Proceedings, Eleventh International Conference on Principles of Knowledge Representation and Reasoning (2008)

ConGolog, Sin Trans: Compiling ConGolog into Basic Action Theories for Planning and Beyond

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 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 second-order axioms altogether. 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.

1 Introduction

ConGolog (De Giacomo, Lespérance, & Levesque 2000) is a logical programming language for specifying high-level agent control, that is defined in the situation calculus. ConGolog's Algol-inspired programming constructs allow a user to program an agent's behavior while leaving parts of the program underconstrained or "open" through the use of non-deterministic constructs. These underconstrained regions of the program are later filled in by a planner. Such integration of planning and programming has proven useful in a variety of diverse applications including soccer playing robots (Ferrein, Fritz, & Lakemeyer 2004), museum tour-guide robots (Burgard *et al.* 1999), and Web service composition (McIlraith & 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. In this paper 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 quadratic in the size of the original program. As a result, we provide semantics to ConGolog without Trans and Final predicates.

The compilation is of both practical and theoretical significance. 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, Fritz, & McIlraith 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. This paper also provides the theoretical justification for the previous compilation.

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.

Another practical advantage resulting from our compilation is that the compiled theory allows us to reason about the executions of programs using regression. This has a variety of interesting applications, including execution monitoring (Fritz & McIlraith 2007).

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 (cf. Reiter 2001).

This paper focuses on mathematical foundations. Pseudocode for our compilation is included in the appendix and an implementation will be available on our web site (`www.cs.toronto.edu/kr/`). Experimental evidence in support of our basic approach can be found in our previous paper (Baier, Fritz, & McIlraith 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 (Reiter 2001). Its basic elements are situations, actions, and fluents. 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 argument (e.g., $F(\vec{x},s)$), and are used to define the state of the world.

A basic action theory in the situation calculus, $\mathcal{D}$, comprises four *domain-independent foundational axioms*, and a set of *domain-dependent axioms*. Foundational axioms define basic properties of the tree structure of situations (full details can be found in Reiter's book (2001)), and contain one second-order induction axiom required to properly define the tree of situations. Included in the domain-dependent axioms are the following sets:

Axioms for the Initial State: sentences relativized to situation S_0, specifying what is true in the initial state.

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, 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.

Action Precondition Axioms: are first-order axioms that specify the conditions under which an action is 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 $\vec{x},s$.

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

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).

2.2 Golog and ConGolog

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

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 place of an object. 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.

ConGolog adds concurrency to Golog, by allowing 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 δ can be legally terminate in s. De Giacomo, Lespérance, & Levesque (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: $Trans(\delta_1;\delta_2,s,\delta',s') \equiv (\exists\gamma).\delta' = (\gamma;\delta_2) \wedge Trans(\delta_1,s,\gamma,s') \vee Final(\delta_1,s) \wedge Trans(\delta_2,s,\delta',s')$.

For concurrency constructs we have:

$Trans(\delta_1 \parallel \delta_2,s,\delta',s') \equiv (\exists\gamma).\delta' = (\gamma \parallel \delta_2) \wedge Trans(\delta_1,s,\gamma,s') \vee \delta' = (\delta_1 \parallel \gamma) \wedge Trans(\delta_2,s,\gamma,s')$,

$Trans(\delta_1 \rangle\!\rangle \delta_2,s,\delta',s') \equiv (\exists\gamma).\delta' = (\gamma \rangle\!\rangle \delta_2) \wedge Trans(\delta_1,s,\gamma,s') \vee \delta' = (\delta_1 \rangle\!\rangle \gamma) \wedge Trans(\delta_2,s,\gamma,s') \wedge (\nexists\zeta,s'').Trans(\delta_1,s,\zeta,s'')$,

$Trans(\delta^{\parallel},s,\delta',s') \equiv (\exists\gamma).\delta' = (\gamma \parallel \delta^{\parallel}) \wedge Trans(\delta,s,\gamma,s')$.

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

$Final(\delta_1 \parallel \delta_2,s) \equiv Final(\delta_1,s) \wedge Final(\delta_2,s)$,

$Final(\delta_1 \rangle\!\rangle \delta_2,s) \equiv Final(\delta_1,s) \wedge Final(\delta_2,s)$,

$Final(\delta^{\parallel},s) \equiv True$.

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 the *Do* predicate that defines the situations that result from executing a program:

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

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 From ConGolog to Basic Action Theories

In this section we describe an algorithm for compiling a given ConGolog program into a 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 the appendix, and a Prolog implementation will be available from our website.

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 δ_{main} 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 of the machine.

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 five steps.

(Step 1) For each procedure $P_j(t_{j_1},\ldots,t_{j_{k_j}},\delta_{P_j}) \in \mathcal{P}$ we call

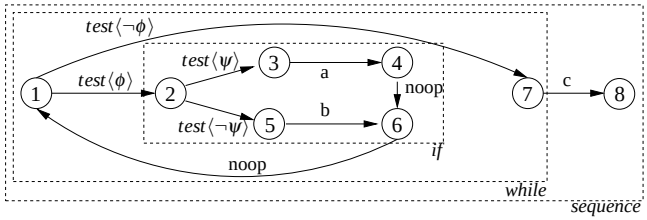
$$(\mathtt{ax}_j,\mathtt{i}_j) = \textbf{COMP}(\delta_{P_j},\{t_{j_1},\ldots,t_{j_{k_j}}\},0,P_j)$$

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. The function **COMP**, defined in Appendix A, takes as input a ConGolog program, a set of program variables, an integer used as a program counter, and a procedure name, used to distinguish different contexts. It outputs a set of sentences $\mathtt{ax}$, and an integer $\mathtt{i}_j$, 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, let us consider the function **COMP** in more detail.

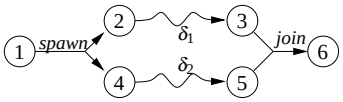
COMP is defined recursively over the structure of programs. Starting from an initial place labeled $(0,main)$, **COMP** incrementally constructs the Petri net, generating new network places as it recurses over the structure of the program. Assume **COMP** is currently at a place labeled with (i,p), where i is the program counter and p a procedure name, and that it encounters a primitive action A 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 A. **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 A. In the described case it generates $Poss(A(th),s) \subset Thread(th,s) \wedge state(th,s) = (i,p)$ which states that we can execute A 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 A is performed in (i,p), the token moves to $(i+1,p)$. The sentence generated in this case is $state(th,do(A(th),s)) = (i+1,p) \subset state(th,s) = (i,p)$.

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 unconditional transitions.[1] To keep the presentation simple, we only show the

[1] Names used for test actions in this example are simplified for clarity. Refer to the pseudo-code for more details.



(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.

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:

$$Poss(test(th,1,2,main),s) \subset (Thread(th,s) \wedge \phi(s) \wedge state(th,s) = (1,main)), \quad (1)$$

$$state(th,do(test(th,1,2,main)),s)) = (2,main). \quad (2)$$

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

$$Poss(c(th),s) \subset (Thread(th,s) \wedge state(th,s) = (7,main)), \quad (3)$$

$$state(th,do(c(th),s)) = (8,main) \subset state(th,s) = (7,main). \quad (4)$$

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 3. §

So far, the Petri net is equivalent to a simple automaton, since we've 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 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 here elaborate briefly on these difficulties.

Procedure calls are realized using two new action *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 runtime instead. Since procedures can be recursive, we require a stack, containing all (recursive) return states. The stack is realized using two functional fluents $stack(th,v,s)$ and $sp(th,s)$, where the former denotes the content of the stack entries, and the latter is a stack-pointer, always pointing to the next free stack position.

Concurrency is realized by using explicit thread names. Each action is given an additional parameter *th*, denoting the thread they are executed in. This is necessary, since there may be situations where two threads intent 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 $Thread(th,s)$, 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 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 $[0,N]$ where N is the number of the child. The k-th child of the n-th child of the main thread is called $[0,n,k]$. 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 *Prio* which indicates any threads that take priority over others. A thread can only proceed when no prioritized thread can perform its next action.

Program variables as created by π constructs, are realized using the fluents $Bound(x,s)$ and $map(x,s) = y$. The former states whether a variable is bound or free, and if bound, *map* states its 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 $(\pi(v,\delta))^\parallel$ where in each thread a new variable of the same name is created. Similarly stack positions are required when program variables are created in recursive procedures.

The combination of program variables and threads was particularly challenging, since program variables may be used in different context than where they were created. Consider for instance $\pi(v,(v = 1? \parallel v = 2?))$. This program is unexecutable since the v's refer to the same variable, which from the perspective of the second thread belongs to a different thread. We realize this through "pointers", formalized as a functional fluent *pipo*.

To compile the main procedure we call

$$(\mathtt{ax}_{main}, \mathtt{i}_{main}) = \textbf{COMP}(\delta_{main}, \emptyset, 0, main),$$

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 $(\mathtt{i}_{main}, main)$, the program has executed successfully, which 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 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 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 $state(th, s) = (\mathtt{i}_j, P_j)$.

(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 $trans(th, s)$, which indicates whether in situation s a given thread *th* can perform its next action. This definition is only required in conjunction with prioritized concurrency, and can be skipped if this language feature is not used.

(Step 5) For each primitive action A (including bookkeeping actions), Step 5 removes all sentences $Poss(A, s) \subset \phi$ from AX and combines them into a new precondition axiom for A, by: (a) disjoining all ϕ's, (b) conjoining the resulting formula with any preexisting preconditions for A, and (c) conjoining the result with an additional expression that governs priority among threads and allows forced execution of a selected thread. 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 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.

The result is a set of precondition and successor state axioms, describing the dynamics of all procedures' Petri nets. We also add the axiom $state([0], S_0) = (0, main)$, 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 are 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 result of compiling $\mathcal{P}$ into $\mathcal{D}$. We can show the following theorems which state that the compilation is both correct and succinct. The proofs can be found in (Fritz, Baier, & McIlraith 2008).

Theorem 1. Let S be any ground situation term of $\mathcal{D}$. Then $\mathcal{D} \models Do(\delta_{main}, S_0, S)$ iff there is a ground situation term S' in $\mathcal{D}_\mathcal{P}$ such that $S = filter_\mathcal{D}(S')$ and $\mathcal{D}_\mathcal{P} \models executable(S')$ and $(\exists th).Thread(th, S') \wedge state(th, S') = (\mathtt{i}_{main}, main)$.

Here $filter_\mathcal{D}(s')$ is a function that removes from the situation term s' any actions not defined in $\mathcal{D}$. This removes all bookkeeping actions from s', 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. Let the size of a set of procedures be the sum of the sizes of its contained programs (procedure bodies). Similarly, the size of an axiom is measured in 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 $< k$, 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. ($\mathcal{D}_\mathcal{P}$ can be generated automatically by our compiler implementation, which will be available on our website.) 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 $\mathcal{D}_\mathcal{P} \models \mathit{state}([0], s) = (\mathtt{i}_{\mathit{main}}, \mathit{main})$, with $\mathtt{i}_{\mathit{main}}$ as defined above, is a legal execution of the program. Regressing the condition $\mathit{state}([0], s) = (\mathtt{i}_{\mathit{main}}, \mathit{main})$ 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 conditions under which a given sequence of actions (whose parameters don't need to be ground) will satisfy formula φ while executing the program. These queries could not be answered using regression in the traditional semantics since neither the semantics of Golog nor ConGolog were in terms of regressable formulae[2]. One practical application for this is noted in Section 4.2.

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 Σ_{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 proven properties of simple Golog programs by representing $\mathcal{D}_\mathcal{P}$ in the higher-order theorem prover PVS (Owre, Rushby, & Shankar 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, Lespérance, & Levesque 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 results 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, Lespérance, & Levesque (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, Lespérance, & Levesque (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 proven automatically by PVS in fractions of a second. Liu 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 compiling 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, Fritz, & McIlraith 2007), and shown that Golog domain control knowledge can speed up search of standard planning benchmarks. In the compilation proposed in this paper we are considering the richer variant ConGolog, that allows programs with possibly recursive procedures, and with various forms of concurrency. 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 & 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 of 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

[2](Reiter 2001, p. 62) defines regressable formulae.

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 (like 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 (i, p) pair.
3. Threads, which in our basic action theories are represented as list, 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, Fritz, & McIlraith 2007).

HTN to PDDL Hierarchical Task Networks (HTNs) (Erol, Hendler, & Nau 1994; Ghallab, Nau, & Traverso 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 flavours of HTNs 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, Nau, & Traverso (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 flavour of HTN to ConGolog is almost trivial.

In the HTN planning 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` $\mathit{head}(m)$ $p_1(\vec{v})$ $t_1(\vec{v}) \ldots p_n(\vec{v})$ $t_n(\vec{v})$`)` 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.

Space precludes us from providing the formal translation algorithm of these HTNs to ConGolog here, but roughly the construction proceeds as follows: For each method m, we create a new procedure $m(\vec{v}, \textbf{if } p_1(\vec{v}) \textbf{ then } \delta_1 \textbf{ else if} \ldots \textbf{else } (p_n(\vec{v})?; \delta_n))$, 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 The compilation method 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 & 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 Discussion and Related Work

In this paper 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 that 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. The compiler will be made available for download from our web site. 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 proven by regression of ConGolog programs followed by (first-order) theorem proving in the initial situation. Regression of ConGolog programs has practical application to execution monitoring.

There are several pieces of related work. In previous work we provided a compilation of Golog programs without procedures into PDDL (Baier, Fritz, & McIlraith 2007), showing that notable speedups 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, Hendler, & Nau (1994) into ConGolog. In this paper, we showed how the HTN formalism (Ghallab, Nau, & Traverso 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 & 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, Lespérance, & Levesque (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've seen above, 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 & 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 & McIlraith 2006)) we could prove similar properties, but restricted to only finite executions.

Acknowledgments

We would like to thank Steven Shapiro for an insightful discussion regarding the objectives and merits of this paper in general, and all aspects regarding the use of PVS for conducting semi-automatic proofs of program properties in particular. Our thanks goes also to Yves Lespérance for his useful comments on an early version of this paper. Finally, we thank the anonymous reviewers for their comments and suggestions, and gratefully acknowledge funding from the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Ontario Ministry of Research and Innovation Early Researcher Award.

References

Baier, J. A., and McIlraith, S. A. 2006. Planning with first-order temporally extended goals using heuristic search. In *Proc. of the 21st National Conference on Artificial Intelligence (AAAI)*, 788–795.

Baier, J. A.; Fritz, C.; and McIlraith, S. A. 2007. Exploiting procedural domain control knowledge in state-of-the-art planners. In *Proc. of the 17th Int'l Conference on Automated Planning and Scheduling (ICAPS)*, 26–33.

Burgard, W.; Cremers, A. B.; Fox, D.; Hähnel, D.; Lakemeyer, G.; Schulz, D.; Steiner, W.; and Thrun, S. 1999. Experiences with an interactive museum tour-guide robot. *Artificial Intelligence* 114(1-2):3–55.

Claßen, J., and Lakemeyer, G. 2008. A logic for non-terminating golog programs. In *Proc. of the 11th Int'l Conference on Knowledge Representation and Reasoning (KR)*.

De Giacomo, G.; Lespérance, Y.; and Levesque, H. 2000. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence* 121(1–2):109–169.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1994. HTN planning: Complexity and expressivity. In *Proc. of the 12th National Conference on Artificial Intelligence (AAAI)*, 1123–1128.

Ferrein, A.; Fritz, C.; and Lakemeyer, G. 2004. Online decision-theoretic Golog for unpredictable domains. In *Proc. of the 4th Int'l Cognitive Robotics Workshop, at ECAI-2004*.

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

Fritz, C., and McIlraith, S. A. 2007. Monitoring plan optimality during execution. In *Proc. of the 17th Int'l Conference on Automated Planning and Scheduling (ICAPS)*, 144–151.

Fritz, C.; Baier, J. A.; and McIlraith, S. A. 2008. ConGolog, Sin Trans: compiling ConGolog into basic action theories for planning and beyond (extended version). Technical Report CSRG-576, University of Toronto.

Funge, J. 1998. *Making Them Behave: Cognitive Models for Computer Animation*. Ph.D. Dissertation, University of Toronto, Toronto, Canada.

Gabaldon, A. 2002. Programming hierarchical task networks in the situation calculus. In *AIPS'02 Workshop on On-line Planning and Scheduling*.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Kuter, U.; Sirin, E.; Nau, D. S.; Parsia, B.; and Hendler, J. A. 2004. Information gathering during planning for web service composition. In *Proc. of the 3rd Int'l Semantic Web Conferece (ISWC)*, 335–349.

Lekavý, M., and Návrat, P. 2007. Expressivity of STRIPS-like and HTN-like planning. In *Proc. of Agent and Multi-Agent Systems: Technologies and Applications, First KES International Symposium (KES-AMSTA)*, 121–130.

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

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

McIlraith, S., and Son, T. 2002. Adapting golog for composition of semantic web services. In *Proc. of the 8th Int'l Conference on Knowledge Representation and Reasoning (KR)*, 482–493.

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research* 20:379–404.

Owre, S.; Rushby, J. M.; and Shankar, N. 1992. PVS: A prototype verification system. In *Proc. of the 11th Int'l Conference on Automated Deduction (CADE)*, 748–752.

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

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

Shapiro, S.; Lespérance, Y.; and Levesque, H. J. 2002. The cognitive agents specification language and verification environment for multiagent systems. In *Proc. of the 1st Int'l Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, 19–26.

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 $\mathtt{i}$ used as a program counter, a set $\mathtt{e}$ of program variables (introduces using the π-construct, see below), and the name of the procedure this program belongs to, $\mathtt{c}$. 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. Note that we here present the details only for the simplified case of the so-called non-synchronized versions of while–loops and if–then–else. The synchronized versions require rather complex additional machinery, including some form of backtracking.

function $\textbf{COMP}(\delta, \mathtt{i}, \mathtt{e}, \mathtt{c})$

Output: a tuple $(\mathtt{ax}, \mathtt{i}')$ with $\mathtt{ax}$ a set of sentences, $\mathtt{i}'$ an integer

Switch δ:
  nil: **return** $(\emptyset, \mathtt{i})$
  $A(t_1, \ldots, t_n)$ (where A is not a procedure):
    $\mathtt{ax} \leftarrow \{Poss(A(th, x_1, \ldots, x_n), s) \subset$
      $Thread(th, s) \wedge state(th, s) = (\mathtt{i}, \mathtt{c}) \wedge$
      $\textbf{BOUND}(\mathtt{e}, [t_1, \ldots, t_n], [x_1, \ldots, x_n]),$
    $state(th, do(A(th, \vec{x}), s)) = (\mathtt{i}+1, \mathtt{c}) \subset$
      $state(th) = (\mathtt{i}, \mathtt{c})\}$
    $\cup\ \textbf{BIND}(\mathtt{e}, \vec{t}, \vec{x}, do(A(th, \vec{t}), s))$
    return $(\mathtt{ax}, \mathtt{i}+1)$
  $(\phi?)$: **return** $(\textbf{TEST}(\phi, \mathtt{i}, \mathtt{i}+1, \mathtt{e}, \mathtt{c}), \mathtt{i}+1)$
  $(\delta_1; \delta_2)$:
    $(\mathtt{ax}_1, \mathtt{i}_1) \leftarrow \textbf{COMP}(\delta_1, \mathtt{i}, \mathtt{e}, \mathtt{c})$
    $(\mathtt{ax}_2, \mathtt{i}_2) \leftarrow \textbf{COMP}(\delta_2, \mathtt{i}_1, \mathtt{e}, \mathtt{c})$
    return $(\mathtt{ax}_1 \cup \mathtt{ax}_2, \mathtt{i}_2)$
  $(\delta_1 | \delta_2)$:
    $(\mathtt{ax}_1, \mathtt{i}_1) = \textbf{COMP}(\delta_1, \mathtt{i}+1, \mathtt{e}, \mathtt{c})$

$(\mathtt{ax}_2,\mathtt{i}_2) = \mathbf{COMP}(\delta_2,\mathtt{i}_1{+}1,\mathtt{e},\mathtt{c})$
$\mathtt{ax} = \{\ \mathbf{NOOP}(\mathtt{i},\mathtt{i}{+}1,\mathtt{c}),\mathbf{NOOP}(\mathtt{i},\mathtt{i}_1{+}1,\mathtt{c}),$
$\mathbf{NOOP}(\mathtt{i}_1,\mathtt{i}_2{+}1,\mathtt{c}),\mathbf{NOOP}(\mathtt{i}_2,\mathtt{i}_2{+}1,\mathtt{c})\}$
return $(\mathtt{ax}\cup\mathtt{ax}_1\cup\mathtt{ax}_2,\mathtt{i}_2{+}1)$
(**if** ϕ **then** δ_1 **else** δ_2):
$(\mathtt{ax}_1,\mathtt{i}_1) = \mathbf{COMP}(\delta_1,\mathtt{i}{+}1,\mathtt{e},\mathtt{c})$
$(\mathtt{ax}_2,\mathtt{i}_2) = \mathbf{COMP}(\delta_2,\mathtt{i}_1{+}1,\mathtt{e},\mathtt{c})$
$\mathtt{ax} \leftarrow \{\mathbf{TEST}(\phi,\mathtt{i},\mathtt{i}{+}1,\mathtt{e},\mathtt{c}),$
$\mathbf{TEST}(\neg\phi,\mathtt{i},\mathtt{i}_1{+}1,\mathtt{e},\mathtt{c}),$
$\mathbf{NOOP}(\mathtt{i}_1,\mathtt{i}_2,\mathtt{c})\}$
return $(\mathtt{ax}_1\cup\mathtt{ax}_2\cup\mathtt{ax},\mathtt{i}_2)$
(**while** ϕ **do** δ'):
$(\mathtt{ax},\mathtt{i}_1) = \mathbf{COMP}(\delta',\mathtt{i}{+}1,\mathtt{e},\mathtt{c})$
return $(\{\mathbf{TEST}(\phi,\mathtt{i},\mathtt{i}{+}1,\mathtt{e},\mathtt{c}),\ \mathbf{TEST}(\neg\phi,\mathtt{i},\mathtt{i}_1,\mathtt{e},\mathtt{c}),$
$\mathbf{NOOP}(\mathtt{i}_1,\mathtt{i},\mathtt{c})\}\cup\mathtt{ax},\mathtt{i}_1)$
(δ'^*):
$(\mathtt{ax},\mathtt{i}_1) = \mathbf{COMP}(\delta',\mathtt{i},\mathtt{e},\mathtt{c})$
return $(\mathtt{ax}\cup\{\mathbf{NOOP}(\mathtt{i},\mathtt{i}_1{+}1,\mathtt{c}),\mathbf{NOOP}(\mathtt{i}_1,\mathtt{i},\mathtt{c})\},\mathtt{i}_1{+}1)$
$(\pi(v,\delta))$:
$(\mathtt{ax}_1,\mathtt{i}_1) = \mathbf{COMP}(\delta,\mathtt{i}{+}1,\mathtt{e}\cup\{v\},\mathtt{c})$
$\mathtt{ax} \leftarrow \{Poss(pi(th,v,\mathtt{c},\mathtt{i}{+}1),s) \subset$
$Thread(th,s)\wedge state(th,s) = (\mathtt{i},\mathtt{c}),$
$Poss(free(th,v,\mathtt{c},\mathtt{i}_1,s) \subset$
$Thread(th,s)\wedge state(th,s) = (\mathtt{i}_1{-}1,\mathtt{c})\}$
return $(\mathtt{ax}\cup\mathtt{ax}_1,\mathtt{i}_1)$
$P(t_1,\ldots,t_n)$ (where $P(x_1,\ldots,x_n)$ is a procedure):
$\mathtt{s}' \leftarrow do(call(th,P,\mathtt{i}{+}1,\mathtt{c}),s)$
$\mathtt{ax} \leftarrow \{Poss(call(th,P,\mathtt{i}{+}1,\mathtt{c}),s) \subset$
$Thread(th,s)\wedge state(th,s) = (\mathtt{i},\mathtt{c})\}$
$\cup\ \mathbf{BINDPROC}(\mathtt{e},[t_1,\ldots,t_n],[x_1,\ldots,x_n],\mathtt{s}')$
return $(\mathtt{ax},\mathtt{i}{+}1)$
$(\delta_1 \parallel \delta_2)$:
$(\mathtt{ax}_1,\mathtt{i}_1) \leftarrow \mathbf{COMP}(\delta_1,\mathtt{i}{+}1,\mathtt{e},\mathtt{c})$
$(\mathtt{ax}_2,\mathtt{i}_2) \leftarrow \mathbf{COMP}(\delta_2,\mathtt{i}_1{+}1,\mathtt{e},\mathtt{c})$
$\mathtt{ax} \leftarrow \{Poss(spawn(th,\mathtt{c},\mathtt{i}{+}1,\mathtt{i}_1{+}1),s) \subset$
$Thread(th,s)\wedge state(th,s) = (\mathtt{i},\mathtt{c}),$
$Poss(join(th,\mathtt{c},\mathtt{i}_2{+}1),s) \subset$
$Thread(th,s)\wedge state(th,s) = (\mathtt{i}_1,\mathtt{c})\wedge$
$state([childp(th,s){-}1|th],s) = (\mathtt{i}_2,\mathtt{c})\}\cup$
$\{pipo(th',0,v,do(spawn(th,\mathtt{c},\mathtt{i}{+}1,\mathtt{i}_1{+}1),s)) = y \subset$
$th' = [childp(th,s)|th]\wedge y = pipo(th,sp(th,s),v)\}_{v\in\mathtt{e}}$
return $(\mathtt{ax}_1\cup\mathtt{ax}_2\cup\mathtt{ax},\mathtt{i}_2{+}1)$
$(\delta^{\parallel})$: **return** $\mathbf{COMP}(nil \parallel iconc(\delta),\mathtt{i},\mathtt{e},\mathtt{c})$
$(iconc(\delta))$:
$(\mathtt{ax}_1,\mathtt{i}_1) \leftarrow \mathbf{COMP}(\delta,\mathtt{i}{+}1,\mathtt{e},\mathtt{c})$
$\mathtt{ax} \leftarrow \{\mathbf{NOOP}(\mathtt{i},\mathtt{i}_1{+}1,\mathtt{c}),$
$Poss(spawn(th,\mathtt{c},\mathtt{i},\mathtt{i}{+}1),s) \subset$
$Thread(th,s)\wedge state(th,s) = (\mathtt{i},\mathtt{c}),$
$Poss(join(th,\mathtt{c},\mathtt{i}_1{+}2),s) \subset$
$Thread(th,s)\wedge state(th,s) = (\mathtt{i}_1{+}1,\mathtt{c})\wedge$
$(childp(th,s) = 0\vee$
$state([childp(th,s){-}1|th],s) = (\mathtt{i}_1,\mathtt{c}))\}\cup$
$\{pipo(th',0,v,do(spawn(th,\mathtt{c},\mathtt{i}{+}1,\mathtt{i}_1{+}1),s)) = y \subset$
$th' = [childp(th,s)|th]\wedge y = pipo(th,sp(th,s),v)\}_{v\in\mathtt{e}}$
return $(\mathtt{ax}_1\cup\mathtt{ax},\mathtt{i}_1{+}2)$
$(\delta_1\rangle\rangle\delta_2)$:
$(\mathtt{ax}_1,\mathtt{i}_1) \leftarrow \mathbf{COMP}(\delta_1,\mathtt{i}{+}1,\mathtt{e},\mathtt{c})$
$(\mathtt{ax}_2,\mathtt{i}_2) \leftarrow \mathbf{COMP}(\delta_2,\mathtt{i}_1{+}1,\mathtt{e},\mathtt{c})$
$\mathtt{ax} \leftarrow \{Poss(spawn(th,\mathtt{c},\mathtt{i}{+}1,\mathtt{i}_1{+}1),s) \subset$
$Thread(th,s)\wedge state(th,s) = (\mathtt{i},\mathtt{c}),$
$Poss(join(th,\mathtt{c},\mathtt{i}_2{+}1),s) \subset$
$Thread(th,s)\wedge state(th,s) = (\mathtt{i}_1,\mathtt{c})\wedge$
$state([childp(th,s){-}1|th],s) = (\mathtt{i}_2,\mathtt{c}),$
$Prio(th,th',do(spawn(th,\mathtt{c},\mathtt{i}{+}1,\mathtt{i}_1{+}1),s)) \subset$
$th' = [childp(th,s)|th],$
$\neg Prio(th,th',do(join(th,\mathtt{c},\mathtt{i}_2{+}1),s)) \subset$
$th' = [childp(th,s){-}1|th]\}\cup$
$\{pipo(th',0,v,do(spawn(th,\mathtt{c},\mathtt{i}{+}1,\mathtt{i}_1{+}1),s)) = y \subset$
$th' = [childp(th,s)|th]\wedge y = pipo(th,sp(th,s),v)\}_{v\in\mathtt{e}}$
return $(\mathtt{ax}_1\cup\mathtt{ax}_2\cup\mathtt{ax},\mathtt{i}_2{+}1)$
EndSwitch

In the algorithm we use the auxiliary functions defined in Algorithm 1 to create additional transitions in the generated Petri net, which may be conditional (*test*) or unconditional (*noop*). In Algorithm 1, $\phi(s)|_{\mathtt{V}}$ denotes the formula resulting from substituting each occurence of v by x_v for every pair $(v,x_v)\in\mathtt{V}$. The following conditions are required for handling program variables in the positions of an actual argument of an action, or appearing in tests.

Algorithm 1 ConGolog2BAT: Auxiliaries

function $\mathbf{TEST}(\phi,\mathtt{i}_1,\mathtt{i}_2,\mathtt{e},\mathtt{c})$
$\mathtt{V} \leftarrow \{(v,x_v) \mid v\in\mathtt{e}\wedge\phi$ mentions $v\}$ // x_v a new var.
$\mathtt{s}' \leftarrow do(test(th,\mathtt{i}_1,\mathtt{i}_2,\mathtt{c},\vec{x}),s)$
return $\{state(th,\mathtt{s}') = (\mathtt{i}_2,\mathtt{c}) \subset True,$
$Poss(test(th,\mathtt{i}_1,\mathtt{i}_2,\mathtt{c},\vec{x}),s) \subset state(th,s) = (\mathtt{i}_1,\mathtt{c})\wedge$
$Thread(th,s)\wedge$
$\bigwedge_{(v,x_v)\in\mathtt{V}}[Bound(pipo(th,sp(th,s),v,s),s) \supset$
$map(pipo(th,sp(th,s),v,s),s) = x_v]\wedge\phi(s)|_{\mathtt{V}}\}$
$\cup\ \{Bound(p,\mathtt{s}') \subset p = pipo(th,sp(th,s),v,s),$
$map(p,\mathtt{s}') = x_v \subset p = pipo(th,sp(th,s),v,s)\}_{(v,x_v)\in\mathtt{V}}$
end function

function $\mathbf{NOOP}(\mathtt{i}_1,\mathtt{i}_2,\mathtt{c})$
return $\{Poss(noop(th,\mathtt{i}_1,\mathtt{i}_2,\mathtt{c}),s) \subset$
$Thread(th,s)\wedge state(th,s) = (\mathtt{i}_1,\mathtt{c}),$
$state(th,do(noop(th,\mathtt{i}_1,\mathtt{i}_2,\mathtt{c}),s)) = (\mathtt{i}_2,\mathtt{c}) \subset True\}$
end function

$\mathbf{BOUND}(\mathtt{e},[\mathtt{t}_1,\ldots,\mathtt{t}_n],[\mathtt{x}_1,\ldots,\mathtt{x}_n]) \stackrel{\text{def}}{=} \bigwedge_{\mathtt{t}_j,\text{ s.t. }\mathtt{t}_j\notin\mathtt{e}} \mathtt{x}_j = \mathtt{t}_j \wedge$
$\bigwedge_{\mathtt{j},\text{ s.t. }\mathtt{t}_j\in\mathtt{e}}[Bound(pipo(th,sp(th,s),\mathtt{t}_j,s),s) \supset$
$map(pipo(th,sp(th,s),\mathtt{t}_j,s),s) = \mathtt{x}_i]$

$\mathbf{BIND}(\mathtt{e},[\mathtt{t}_1,\ldots,\mathtt{t}_n],[\mathtt{x}_1,\ldots,\mathtt{x}_n],\mathtt{s}') \stackrel{\text{def}}{=}$
$\{Bound(p,\mathtt{s}') \subset p = pipo(th,sp(th,s),\mathtt{t}_j,s),$
$map(p,\mathtt{s}') = \mathtt{x}_i \subset p = pipo(th,sp(th,s),\mathtt{t}_j,s)\}_{\mathtt{j}\text{ s.t. }\mathtt{t}_j\in\mathtt{e}}$

$\mathbf{BINDPROC}(\mathtt{e},[\mathtt{t}_1,\ldots,\mathtt{t}_n],[\mathtt{x}_1,\ldots,\mathtt{x}_n],\mathtt{s}') \stackrel{\text{def}}{=}$
$\{pipo(th,y,\mathtt{x}_j,\mathtt{s}') = p \subset$
$y = sp(th,s){+}1\wedge p = pipo(th,sp(th,s),\mathtt{t}_j,s)\}_{\mathtt{j}\text{ s.t. }\mathtt{t}_j\in\mathtt{e}}$
$\cup\ \{Bound(p,\mathtt{s}') \subset p = (th,sp(th,s){+}1,\mathtt{x}_j),$
$map(p,\mathtt{s}') = z \subset p = (th,sp(th,s){+}1,\mathtt{x}_j)\wedge z = \mathtt{t}_j(s),$
$pipo(th,y,\mathtt{x}_j,s) = (th,y,\mathtt{x}_j) \subset y = sp(th,s){+}1\}_{\mathtt{j}\text{ s.t. }\mathtt{t}_j\notin\mathtt{e}}$

Note that in $\mathbf{BINDPROC}$ the variables $\mathtt{t}_i$ serve as actual parameters and $\mathtt{x}_i$ as formal parameters, and that we apply *call-by-value* by evaluating all actual parameters which are not program variables before passing them to the procedure ($z = t_j(s)$).

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). $\mathtt{ax}_{\text{common}} = \{$
$sp(th,do(call(th,x_1,x_2,x_3),s)) = y \subset y = sp(th,s){+}1,$

$state(th, do(call(th, P, x_1, x_2), s)) = y \subset y = (0, P)$,
$stack(th, v, do(call(th, x_1, i, \mathsf{c}), s)) = y \subset y = (i, \mathsf{c}) \wedge v = sp(th, s)+1$,
$state(th, do(return(th), s) = y \subset y = stack(th, sp(th, s), s)$,
$sp(th, do(return(th), s)) = y \subset y = sp(th, s)–1$,
$Thread(th', do(spawn(th, x_1, x_2, x_3), s)) \subset th' = [childp(th, s)|th]$,
$sp(th', do(spawn(th, x_1, x_2, x_3), s)) = y \subset$
$\quad y = 0 \wedge th' = [childp(th, s)|th]$,
$childp(th, do(spawn(th, x_1, x_2, x_3), s)) = y \subset y = childp(th, s)+1$,
$childp(th', do(spawn(th, x_1, x_2, x_3), s)) = y \subset$
$\quad y = 0 \wedge th' = [childp(th, s)|th]$,
$Forced(th', do(spawn(th, x_1, x_2, x_3), s)) \subset$
$\quad Forced(th) \wedge th' = [childp(th, s)|th]$,
$state(th', do(spawn(th, \mathsf{c}, x_1, i), s)) = y \subset$
$\quad y = (i, \mathsf{c}) \wedge th' = [childp(th, s)|th]$,
$state(th, do(spawn(th, \mathsf{c}, i, x_1), s)) = y \subset y = (i, \mathsf{c})$,
$Prio(th', x, do(spawn(th, x_1, x_2, x_3), s)) \subset$
$\quad th' = [childp(th, s)|th] \wedge Prio(th, x)$,
$Prio(x, th', do(spawn(th, x_1, x_2, x_3), s)) \subset$
$\quad th' = [childp(th, s)|th] \wedge Prio(x, th)$,
$\neg Thread(th', do(join(th, x_1, x_2), s)) \subset$
$\quad childp(th, s) > 0 \wedge th' = [childp(th, s)–1|th]$,
$childp(th, do(join(th, x_1, x_2), s)) = y \subset$
$\quad childp(th, s) > 0 \wedge y = childp(th, s)–1$,
$Forced(th, do(join(th, x_1, x_2), s)) \subset Forced([childp(th, s)|th]$,
$pipo(th', v, z, do(pi(th, z, x_1, x_2), s)) = y \subset$
$\quad y = (th, v, z) \wedge th' = th \wedge v = sp(th, s)$,
$state(th, do(pi(th, x_1, \mathsf{c}, i), s)) = y \subset y = (i, \mathsf{c})$,
$\neg Bound((th, v, z), do(free(th, z, x_2, x_3), s)) \subset v = sp(th, s)$,
$state(th, do(free(th, x_1, \mathsf{c}, i), s)) = y \subset y = (i, \mathsf{c})$,
$Thread([0], S_0) \subset True$,
$state([0], S_0) = (0, \text{'main'}) \subset True$,
$sp([0], S_0) = 0 \subset True$,
$stack([0], 0, S_0) = \text{'final'} \subset True$,
$childp([0], S_0) = 0 \subset True\}$