms in compiled theories containing action terms without this additional thread argument. We interpret these – in the context of a compiled theory – as macros, expanding into action terms with a dummy thread $[-1]$ as their first argument (e.g. $A(t_1, \ldots, t_n)$ becomes $A([-1], t_1, \ldots, t_n)$). Even though the implied situation term is not executable in the respective theory (since that dummy thread is never active), it serves our purposes, as the action still has its intended effects on the state of the world, i.e. on all non-bookkeeping fluents. This convention ensures that all situation terms of the original theory $\mathcal{D}$ are also situation terms in any new theory, resulting from the compilation of some program, and represent the same state of the world.

- For two situation terms S and $S' = do(\vec{a}, S)$, we refer to the action sequence $\vec{a}$ by $S' - S$. We denote the concatenation of two action sequences $\vec{a}, \vec{b}$ by $\vec{a} \cdot \vec{b}$.

- We call a situation S' a *continuation* of another situation S, if $S \sqsubseteq S'$. We call a continuation *proper* if $\sigma = S' - S$ is non-empty. We say that a continuation is *in thread th* to express that all actions in σ are executed in thread *th*, i.e. their first argument is *th*.

- For two situations s, s' with $s \sqsubset s'$, we write $exec(s, s')$ to state that the part of s' that extends s is executable, formally:

$$exec(s, s') \stackrel{\text{def}}{=} (\forall a, s^*).s \sqsubset do(a, s^*) \sqsubseteq s' \supset Poss(a, s^*).$$

  Similarly, $exec_{th}(s, s')$ denotes that all actions in the part of s' that extends s that are in thread *th* or any of its descendants, are executable. Formally:

$$exec_{th}(s, s') \stackrel{\text{def}}{=}$$
$$(\forall a, th', \vec{x}, s^*).s \sqsubset do(a(th', \vec{x}), s^*) \sqsubseteq s' \wedge Suffix(th, th') \supset Poss(a(th', \vec{x}), s^*).$$

  where $Suffix(L_1, L_2)$ holds if L_1 and L_2 are lists and the former is a suffix of the latter.

- We call domain actions and *rtest* actions *real actions*, and call all other bookkeeping actions *pseudo actions*.

We define a new predicate, $\widehat{\mathit{Trans}}$, which is just like *Trans* but allows the execution of arbitrary actions at any point during program execution. These actions are marked, so that later it can be distinguished whether they are the result of executing program steps or whether they were injected. This is to formalize the notion of executability of a program, conditioned on the execution of other actions in concurrently executing threads.

Definition 1. Let $\mathcal{D}$ be any basic action theory whose set of actions is $\mathcal{A}$. Then $\widehat{\mathcal{D}}$ is just like $\mathcal{D}$, except that the set of actions is $\mathcal{A} \cup \widehat{\mathcal{A}}$ where $\widehat{\mathcal{A}} = \{\widehat{a} \mid a \in \mathcal{A}\}$, and the effects and preconditions of $\widehat{a}$ are equal to those of a.

We define:

$$\widehat{\mathit{Trans}}(\delta, s, \delta', s') \overset{\text{def}}{=} \mathit{Trans}(\delta, s, \delta', s') \vee (\delta' = \delta \wedge (\exists \widehat{a}).s' = do(\widehat{a}, s)).$$

and define $\widehat{\mathit{Trans}}^*$ just like Trans^* but for $\widehat{\mathit{Trans}}$ instead of *Trans*. Further, $\mathit{pure}(s)$ denotes the situation obtained by replacing any marked action in s by its unmarked counterpart.

In order to compare two situation terms $\widehat{S}$ in $\widehat{\mathcal{D}}$ and S_δ in a compiled theory $\mathcal{D}_\delta$, we define $S_\delta \mathrel{\widehat{=}_{th}} \widehat{S}$ as follows:

$$\begin{array}{rll}
do(a(th', \vec{x}), s) \mathrel{\widehat{=}_{th}} do(a(\vec{x}), s') & \text{if} & s \mathrel{\widehat{=}_{th}} s' \text{ and } a \in \mathcal{A} \\
 & & \text{and } (th' = th \text{ or } \mathit{Suffix}(th, th')) \\
do(a(th', \vec{x}), s) \mathrel{\widehat{=}_{th}} do(\widehat{a}(\vec{x}), s') & \text{if} & s \mathrel{\widehat{=}_{th}} s' \text{ and } a \in \mathcal{A} \\
 & & \text{and } th' \neq th \text{ and } \neg \mathit{Suffix}(th, th') \\
do(a(th', \vec{x}), s) \mathrel{\widehat{=}_{th}} s' & \text{if} & s \mathrel{\widehat{=}_{th}} s' \text{ and } a \notin \mathcal{A} \\
s \mathrel{\widehat{=}_{th}} s & &
\end{array}$$

That is, the actions are matched in order, where each domain action executed in *th* or its descendants matches an action in $\mathcal{D}$, each domain action executed in any other thread matches an inserted action, and all non-domain actions are ignored.

C.2 Lemmata

We use the following auxiliary lemma.

Lemma 1. All precondition axioms in $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ are conjunctions where one of the conjuncts is either of the form $\mathit{state}(th, s) = (\mathtt{i}_1, \mathtt{c}_1) \vee \cdots \vee \mathit{state}(th, s) = (\mathtt{i}_n, \mathtt{c}_n)$, or *false*, and $\mathtt{i} \leq \mathtt{i}_j < \mathtt{i}'$ for all $1 \leq j \leq n$, and another conjunct is $\mathit{Thread}(th, s)$. Further, let s be any situation in $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ such that $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{Thread}(th, s) \wedge \mathit{state}(th, s) = (\mathtt{i}, \mathtt{c})$. Then there is no continuation s' of s and an integer $\mathtt{i}'' > \mathtt{i}'$ such that $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{exec}_{th}(s, s') \wedge \mathit{state}(th, s') = (\mathtt{i}'', \mathtt{c})$.
Proof: The theorem follows by construction of the compilation. □

Intuitively, the first part of the lemma states that all action preconditions in the compiled theory explicitly enumerate all *state*'s in which the action can execute, these states are within the parameters input and output by the compilation algorithm, and that only active threads can execute. The second part says that executable actions do not result in states whose number is beyond the final state, defined by the compilation. As a consequence of this lemma we have that if in a situation s a thread *th* is in a state $\mathtt{i}^*$ of some context $\mathtt{c}$ (i.e. $\mathit{state}(\mathit{th}, s) = (\mathtt{i}^*, \mathtt{c})$), then the only actions that might be executable in s in *th* are those listed in $\mathtt{ax}$ of $(\mathtt{ax}, \mathtt{i}') = \mathbf{comp}(\delta, \mathtt{i}, \emptyset, \mathtt{c})$ for some $\mathtt{i} \leq \mathtt{i}^*$ and some $\mathtt{i}' > \mathtt{i}^*$.

The following is going to be our main lemma. The theorem is going to follow as a special case of it. The lemma indeed states something stronger than the actual theorem, namely, intuitively, that the compiled theory and the original semantics admit the same set of future situations, irrespective of whether these situations denote complete program execution or can be extended into one. This is needed in order to make the induction work.

Lemma 2. Let δ be any ConGolog program, and S any ground situation term in $\mathcal{D}$. Then:

"$\Rightarrow$"

1. for any thread name *th*, integer $\mathtt{i}$, and situation S_δ in $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ such that $\mathit{filter}(S_\delta, \mathcal{D}) = S$ and $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{Thread}(\mathit{th}, S_\delta) \wedge \mathit{state}(\mathit{th}, S_\delta) = (\mathtt{i}, \mathtt{c})$, if there exists a continuation S'_δ of S_δ such that $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{exec}_{\mathit{th}}(S_\delta, S'_\delta)$ then there is a program δ' and a continuation $\widehat{S}'$ of S in $\widehat{\mathcal{D}}$ such that $S'_\delta \mathrel{\widehat{=}}_{\mathit{th}} \widehat{S}'$ and $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta, S, \delta', \widehat{S}')$; and
2. if further $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{state}(\mathit{th}, S'_\delta) = (\mathtt{i}', \mathtt{c})$, then also $\widehat{\mathcal{D}} \models \mathit{Final}(\delta', \widehat{S}')$.

"$\Leftarrow$"

1. if there is a program δ' and a continuation $\widehat{S}'$ of S in $\widehat{\mathcal{D}}$ such that $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta, S, \delta', \widehat{S}')$ then, for any thread name *th*, integer $\mathtt{i}$, and situation S_δ in $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ such that $\mathit{filter}(S_\delta, \mathcal{D}) = S$ and $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{Thread}(\mathit{th}, S_\delta) \wedge \mathit{state}(\mathit{th}, S_\delta) = (\mathtt{i}, \mathtt{c})$, there exists a continuation S'_δ of S_δ such that $S'_\delta \mathrel{\widehat{=}}_{\mathit{th}} \widehat{S}'$ and $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{exec}_{\mathit{th}}(S_\delta, S'_\delta)$; and
2. if further $\widehat{\mathcal{D}} \models \mathit{Final}(\delta', \widehat{S}')$, then also $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{state}(\mathit{th}, S'_\delta) = (\mathtt{i}', \mathtt{c})$.

Note that as a special case of this, if $\widehat{S}'$ does not mention any marked actions, then also S'_δ does not mention any actions not in thread *th* or its descendants, and vice versa.

Proof of Lemma 2 for programs without the π construct:
The proof proceeds by induction over the structure of program δ. We refer to the definition of *Trans* and *Final* of [De Giacomo *et al.*, 2000].

The induction has several base cases:

$\delta = nil$**:**

$\Rightarrow$:

By definition of **comp**, we have that $\mathbf{comp}(\delta, \mathtt{i}, \emptyset, \mathtt{c}) = (\emptyset, \mathtt{i})$. Since `ax` is empty, so is the disjunction of ϕ's (preconditions) in Step 5 of the compilation (cf. Lemma 1). Since an empty disjunction is equivalent to *false*, no action is ever possible. Hence, the new action theory $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ has no executable situations which are proper continuations of S_δ and which mention domain actions in *th*. Since $\mathcal{D} \models (\forall s) \mathit{Final}(nil, s)$, the second implication trivially holds for $S = \mathit{filter}(S_\delta)$.

$\Leftarrow$:

By definition, $\mathcal{D} \models (\forall s \not\exists \delta', s') \mathit{Trans}(nil, s, \delta', s')$. Hence, $\widehat{S}'$ cannot contain any unmarked actions. Any situation S'_δ such that $S'_\delta \mathrel{\widehat{=}_{th}} \widehat{S}'$, hence does not contain any actions in *th*. Since by construction the only domain actions that can change the state of thread *th* are those in *th* itself, we get that any such S'_δ satisfies:

$$\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{exec}_{th}(S_\delta, S'_\delta) \wedge \mathit{state}(th, S'_\delta) = (\mathtt{i}, \mathtt{c})$$

and hence the thesis for this case.

$\delta = A(t_1, \ldots, t_n)$ **where** A **is a primitive domain action:**

$\Rightarrow$:

By construction of **comp** for this case and due to Lemma 1, the only potentially executable action in S_δ in $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ in thread *th* is $A(th, x_1, \ldots, x_n)$, where, by definition of **comp**, $x_i = t_i(S_\delta)$ (recall that $\mathtt{e} = \emptyset$). It is possible, only if A's original preconditions are satisfied as well. By definition of **comp** this action causes $\mathit{state}(th, do(A(th, x_1, \ldots, x_n), S_\delta)) = (\mathtt{i}', \mathtt{c})$, and hence does not admit any further actions in thread *th*, due to Lemma 1. By definition of *Trans* it follows that $\mathcal{D} \models \mathit{Trans}(A(t_1, \ldots, t_n), S, nil, do(A(t_1(S), \ldots, t_n(S)), S))$, and obviously $do(A(th, x_1, \ldots, x_n), S_\delta) \mathrel{\widehat{=}_{th}} do(A(t_1(S), \ldots, t_n(S)), S)$ from the above, and the assumption that $\mathit{filter}(S_\delta, \mathcal{D}) = S$.

Further, by definition, $\mathcal{D} \models \mathit{Final}(nil, do(A(t_1(S), \ldots, t_n(S)), S))$, hence the thesis for this case ($\widehat{S}' = do(A(t_1(S), \ldots, t_n(S)), S)$).

$\Leftarrow$:

By definition of *Trans*, $\widehat{S}' = do(A(t_1(S), \ldots, t_n(S)), S)$ is the only continuation of S such that there exists a program δ' such that $\mathcal{D} \models \mathit{Trans}(A(t_1, \ldots, t_n), S, \delta', \widehat{S}')$, and hence the only continuation in $\widehat{\mathcal{D}}$ mentioning only unmarked actions such that $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}(A(t_1, \ldots, t_n), S, \delta', \widehat{S}')$. Hence, any continuation S'_δ such that $S'_\delta \mathrel{\widehat{=}_{th}} \widehat{S}'$ containing $A(th, t_1(S), \ldots, t_n(S))$ as its only action in *th* and not containing any pseudo-actions, satisfies the condition. It hence follows, as before:

$$\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{exec}_{th}(S_\delta, S'_\delta) \wedge \mathit{state}(th, S'_\delta) = (\mathtt{i}', \mathtt{c}).$$

$\delta = \phi?$**:**

$\Rightarrow$:

By definition of **comp** and due to Lemma 1, the only potentially possible action in situation S_δ in thread *th*, is $rtest(th, \mathtt{i}, \mathtt{i}', \mathtt{c})$, and after that no more actions are possible in *th*. Also note that $\mathtt{i}' = \mathtt{i} + 1$. Hence, we can assume that S'_δ only contains this action in *th*. Let σ, σ' be such that $S'_\delta = do(\sigma', do(rtest(th, \mathtt{i}, \mathtt{i}', \mathtt{c}), do(\sigma, S_\delta)))$, i.e. the sequences of actions (in other threads) before and after this action. The action is, by construction, only possible if $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \phi(do(\sigma, S_\delta))$ (recall, $\mathtt{e} = \emptyset$). Since ϕ cannot mention any of the bookkeeping fluents introduced by the compilation, we have $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \phi(do(\sigma, S_\delta))$ iff $\mathcal{D} \models \phi(filter(do(\sigma, S_\delta), \mathcal{D}))$. Let $\widehat{S}'$ be any continuation of S such that $S'_\delta \mathrel{\widehat{=}}_{th} \widehat{S}'$. By definition of $\mathrel{\widehat{=}}_{th}$ we have that there is a situation $\widehat{S}''$ such that $S \sqsubset \widehat{S}'' \sqsubset \widehat{S}'$ and which satisfies $do(\sigma, S) \mathrel{\widehat{=}}_{th} \widehat{S}''$. It follows by definition of *Trans* that $\widehat{\mathcal{D}} \models \widehat{Trans}(\phi, \widehat{S}'', nil, \widehat{S}'')$, and hence $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\phi, S, nil, \widehat{S}')$

As before, $\mathcal{D} \models (\forall s).Final(nil, s)$ and hence the thesis.

$\Leftarrow$:

By definition of *Trans*, $\widehat{S}' = S$ is the only (improper) continuation of S such that there exists a program δ' such that $\mathcal{D} \models Trans(\phi, S, \delta', \widehat{S}')$, and hence the only continuation in $\widehat{\mathcal{D}}$ mentioning only unmarked actions such that $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\phi, S, \delta', \widehat{S}')$. Hence, any situation S'_δ such that $S'_\delta \mathrel{\widehat{=}}_{th} \widehat{S}'$ containing no actions in *th* and not containing any pseudo-actions, satisfies the condition. It hence follows, as before:

$$\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th}(S_\delta, S'_\delta) \wedge state(th, S'_\delta) = (\mathtt{i}', \mathtt{c})$$

The induction steps regard all other programming constructs and are as follows:

$(\delta_1; \delta_2)$**:**

$\Rightarrow$:

There are two cases to distinguish: (a) S'_δ is such that the integer $\mathtt{i}^*$ such that $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models state(th, S'_\delta) = (\mathtt{i}^*, \mathtt{c})$ is $\leq \mathtt{i}_1$, where $\mathtt{i}_1$ is as defined by **comp** for this case, or (b) $\mathtt{i}^* \geq \mathtt{i}_1$. In case (a) the thesis follows immediately by induction hypothesis. We hence only need to consider case (b).

By Lemma 1 and definition of **comp** there is a prefix σ_1 of $S'_\delta - S_\delta$, such that

$$\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th}(S_\delta, do(\sigma_1, S_\delta)) \wedge state(th, do(\sigma_1, S_\delta)) = (\mathtt{i}_1, \mathtt{c})$$

where $\mathtt{i}_1$ is the integer defined in **comp** for this case. By induction hypothesis, there hence is a continuation $\widehat{S}'_1$ of S in $\widehat{\mathcal{D}}$ and a program δ' such that $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_1, S, \delta', \widehat{S}'_1) \wedge Final(\delta', \widehat{S}'_1)$.

Furthermore, since $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{state}(\mathit{th}, \mathit{do}(\sigma_1, S_\delta)) = (\mathtt{i}_1, \mathtt{c})$, induction hypothesis also applies to $\mathit{do}(\sigma_1, S_\delta)$ and δ_2: Since S'_δ is a continuation of $\mathit{do}(\sigma_1, S_\delta)$ which, by the initial assumption, satisfies $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{exec}_{\mathit{th}}(\mathit{do}(\sigma, S_\delta), S'_\delta)$ it follows that there also exists a ground continuation $\widehat{S}'_2$ of $\mathit{filter}(\mathit{do}(\sigma_1, S_\delta), \mathcal{D})$ in $\widehat{\mathcal{D}}$ such that there is δ' such that $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta_2, \mathit{filter}(\mathit{do}(\sigma_1, S_\delta), \mathcal{D}), \delta', \widehat{S}'_2)$. Hence $\widehat{S}' = \mathit{do}(S'_2 - \mathit{filter}(\mathit{do}(\sigma_1, S_\delta), \mathcal{D}), \mathit{do}(\widehat{S}'_1 - S, S))$ satisfies, by definition of *Trans* for sequences: $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta, S, \delta', \widehat{S}')$.

Further by induction hypothesis, if $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{state}(\mathit{th}, S'_\delta) = (\mathtt{i}', \mathtt{c})$ then also $\widehat{\mathcal{D}} \models \mathit{Final}(\delta', \widehat{S}')$. Hence the thesis.

$\Leftarrow$:

In analogy to above, we can again distinguish the two cases: (a) there is no situation $\widehat{S}''$ in $\widehat{\mathcal{D}}$ such that $S \sqsubset \widehat{S}'' \sqsubset \widehat{S}'$ and $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta, S, \mathit{nil}; \delta_2, \widehat{S}'')$, or (b) there is. In case (a) the thesis again follows immediately by induction hypothesis. In the following we show case (b).

By definition of $\widehat{\mathit{Trans}}^*$ there is a prefix σ of $\widehat{S}' - S$ such that $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta_1; \delta_2, S, \mathit{nil}; \delta_2, \mathit{do}(\sigma, S))$. Hence, by induction hypothesis, there is a continuation S'_{δ_1} for any S_δ, where the latter is as described in the lemma, such that $S'_{\delta_1} \mathrel{\widehat{=}}_{\mathit{th}} \mathit{do}(\sigma, S)$ and

$$\mathcal{D}_{\delta_1,\mathtt{i},\mathtt{c},\mathtt{i}_1} \models \mathit{exec}_{\mathit{th}}(S_\delta, S'_{\delta_1}) \wedge \mathit{state}(\mathit{th}, S'_{\delta_1}) = (\mathtt{i}_1, \mathtt{c})$$

for $\mathtt{i}_1$ as defined by **comp**.

We can once again apply induction hypothesis on the second sub-program, using the situation S'_{δ_1}, since it satisfies the constraints. The situation $\widehat{S}'$ is a continuation of $\mathit{pure}(\mathit{do}(\sigma, S))$ in $\widehat{\mathcal{D}}$ that satisfies $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta_2, \mathit{pure}(\mathit{do}(\sigma, S)), \delta', \widehat{S}')$. Hence, there exists a continuation S'_δ of S'_{δ_1} such that $S'_\delta \mathrel{\widehat{=}}_{\mathit{th}} \widehat{S}'$ and $\mathcal{D}_{\delta_2,\mathtt{i}_1,\mathtt{c},\mathtt{i}'} \models \mathit{exec}_{\mathit{th}}(S'_{\delta_1}, S'_\delta)$. It follows by definition of $\mathit{exec}_{\mathit{th}}$ that also $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{exec}_{\mathit{th}}(S_\delta, S'_\delta)$.

Further by this second application of induction hypothesis, we get that if $\widehat{\mathcal{D}} \models \mathit{Final}(\delta', \widehat{S}')$, then also $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{state}(\mathit{th}, S'_\delta) = (\mathtt{i}', \mathtt{c})$.

$(\delta_1 \parallel \delta_2)$**:**

$\Rightarrow$:

By definition of **comp** and Lemma 1, the first action in $\sigma = S'_\delta - S_\delta$ in thread *th* can only be $\mathit{spawn}(\mathit{th}, \mathtt{c}, \mathtt{i}+1, \mathtt{i}_1+1)$ where $\mathtt{i}_1$ is an integer defined in the compilation. By $\mathtt{ax}_{\mathit{common}}$ the reached situation $S_1 = \mathit{do}(\mathit{spawn}(\mathit{th}, \mathtt{c}, \mathtt{i}_2+1, \mathtt{i}+1, \mathtt{i}_1+1), S_\delta)$ is such that two new threads exist which we here denote $\mathit{th}_1, \mathit{th}_2$, one in state $\mathtt{i}+1$ and another in $\mathtt{i}_1+1$ of the same context.

We consider both cases of the lemma separately.

1. By definition of $exec_{th}$ and Lemma 1 it follows that S'_δ satisfies both $\mathcal{D}_{\delta_1,\mathtt{i},\mathtt{c},\mathtt{i}_1} \models exec_{th_1}(S_1, S'_\delta)$ and $\mathcal{D}_{\delta_2,\mathtt{i}_1+1,\mathtt{c},\mathtt{i}_2} \models exec_{th_2}(S_1, S'_\delta)$, and also $filter(S_1, \mathcal{D}) = S$. Hence, by induction hypothesis, there are programs δ'_1, δ'_2 and respective continuations $\widehat{S}'_1$ and $\widehat{S}'_2$ of S in $\widehat{\mathcal{D}}$ such that $S'_\delta \mathrel{\widehat{=}}_{th_1} \widehat{S}'_1$, $S'_\delta \mathrel{\widehat{=}}_{th_2} \widehat{S}'_2$, $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_1, S, \delta'_1, \widehat{S}'_1)$, and $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_2, S, \delta'_2, \widehat{S}'_2)$.

   It is easy to show from the definition of $\widehat{=}_{th}$ that all unmarked actions of $\widehat{S}'_1$ then also appear marked in $\widehat{S}'_2$ in the same order, and vice versa, and that all other marked actions are shared. Hence, $pure(\widehat{S}'_1) = pure(\widehat{S}'_2)$. Construct $\widehat{S}'$ as follows: take $\widehat{S}'_1$ and unmark all actions that appear unmarked in $\widehat{S}'_2$. It is easy to see from the definition of $\widehat{=}_{th}$ that $S'_\delta \mathrel{\widehat{=}}_{th} \widehat{S}'$. Similarly, from the definition of $\widehat{Trans}$ it follows that $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_1 \parallel \delta_2, S, \delta'_1 \parallel \delta'_2, \widehat{S}')$.

2. If further $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models state(th, S'_\delta) = (\mathtt{i}', \mathtt{c})$, then by definition of **comp** the last action of S'_δ in thread th can only be $join(th, \mathtt{c}, \mathtt{i}_2 + 2)$, which in turn can only occur after executing $finalize(th_1)$ and $finalize(th_2)$. These actions are only possible in a situation s where $\mathcal{D}_{\delta_1,\mathtt{i},\mathtt{c},\mathtt{i}_1} \models state(th_1, s) = (\mathtt{i}_1, \mathtt{c})$ and $\mathcal{D}_{\delta_2,\mathtt{i}_1+1,\mathtt{c},\mathtt{i}_2} \models state(th_2, s) = (\mathtt{i}_2, \mathtt{c})$ respectively. Hence, by induction hypothesis also $\widehat{\mathcal{D}} \models Final(\delta'_1, \widehat{S}'_1)$ and $\widehat{\mathcal{D}} \models Final(\delta'_1, \widehat{S}'_1)$. Hence, by definition of $Trans$, $\widehat{\mathcal{D}} \models Final(\delta'_1 \parallel \delta'_2, \widehat{S}')$.

$\Leftarrow$:

We again consider the two parts:

1. By the definition of $\widehat{Trans}^*$ the situation $\widehat{S}'$ can be transformed into two situations $\widehat{S}'_1$ and $\widehat{S}'_2$ such that $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_1, S, \delta'_1, \widehat{S}'_1)$ and $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_2, S, \delta'_2, \widehat{S}'_2)$ for some δ'_1, δ'_2, by marking the actions executed in the respectively other sub-program.

   We hence get by induction hypothesis that for any situation S^*_δ which satisfies $filter(S^*_\delta, \mathcal{D}) = S$ and $\mathcal{D}_{\delta_1,\mathtt{i},\mathtt{c},\mathtt{i}_1} \models Thread(th_1) \wedge state(th_1, S^*_\delta) = (\mathtt{i}+1, \mathtt{c})$ and $\mathcal{D}_{\delta_2,\mathtt{i}_1+1,\mathtt{c},\mathtt{i}_2} \models Thread(th_2) \wedge state(th_2, S^*_\delta) = (\mathtt{i}_1 + 1, \mathtt{c})$, for any $\mathtt{i}, \mathtt{i}_1$, that there exist continuations $S'_{\delta_1}, S'_{\delta_2}$ of S^*_δ such that $S'_{\delta_1} \mathrel{\widehat{=}}_{th} \widehat{S}'_1$ and $S'_{\delta_2} \mathrel{\widehat{=}}_{th} \widehat{S}'_2$, and $\mathcal{D}_{\delta_1,\mathtt{i},\mathtt{c},\mathtt{i}_1} \models exec_{th_1}(S^*_\delta, S'_{\delta_1})$ and $\mathcal{D}_{\delta_2,\mathtt{i}_1+1,\mathtt{c},\mathtt{i}_2} \models exec_{th_2}(S^*_\delta, S'_{\delta_2})$.

   It is easy to show that $S_1 = do(spawn(th, \mathtt{c}, \mathtt{i}_2 + 1, \mathtt{i} + 1, \mathtt{i}_1 + 1), S_\delta)$ satisfies the above conditions for any S_δ satisfying the conditions in the lemma. By definition of **comp**, the action $spawn(th, \mathtt{c}, \mathtt{i}_2 + 1, \mathtt{i} + 1, \mathtt{i}_1 + 1)$ is executable in S_δ in $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$, following the assumptions about this situation.

   Construct a new situation S''_δ as follows: Iterate concurrently through S'_1 and S'_2 (remember they are essentially the same, ignoring the marking): if the next unmarked action is in S'_1 then move all actions S'_{δ_1} up to and including this action to S''_δ. Otherwise, do the same using S'_{δ_2}. Since the domain actions are the same and have not changed compared to S'_{δ_1} and S'_{δ_2} it follows that $S'_\delta = do(S''_\delta - S_\delta, do(spawn(th, \mathtt{c}, \mathtt{i}_2 + 1, \mathtt{i} + 1, \mathtt{i}_1 + 1), S_\delta))$ satisfies: $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th_1}(S_\delta, S'_\delta)$ and $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th_2}(S_\delta, S'_\delta)$, and hence, by definition of $exec_{th}$ also $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th}(S_\delta, S'_\delta)$. Further, by construction, $S'_\delta \mathrel{\widehat{=}}_{th} \widehat{S}'$ (note that *spawn* is not in $\mathcal{A}$).

2. If further $\widehat{\mathcal{D}} \models Final(\delta', \widehat{S}')$, then, by definition of *Final* for concurrency, also $\widehat{\mathcal{D}} \models Final(\delta'_1, \widehat{S}'_1)$ and $\widehat{\mathcal{D}} \models Final(\delta'_2, \widehat{S}'_2)$. It hence follows by induction hypothesis that also $\mathcal{D}_{\delta_1,\mathtt{i},\mathtt{c},\mathtt{i}_1} \models state(th, S'_{\delta_1}) = (\mathtt{i}_1, \mathtt{c})$ and $\mathcal{D}_{\delta_2,\mathtt{i}_1+1,\mathtt{c},\mathtt{i}_2} \models state(th, S'_{\delta_2}) = (\mathtt{i}_2, \mathtt{c})$, with $\mathtt{i}_1, \mathtt{i}_2$ as defined in the compilation.

   Let S''_δ be as constructed above, but if after iterating over S'_1/S'_2 a sequence of (pseudo-)actions σ_1 remains in S'_{δ_1} in th_1 and/or σ_2 in S'_{δ_2} in th_2, then create $S'''_\delta = do(\sigma_1 \cdot [finalize(th_1), backtrack(th_1)] \cdot \sigma_2 \cdot [finalize(th_2), join(th, \mathtt{c}, \mathtt{i}_2 + 2)], S''_\delta)$. This situation is such that $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th}(S_\delta, S'''_\delta)$ by the above, definition of **comp** for concurrency, and $\mathtt{ax}_{backtrack}$, and it satisfies: $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models state(th, S'''_\delta) = (\mathtt{i}', \mathtt{c})$.

$(\delta_1|\delta_2)$:

$\Rightarrow$:

By Lemma 1 and definition of **comp**, the first action in thread th in $\sigma = S'_\delta - S_\delta$ is either $noop(th, \mathtt{i}, \mathtt{i} + 1, \mathtt{c})$ or $noop(th, \mathtt{i}, \mathtt{i}_1 + 1, \mathtt{c})$. We here only show the thesis for the first case, as the second follows analogously. The resulting situation S_1 satisfies $\mathcal{D}_{\delta_1,\mathtt{i},\mathtt{c},\mathtt{i}_1} \models state(th, S_1) = (\mathtt{i} + 1, \mathtt{c})$.

1. First consider the case where $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \not\models state(th, S'_\delta) = (\mathtt{i}', \mathtt{c})$. Then by Lemma 1 and definition of **comp**, S'_δ is a continuation of S_1 such that $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th}(S_1, S'_\delta)$ and hence, by induction hypothesis, then there is a program δ'_1 and a continuation $\widehat{S}'$ of S in $\widehat{\mathcal{D}}$ such that $S'_\delta \mathrel{\widehat{=}}_{th} \widehat{S}'$ and $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_1, S, \delta'_1, \widehat{S}')$. Hence, by definition of *Trans* also $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_1|\delta_2, S, \delta'_1, \widehat{S}')$.
2. Otherwise, if $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models state(th, S'_\delta) = (\mathtt{i}', \mathtt{c})$, then the last action in thread th in S'_δ can only be $noop(th, \mathtt{i}_1, \mathtt{i}_2 + 1, \mathtt{c})$, by definition of **comp** and Lemma 1. This action is only possible if for S'_{δ_1} such that $S'_\delta = do(noop(th, \mathtt{i}_1, \mathtt{i}_2 + 1, \mathtt{c}), S'_{\delta_1})$ we have that $\mathcal{D}_{\delta_1,\mathtt{i},\mathtt{c},\mathtt{i}_1} \models state(th, S'_{\delta_1}) = (\mathtt{i}_1, \mathtt{c})$. Hence, by induction hypothesis and definition of $\widehat{=}_{th}$, $\widehat{\mathcal{D}} \models Final(\delta'_1, \widehat{S}')$.

$\Leftarrow$:

1. By definition of *Trans* either $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_1, S, \delta', \widehat{S}')$ or $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_2, S, \delta', \widehat{S}')$. Without loss of generality we here assume the first case.

   Then, by induction hypothesis, for any S_{δ_1} such that $filter(S_{\delta_1}, \mathcal{D}) = S$ and $\mathcal{D}_{\delta_1,\mathtt{i}+1,\mathtt{c},\mathtt{i}_1} \models Thread(th, S_{\delta_1}) \wedge state(th, S_{\delta_1}) = (\mathtt{i}+1, \mathtt{c})$ there is a continuation S'_{δ_1} such that $S'_{\delta_1} \mathrel{\widehat{=}}_{th} \widehat{S}'$ and $\mathcal{D}_{\delta_1,\mathtt{i}+1,\mathtt{c},\mathtt{i}_1} \models exec_{th}(S_{\delta_1}, S'_{\delta_1})$. Let $\sigma = S'_{\delta_1} - S_{\delta_1}$.

   The situation $S^*_{\delta_1} = do(noop(th, \mathtt{i}, \mathtt{i}+1, \mathtt{c}), S_\delta)$ satisfies above conditions for any situation S_δ which is as described in the lemma. Since $noop(th, \mathtt{i}, \mathtt{i}+1, \mathtt{c})$ is executable in S_δ, by definition of **comp** we get: $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th}(S_\delta, do(\sigma, S^*_\delta))$.

2. if $\widehat{\mathcal{D}} \models Final(\delta', \widehat{S}')$, then, by induction hypothesis, $\mathcal{D}_{\delta_1,\mathtt{i}+1,\mathtt{c},\mathtt{i}_1} \models state(th, S'_{\delta_1}) = (\mathtt{i}_1, \mathtt{c})$. Hence, $S'_\delta = do(noop(th, \mathtt{i}_1, \mathtt{i}_2{+}1, \mathtt{c}), do(\sigma, S^*_\delta))$ is such that both $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th}(S_\delta, S'_\delta)$ and $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models state(th, S'_\delta) = (\mathtt{i}', \mathtt{c})$, and obviously still $S'_\delta \mathrel{\widehat{=}}_{th} \widehat{S}'$.

(while ϕ do δ_1):

$\Rightarrow$:

By Lemma 1 and definition of **comp** the first action of $\sigma = S'_\delta - S_\delta$ in thread *th* can only be $noop(th, \mathtt{i}, \mathtt{i}+1, \mathtt{c})$. Thereafter, if (a) $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \not\models \phi(do(noop(th, \mathtt{i}+1, \mathtt{i}+2, \mathtt{c}), S_\delta))$ or $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models blockeds(th, sp(th, S_\delta), \mathtt{i}+2, \mathtt{c}, S_\delta)$ then the next action in *th* is $test(th, \mathtt{i}+1, \mathtt{i}_1+1, \mathtt{c})$. Otherwise, (b) it is $test(th, \mathtt{i}+1, \mathtt{i}+2, \mathtt{c})$.

We show the thesis for this case by induction over the cycles of the loop (we refer to this as the inner induction). There are two base cases: case (a) above, and case (b) with zero cycles – intuitively the situation "ends in δ_1". For the inner induction step we assume (b) and that one cycle less remains. Such in induction is possible, since we assume finite situation terms.

(a) In this case we immediately get that $\mathcal{D} \models \neg\phi(S)$ and hence $\mathcal{D} \models Final(\textbf{while } \phi \textbf{ do } \delta_1, \widehat{S}')$ for any $\widehat{S}'$ not mentioning any unmarked actions (this also implies, trivially $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta, S, \delta, \widehat{S}')$). Since, $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ does not admit any further actions in *th* and the above are not domain actions, we have $S'_\delta \mathrel{\widehat{=}}_{th} \widehat{S}'$ for any such situation.

(b) zero cycles left: Let $S^*_\delta = do(test(th, \mathtt{i}{+}1, \mathtt{i}{+}2, \mathtt{c}), do(noop(th, \mathtt{i}, \mathtt{i}+1, \mathtt{c}), S_\delta))$. Then $filter(S^*_\delta, \mathcal{D}) = S$, since *test* and *noop* are not domain actions, and, by construction, $\mathcal{D}_{\delta_1,\mathtt{i}+2,\mathtt{c},\mathtt{i}_1} \models Thread(th, S^*_\delta) \wedge state(th, S^*_\delta) = (\mathtt{i}+2, \mathtt{c})$. Let $\sigma' = S'_\delta - S^*_\delta$. Then, S'_δ is a continuation of S^*_δ such that $\mathcal{D}_{\delta_1,\mathtt{i}+2,\mathtt{c},\mathtt{i}_1} \models exec_{th}(S^*_\delta, S'_\delta)$. Hence, by induction hypothesis, there is a program δ'_1 and a continuation $\widehat{S}'_1$ of S in $\widehat{\mathcal{D}}$ such that $S'_\delta \mathrel{\widehat{=}}_{th} \widehat{S}'_1$ and $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta, S, \delta'_1, \widehat{S}'_1)$. Hence, by definition of *Trans* and the above assumption that $\mathcal{D} \models \phi(S)$, it follows that $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\textbf{while } \phi \textbf{ do } \delta_1, S, \delta_1'; \textbf{while } \phi \textbf{ do } \delta_1, \widehat{S}_1')$.

By assumption, $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \not\models state(th, S_\delta') = (\mathtt{i}', \mathtt{c})$, hence, the second implication holds trivially in this case.

(b) induction step: In the inner induction case, there is a prefix σ_1 of $S_\delta' - do(test(th, \mathtt{i}+1, \mathtt{i}+2, \mathtt{c}), do(noop(th, \mathtt{i}, \mathtt{i}+1, \mathtt{c}), S_\delta))$ such that for $S_{\delta_1}' = do(\sigma_1, do(test(th, \mathtt{i}+1, \mathtt{i}+2, \mathtt{c}), do(noop(th, \mathtt{i}, \mathtt{i}+1, \mathtt{c}), S_\delta)))$ we have $\mathcal{D}_{\delta_1,\mathtt{i}+2,\mathtt{c},\mathtt{i}_1} \models state(th, S_{\delta_1}') = (\mathtt{i}_1, \mathtt{c})$. Hence, by outer induction hypothesis there is a program δ_1' and a continuation $\widehat{S}_1'$ of S in $\widehat{\mathcal{D}}$ such that $S_{\delta_1}' \mathrel{\widehat{=}_{th}} \widehat{S}_1'$ and $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_1, S, \delta_1', \widehat{S}_1')$ and $\widehat{\mathcal{D}} \models Final(\delta_1', \widehat{S}_1')$.

Hence, we get by definition of $\widehat{Trans}$ that

$$\widehat{\mathcal{D}} \models \widehat{Trans}^*(\textbf{while } \phi \textbf{ do } \delta_1, S, \delta_1'; \textbf{while } \phi \textbf{ do } \delta_1, S_1')$$

and since $\widehat{\mathcal{D}} \models Final(\delta_1', \widehat{S}_1')$ further

$$\begin{aligned}\mathcal{D} \models \quad & \widehat{Trans}^*(\textbf{while } \phi \textbf{ do } \delta_1, S, \delta_1'; \textbf{while } \phi \textbf{ do } \delta_1, S_1') \equiv \\ & \widehat{Trans}^*(\textbf{while } \phi \textbf{ do } \delta_1, S, \textbf{while } \phi \textbf{ do } \delta_1, S_1').\end{aligned}$$

Applying inner induction hypothesis we get that there is also a program δ' and a continuation $\widehat{S}'$ of S in $\widehat{\mathcal{D}}$ such that $S_\delta' \mathrel{\widehat{=}_{th}} \widehat{S}'$ and $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta, S, \delta', \widehat{S}')$. Further, again from inner induction hypothesis (in particular if case (a) was used to terminate the induction) if $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models state(th, S_\delta') = (\mathtt{i}', \mathtt{c})$, then also $\widehat{\mathcal{D}} \models Final(\delta', \widehat{S}')$.

$\Leftarrow$:

By definition of $\widehat{Trans}$, there is either $\widehat{S}'$ and δ' such that

$$\widehat{\mathcal{D}} \models \widehat{Trans}^*(\textbf{while } \phi \textbf{ do } \delta_1, S, \delta_1'; \textbf{while } \phi \textbf{ do } \delta_1, \widehat{S}')$$

or $\widehat{\mathcal{D}} \models Final(\textbf{while } \phi \textbf{ do } \delta_1, S)$.

In the second case, by definition $\mathcal{D} \models \neg\phi(S)$ and hence, as above, $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th}(S_\delta, do([noop(th, \mathtt{i}, \mathtt{i}+1, \mathtt{c}), test(th, \mathtt{i}+1, \mathtt{i}_1+1, \mathtt{c})], S_\delta))$, and by definition of **comp** (and **test** in particular), $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models state(th, do([noop(th, \mathtt{i}, \mathtt{i}+1, \mathtt{c}), test(th, \mathtt{i}+1, \mathtt{i}_1+1, \mathtt{c})], S_\delta)) = (\mathtt{i}', \mathtt{c})$, since $\mathtt{i}' = \mathtt{i}_1 + 1$.

The first case is again shown by induction over the cycles of the loop. We will again refer to this induction as the inner induction.

1. In the base case, $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\textbf{while } \phi \textbf{ do } \delta_1, S, \delta_1'; \textbf{while } \phi \textbf{ do } \delta_1, \widehat{S}')$ implies that $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_1, S, \delta_1', \widehat{S}')$. Hence, by outer induction hypothesis we get that for any thread name th, integer $\mathtt{i}+2$, and situation S_{δ_1} in $\mathcal{D}_{\delta_1,\mathtt{i}+2,\mathtt{c},\mathtt{i}_1}$ such that $filter(S_{\delta_1}, \mathcal{D}) = S$ and $\mathcal{D}_{\delta_1,\mathtt{i}+2,\mathtt{c},\mathtt{i}_1} \models$

$\mathit{Thread}(th, S_{\delta_1}) \wedge \mathit{state}(th, S_{\delta_1}) = (\mathtt{i}+2, \mathtt{c})$, there exists a continuation S'_{δ_1} of S_{δ_1} such that $S'_{\delta_1} \mathrel{\widehat{=}}_{th} \widehat{S}'$ and $\mathcal{D}_{\delta_1,\mathtt{i}+2,\mathtt{c},\mathtt{i}_1} \models \mathit{exec}_{th}(S_{\delta_1}, S'_{\delta_1})$ and if $\widehat{\mathcal{D}} \models \mathit{Final}(\delta'_1, \widehat{S}')$, then also $\mathcal{D}_{\delta_1,\mathtt{i}+2,\mathtt{c},\mathtt{i}_1} \models \mathit{state}(th, S'_{\delta_1}) = (\mathtt{i}_1, \mathtt{c})$. For any S_δ as described in the lemma, we can hence choose $S'_\delta = do(S'_{\delta_1} - S_{\delta_1}, do([\mathit{noop}(th, \mathtt{i}, \mathtt{i}+1, \mathtt{c}), \mathit{test}(th, \mathtt{i}+1, \mathtt{i}+2, \mathtt{c})], S_\delta))$.

If $\widehat{\mathcal{D}} \models \mathit{Final}(\delta', \widehat{S}')$, which is the case when $\widehat{\mathcal{D}} \models \mathit{Final}(\delta', \widehat{S}')$ and $\widehat{\mathcal{D}} \models \neg\phi(\widehat{S}')$, then we extend S'_δ by the action sequence $[\mathit{noop}(th, \mathtt{i}_1, \mathtt{i}+1, \mathtt{c}), \mathit{test}(th, \mathtt{i}+1, \mathtt{i}_1+1, \mathtt{c})]$, which is executable in any situation s for which $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{state}(th, s) = (\mathtt{i}_1, \mathtt{c}) \wedge \neg\phi(s)$. The former is provided by the above application of (outer) induction hypothesis, and the latter is true, since $S'_\delta \mathrel{\widehat{=}}_{th} \widehat{S}'$, which in particular means that these two situations contain the same domain actions (in the same order, ignoring marking), since pseudo-actions by construction do not affect domain fluents, and since ϕ cannot mention any bookkeeping fluents.

2. In the inner induction case, there is a shortest prefix σ of $\widehat{S}' - S$ such that $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta_1, S, \delta''_1, do(\sigma, S))$ and $\widehat{\mathcal{D}} \models \mathit{Final}(\delta''_1, do(\sigma, S))$. Then, by outer induction hypothesis, as above, for any thread name th, integer $\mathtt{i}+2$, and situation S_{δ_1} in $\mathcal{D}_{\delta_1,\mathtt{i}+2,\mathtt{c},\mathtt{i}_1}$ such that $\mathit{filter}(S_{\delta_1}, \mathcal{D}) = S$ and $\mathcal{D}_{\delta_1,\mathtt{i}+2,\mathtt{c},\mathtt{i}_1} \models \mathit{Thread}(th, S_{\delta_1}) \wedge \mathit{state}(th, S_{\delta_1}) = (\mathtt{i}+2, \mathtt{c})$, there exists a continuation S'_{δ_1} of S_{δ_1} such that $S'_{\delta_1} \mathrel{\widehat{=}}_{th} \widehat{S}'$ and $\mathcal{D}_{\delta_1,\mathtt{i}+2,\mathtt{c},\mathtt{i}_1} \models \mathit{exec}_{th}(S_{\delta_1}, S'_{\delta_1}) \wedge \mathit{state}(th, S'_{\delta_1}) = (\mathtt{i}_1, \mathtt{c})$. Consider the situation $S''_\delta = do(\mathit{noop}(th, \mathtt{i}_1, \mathtt{i}+1, \mathtt{c}), do(S'_{\delta_1} - S_{\delta_1}, do([\mathit{noop}(th, \mathtt{i}, \mathtt{i}+1, \mathtt{c}), \mathit{test}(th, \mathtt{i}+1, \mathtt{i}+2, \mathtt{c})], S_\delta))$. This situation is executable in $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ as argued above regarding the 'Final' case, and it is such that $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{state}(th, S''_\delta) = (\mathtt{i}+1, \mathtt{c})$. Hence, by inner induction hypothesis there is a continuation S'_δ of this situation such that $S'_\delta \mathrel{\widehat{=}}_{th} \widehat{S}'$ and $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{exec}_{th}(S_\delta, S'_\delta)$. And if $\widehat{\mathcal{D}} \models \mathit{Final}(\delta'_1; \mathbf{while}\ \phi\ \mathbf{do}\ \delta_1, \widehat{S}')$ then also $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{state}(th, S'_\delta) = (\mathtt{i}', \mathtt{c})$. Hence the thesis.

$P(t_1, \ldots, t_n)$ **where** $P(x_1, \ldots, x_n)$ **is a procedure:**

Following the steps of the compilation, we assume that any procedure $P'(x_1, \ldots, x_n)$ has been compiled into $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ and the returned integer was $\mathtt{i}_{P'}$. Note also that we do not consider nested procedure definitions. [4]

The treatment of actual procedure parameters is done the same as for program variables. Effectively, when a procedure is called, the parameters are evaluated and stored in the *map* fluent. Each time an action or

[4]While it would be more descriptive to refer to the resulting theory, here and above, as $\mathcal{D}_{\{P_1(\vec{x_1},\delta_{P_1});\ldots;P_n(\vec{x_n},\delta_{P_n})\};\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$, we choose to still refer to it as $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ for parsimony and readability, and understand that all referred procedures have been compiled as well. Note that this is not a problem, since the compilation is purely syntactic and in particular, the order of compiling the procedures does not depend on their semantic dependencies.

condition refers to the formal parameters, reference is made to this map instead (more details can be found in the proof for programs with program variables below).

$\Rightarrow$:

The thesis is shown by induction over the recursions of P (we call this induction again the inner induction, to distinguish it from the outer induction over the structure of programs).

In the base case, the currently executing procedure does not actually call other procedures nor itself. This case follows immediately by (outer) induction hypothesis (it is just a regular program without procedures).

In the induction case, we are assuming that the property holds for situation terms that contain n procedure calls, and show it for $n+1$. By Lemma 1 the only action possible in $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ in th in S_δ is $call(th, P, \mathtt{i}+1, \mathtt{c})$. By definition of $\mathtt{ax}_{common}$, this action stores the return address on the stack and establishes the state $(0, P)$. Hence, S'_δ is such that $\mathcal{D}_{\delta_P,0,P,\mathtt{i}_P} \models exec_{th}(do(call(th, P, \mathtt{i}+1, \mathtt{c}), S_\delta), S'_\delta)$. Hence, by inner induction hypothesis there is a program δ' and a continuation $\widehat{S}'$ of S in $\widehat{\mathcal{D}}$ such that $S'_\delta \mathrel{\widehat{=}}_{th} \widehat{S}'$ and $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_{P(t_1,\ldots,t_n)}, S, \delta', \widehat{S}')$. By definition of *Trans* we then also get $\widehat{\mathcal{D}} \models \widehat{Trans}^*(P(t_1,\ldots,t_n), S, \delta', \widehat{S}')$. Further, if $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models state(th, S'_\delta) = (\mathtt{i}', \mathtt{c})$ then the last action in thread th in S'_δ can only be $return(th)$, due to the definition of **comp**. This action, is only possible in situations s where $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models state(th, s) = (\mathtt{i}_P, \mathtt{c})$ (cf. Step 3 of the compilation). Since then also $\mathcal{D}_{\delta_P,0,P,\mathtt{i}_P} \models state(th, S'_\delta) = (\mathtt{i}_P, P)$ we get again by inner induction hypothesis that also $\widehat{\mathcal{D}} \models Final(\delta', \widehat{S}')$.

$\Leftarrow$:

We again show this, by induction over the number of recursive procedure calls. The base case, where the currently executing program does not actually mention procedure calls, is again trivially given by outer induction hypothesis.

In the induction step we assume that there is a program δ' and a continuation $\widehat{S}'$ of S in $\widehat{\mathcal{D}}$ such that $\widehat{\mathcal{D}} \models \widehat{Trans}^*(P(t_1,\ldots,t_n), S, \delta', \widehat{S}')$. By definition of *Trans* it is hence the case that $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_{P(t_1,\ldots,t_n)}, S, \delta', \widehat{S}')$. Since there hence remain one less procedure call in the trail of configurations from $\langle \delta_{P(t_1,\ldots,t_n)}, S \rangle$ to $\langle \delta', \widehat{S}' \rangle$, we get by inner induction hypothesis that for any thread name th, integer $\mathtt{i}$, and situation S_{δ_P} in $\mathcal{D}_{\delta_P,0,P,\mathtt{i}_P}$ such that $filter(S_{\delta_P}, \mathcal{D}) = S$ and $\mathcal{D}_{\delta_P,0,P,\mathtt{i}_P} \models Thread(th, S_{\delta_P}) \wedge state(th, S_{\delta_P}) = (0, P)$, there exists a continuation S'_{δ_P} of S_{δ_P} such that $S'_{\delta_P} \mathrel{\widehat{=}}_{th} \widehat{S}'$ and $\mathcal{D}_{\delta_P,0,P,\mathtt{i}_P} \models exec_{th}(S_{\delta_P}, S'_{\delta_P})$.

The situation $do(call(th, P, \mathtt{i}+1, \mathtt{c}), S_\delta)$ satisfies this condition for any S_δ as described in the lemma, and by definition of **comp** $call(th, P, \mathtt{i}+1, \mathtt{c})$ is possible in any such S_δ. Hence, $S'_\delta = do(S'_{\delta_P} - S_{\delta_P}, do(call(th, P, \mathtt{i}+1, \mathtt{c}), S_\delta))$ is such that $S'_\delta \mathrel{\widehat{=}}_{th} \widehat{S}'$ and $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th}(S_\delta, S'_\delta)$.

Further, also by inner induction hypothesis, if $\widehat{\mathcal{D}} \models \mathit{Final}(\delta', \widehat{S}')$, then the mentioned situation S'_{δ_P} is such that $\mathcal{D}_{\delta_P,0,P,\mathtt{i}_P} \models \mathit{state}(\mathit{th}, S'_{\delta_P}) = (\mathtt{i}_P, \mathtt{c})$. In that case, we can append the action $\mathit{return}(\mathit{th})$ to it, which is executable in that situation in $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$, due to Step 3 of the compilation, and which has the effect of establishing the state denoted by the highest stack position. Following the definition of $\mathtt{ax}_{\mathrm{procs}}$, the action $\mathit{call}(\mathit{th}, P, \mathtt{i}+1, \mathtt{c})$ had the effect of establishing the value $(\mathtt{i}+1, \mathtt{c})$ for this. Hence, $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{state}(\mathit{th}, \mathit{do}(\mathit{return}(\mathit{th}), S'_{\delta})) = (\mathtt{i}+1, \mathtt{c})$, i.e. the thesis, since $\mathtt{i}' = \mathtt{i}+1$.

We only outline the remaining cases, since they are all quite similar to one of the above.

(δ'^*):

This case follows in close analogy to the case of **while**–loops, just replacing the *test* actions by *noop* actions.

(if ϕ then δ_1 else δ_2):

This case follows in close analogy to the case of non-deterministic choice of sub-programs $(a|b)$, but replacing the initial *noop* actions with *test* actions.

($\delta^{||}$):

This case follows by induction over the number of concurrent iterations, where the base case is that of not actually executing δ even once, and the induction step is provided by the (outer) induction step for normal concurrency.

($\delta_1 \rangle\!\rangle \delta_2$):

$\Rightarrow$:

This case follows in analogy to the normal concurrency case $(\delta_1 \parallel \delta_2)$. If δ_1 is executed until completion before any actions are executed in δ_2, the case is just like for $\parallel$. On the other hand, if actions are performed in th_2 before then, then by definition of **comp** and Step 4 of the compilation, no actions were possible in that situation in th_1. From the latter, by induction hypothesis for the $\Leftarrow$ direction, it follows that also there don't exist $\delta'_1, \widehat{S}'$ such that $\widehat{S}' - S$ contains unmarked actions and $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta_1, S, \delta'_1, \widehat{S}')$. Hence, by definition of *Trans*, there is a program $\delta'_2, \widehat{S}'$ such that $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta_1 \rangle\!\rangle \delta_2, S, \delta_1 \rangle\!\rangle \delta'_2, \widehat{S}')$.

The remainder is again analogous to the $\parallel$ case.

$\Leftarrow$:

Again, the case where the first transition is over δ_1 is analogous to the $\parallel$ case. If δ_2 executes, then $\mathcal{D} \models (\nexists \delta'_1, S').\mathit{Trans}(\delta_1, S, \delta'_1, S')$, by definition of *Trans*. Hence, again by induction hypothesis of the $\Rightarrow$ direction, there is no executable continuation whose first domain action is in th_1, hence allowing actions in th_2 to execute (either directly, or after executing a number of pseudo-actions first, followed by the $backtrack(th)$ action). The remaining reasoning is as in the $\|$ case.

$\Box$

Proof of Lemma 2 for programs with the π construct:
Our solution for treating programs with the π construct and hence program variables is similar to Skolemization. In ConGolog, $\pi(v, \delta)$ is interpreted as the execution of the program δ where all occurrences of the constant v are substituted by a new existentially quantified variables. This method is not possible in our compilation, since π can appear in loops, whose body we only want to consider once during compilation. Instead, we replace these existentially quantified variables with functional fluents. Similar to Skolemization, these functions need to be relative to the context the variable appears in, namely the stack position (for πs in recursive procedures) and the thread (for πs inside of the $\delta^{\|}$ construct, or inside a procedure which is called in two concurrent threads).

By definition of **comp**, a π construct causes the execution of the $pi(th, v, x, \mathtt{c}, \mathtt{i})$ action, whose effect due to $\mathtt{ax}_\pi$ is that $map(th, p, v) = x$, where p is the current value of the stack pointer. Note that x is a free parameter – not mentioned in the preconditions. It can hence be any object. Since the π action is the only action affecting the map fluent, this value in map pertains until the same construct is visited again. Note that we disallow redefinitions of program variables, i.e. for instance $\pi(v, A(v); \pi(v, B(v)))$ is not allowed in the original program. This is not a restriction, since any such program could be transformed to be free of such redefinitions, by simply renaming the program variables.

The thesis then follows by inspection and induction over the nesting depth of program variables. Recall that program variables can only occur in places of action parameters and in conditions. In both, as provided by **comp** for primitive actions and procedure calls and by the **test** and **rtest** functions for conditions, these occurrences are forced to be equal to the values stored in map.

The base case of the induction is for the case of a program without program variables and is immediately provided by the above proof for such programs.

For the induction step, let δ be a program with n program variables. Then by above considerations, induction hypothesis, and the fact that pi is a pseudo-action and hence filtered out by $filter$: for any ground situation term S, for program $\pi(v, \delta)$, there is an object O, a situation $\widehat{S}'$, and a program δ' such that $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta|_{v/o}, S, \delta', \widehat{S}')$ if and only if there is a sequence of ground action terms σ in $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ such that $S'_\delta = do(\sigma, do(pi([0], v, O, main, 1), S_\delta))$ such that $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th}(S_\delta, S'_\delta)$ and $S'_\delta \mathrel{\widehat{=}}_{th} \widehat{S}'$ for any situation term S_δ as described in the lemma. And further, if $\widehat{\mathcal{D}} \models Final(\delta', \widehat{S}')$ then also $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models state(th, S'_\delta) = (\mathtt{i}', \mathtt{c})$. Hence the thesis holds for programs with $n+1$ program variables and hence, by induction, for any program with any number of program variables. $\Box$

C.3 Proof of the Theorem

The theorem follows from Lemma 2 for the special case of $S = S_0$, $th = [0]$, $\mathtt{i} = 0$, and $\mathtt{c} = main$: Since $S'_{\mathcal{P}}$ cannot mention any actions not in $[0]$ or its descendants, we have that $exec_{th}(S_0, S'_{\mathcal{P}})$ implies $executable(S'_{\mathcal{P}})$. Further, by definition of $\widehat{=}_{th}$ also $\widehat{S}'$ cannot mention any marked actions, and hence, following Lemma 2

$$\mathcal{D}_{\mathcal{P}} \models executable(S'_{\mathcal{P}}) \wedge state([0], S'_{\mathcal{P}}) = (\mathtt{i}_{main}, main)$$

iff there exists a program δ' such that for $S' = filter(S'_{\mathcal{P}}, \mathcal{D})$ we have that

$$\mathcal{D} \models Trans^*(\mathcal{P}, S_0, \delta', S') \wedge Final(\delta', S')$$

which by definition is equivalent to $\mathcal{D} \models Do_2(\mathcal{P}, S_0, S')$. □

D Proofs of Theorems 2 and 3

These two theorems both rely on the fact that the compilation visits each program construct and logical connective mentioned in a condition exactly once, and that each time a constantly bounded number of additional axioms are introduced, each of size at most n. This is shown in the following lemma.

Lemma 3. Let δ be any ConGolog program of size n, whose set of free program variables $\mathtt{e}$ has size k, and let $\mathtt{i}$ be any integer, and $\mathtt{c}$ any procedure name. Let further be l the maximal cardinality of formal parameters of all the procedures that are called in the program. The invocation of $\mathbf{comp}(\delta, \mathtt{i}, \mathtt{e}, \mathtt{c})$ makes at most $n - 1$ recursive calls to **comp**, and each invocation adds at most a constant number of sentences to the set of returned sentences, each of size at most $O(max(k, l))$.

Proof: By inspection. It is easy to see from the definition of **comp** that the sum of the sizes of all sub-programs appearing as parameters in the recursive invocations is less or equal to $n - 1$. Further, the cardinality of the set of sentences added to the eventually returned set `ax` is bound by a constant in all cases, since all auxiliary functions return only a constant number of sentences. The size of each such sentence is in $O(max(k, l))$, since the only parametrized connective appearing in the definitions is $\bigwedge_{\Psi}$, where Ψ is a set of cardinality $\leq max(l, k)$. □

Proof of Theorems 2 and 3: Let as before be $\mathcal{P} = \{P_1(\vec{t_1}, \delta_{P_1}); \ldots; P_n(\vec{t_n}, \delta_{P_n}); \delta_{main}\}$, and let m be the size of $\mathcal{P}$. Then the sum of the number of occurrences of π constructs k, and the maximal cardinality of formal procedure parameters l must be $< m$. Hence, by Lemma 3, the set `AX` (see Step 2, p. 12) is of size $O(m^2)$. Since both the time and the space complexity of each remaining step of the compilation is linear in the size of `AX`, we get the thesis. □