An Embedding of ConGolog in 3APL

Koen V. Hindriks[1] and **Yves Lespérance**[2] and **Hector Levesque**[3]

Abstract. Several high-level programming languages for programming agents and robots have been proposed in recent years. Each of these languages has its own features and merits. It is still difficult, however, to compare different programming frameworks and evaluate the relative benefits and disadvantages of these frameworks. In this paper, we present a general method for comparing agent programming frameworks based on a notion of bisimulation, and use it to formally compare the languages ConGolog and 3APL.

1 Introduction

A number of proposals for agent programming languages exist in the literature. Some of these languages are based on a notion of agent that associates a mental state consisting of beliefs and goals with the agent [11, 10, 5]. Although on first sight these languages may seem quite different, in [2, 4] it is shown that they are closely related. An interesting alternative for agent programming, based on a logical perspective, is offered by the concurrent language ConGolog [1]. In this paper, we present a formal comparison of ConGolog with the agent language 3APL.

ConGolog is a language for high-level robot programming. ConGolog, like its predecessor Golog [6], is an extension of the situation calculus that supports complex actions as well as a logic programming language for agents and robots. 3APL is an agent programming language and its semantics offers a more operational perspective on agents. The language is a combination of logic and imperative programming and provides operators for beliefs, goals and plans of an agent. We show that ConGolog and 3APL are closely related languages by constructing an embedding of ConGolog in 3APL. This embedding shows how ConGolog programs can be translated into equivalent 3APL programs. A number of interesting issues need to be resolved to construct the embedding. These include a comparison of states in 3APL with situations in ConGolog, the form of basic action theories, complete vs. incomplete knowledge, and execution models specifying the flow of control in agent programs.

2 The Situation Calculus

ConGolog is a programming language specified in and based upon the *situation calculus* [8]. ConGolog can be viewed upon as an extension of basic action theories in the situation calculus to a real programming language, in the sense that it allows for more complex program structures built from basic actions. We do not present a detailed overview of the situation calculus here (see [9]), but only give an outline of its main features. The situation calculus is a three-sorted, first-order logical theory (extended with some second order features). The language of the situation calculus $\mathcal{L}_{sitcalc}$ has three sorts: *situations*, *actions*, and *objects*. The intended interpretation of situations is that they are *action histories*. Actions are deterministic transitions from one situation to the next. The sort *object* is a catch-all sort for everything that is not a situation or an action. A constant S_0 of sort situation is used to denote the *initial situation*, and a function $do(a, s)$ of sort : *action* $\times$ *situation* $\rightarrow$ *situation* is used to denote the *successor situation* resulting from performing action a in situation s. A special predicate $Poss(a, s)$ is used to state that a can be executed in s. Predicate symbols of sort $object^n \times situation$ whose truth values may vary from situation to situation are called *relational fluents*. In this paper, we do not allow *functional fluents*, i.e. functions whose values varies from situation to situation.

In the sequel, we will often be interested in the formulas that hold in the 'current' situation. To identify the formulas that talk about a particular situation, we introduce the notions of a *uniform term* and *uniform formula*. A term or formula that is uniform *in a situation S* only refers to this situation S.

Definition 2.1 Let S be any term of sort situation. Then the set T_S of terms *uniform in S* is inductively defined by: (i) $S \in T_S$, (ii) if a term t does not mention a term of sort situation, then $t \in T_S$, (iii) if f is an n-ary function symbol other than *do* and $t_1, \ldots, t_n \in T_S$ whose sorts are appropriate for f, then $f(t_1, \ldots, t_n) \in T_S$.
The set $\mathcal{L}_S$ of formulas *uniform in S* is inductively defined by: (i) if $t_1, t_2 \in T_S$ are of the same sort, then $t_1 = t_2 \in \mathcal{L}_S$, (ii) if P is an n-ary predicate symbol, other than *Poss*, and $t_1, \ldots, t_n \in T_S$ are of the appropriate sorts, then $P(t_1, \ldots, t_n) \in \mathcal{L}_S$ (iii) if $\varphi_1, \varphi_2 \in \mathcal{L}_S$, then $\neg\varphi_1, \varphi_1 \wedge \varphi_2 \in \mathcal{L}_S$ (iv) if $\varphi \in \mathcal{L}_S$ and x is a variable not of sort situation, then $\forall x(\varphi) \in \mathcal{L}_S$.

We introduce a special constant *now* of sort situation and denote by $\mathcal{L}_{now}$ the set of formulas uniform in *now*. The intended interpretation of this constant is that it denotes the current situation. If σ is any (set of) formula(s) that is uniform in *now*, we denote by $\sigma[S]$ the (set of) formula(s) that is obtained by substituting S for *now* in σ. Note that $\sigma[S]$ is uniform in S.

3 Basic Action Theories

The basic intuitions associated with situations are captured by a number of *foundational axioms* (cf. [9]), but these are not discussed here since they do not play an important role in this paper. We are interested in the situation calculus as a framework for specifying actions, and we will use theories of a particular type for this. Axioms of the form $Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$ define the predicate *Poss* and specify

[1] Institute of Information & Computing Sciences, University Utrecht, Padualaan 14, De Uithof, Postbus 80 089, 3508 TB Utrecht, Holland, email: koenh@cs.uu.nl
[2] Department of Computer Science, York University, 4700 Keele Street, North York, Ontario, Canada M3J 1P3, email: lesperan@cs.yorku.ca
[3] Department of Computer Science, 6 King's College Road, Pratt Building, Room PT290C, University of Toronto, Toronto, Ontario M5S 3H5, email: hector@cs.toronto.edu

when an action is *enabled*, i.e. can be executed. They are called *action precondition axioms*. The formula $\Pi_A(\vec{x}, s)$ must be uniform in s to make sure that the preconditions of an action $A(\vec{t})$ depend only on the current situation s . *Successor state axioms* relate the value of fluents in the situation that results from doing an action to their value in the previous situation, and define the effects of executing an action. They also provide a solution for the frame problem. These axioms are 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 uniform in s . The uniformity condition on $\Phi_F(\vec{x}, a, s)$ guarantees that the successor state (the database of formulas that hold in a situation) can be computed from the the previous state. Finally, *unique names axioms for actions* are introduced to make sure that action names refer to different actions and include axioms of the form $A(\vec{x}) \neq B(\vec{y})$ for any two action symbols A and B and axioms of the form $A(\vec{x}) = A(\vec{y}) \rightarrow \vec{x} = \vec{y}$ for all action symbols A.

A basic action theory is a collection of these axioms. Our definition of a basic action theory slightly differs from the one in [7]. The main difference is that we do not include the initial situation axioms or *initial database* in the action theory.

Definition 3.1 A *basic action theory* is a set $\mathcal{A} = \Sigma \cup \mathcal{A}_{ss} \cup \mathcal{A}_{ap} \cup \mathcal{A}_{una}$ where Σ is a set of *foundational axioms*, $\mathcal{A}_{ss}$ a set of *successor state axioms* for relational fluents, one for each fluent, $\mathcal{A}_{ap}$ a set of *action precondition axioms*, one for each action symbol, and $\mathcal{A}_{una}$ a set of *unique names axioms for actions*.

An *initial database* is a finite set of (first-order) formulas from $\mathcal{L}_{sitcalc}$ that are uniform in S_0.

One of the more important reasons for excluding functional fluents is that in the presence of functional fluents the restriction that preconditions depend only on the current state is easily violated. Substitution of functional fluents $\vec{t}$ for some of the parameters of an action $A(\vec{x})$ that refer to other situations than the 'current' situation S may result in a precondition $\Pi_A(\vec{t}, S)$ that is not uniform. For example, by substituting the term $loc(Ball, do(throw(Ball), S_0))$ for x and situation S_0 for s in $Poss(goto(x), s) \equiv reachable(x, s)$ we obtain the precondition $reachable(loc(Ball, do(throw(Ball), S_0)), S_0)$ which is *not* uniform in S_0.

There is an important difference between situations in the situation calculus and states in a state-based approached. The difference is due to certain logical features of the situation calculus, like quantification over situations and functional fluents. The main characteristic of a state-based approach is that the action that is performed and the current state completely determine the next state. In contrast, successor state axioms and a database associated with a situation S may not be enough to compute the database associated with a successor situation $do(A, S)$. As an example, consider: $corner(loc(Robot, do(goto(loc(Ball, do(trow(Ball), S_0)), S_0))))$. This formula is uniform in $do(goto(loc(Ball, do(trow(Ball), S_0)), S_0)$, but cannot be evaluated by inspection of a single situation. Informally, the formula states that *Robot* is at location $loc(Ball, do(throw(Ball), S_0))$, which is a corner, after going to that location in situation S_0. This formula, however, can only be evaluated by inspecting the would-be situation resulting from throwing the *Ball* in situation S_0, and checking whether the location of the *Ball* in *that* situation is a corner, assuming that a goto action always succeeds.

4 ConGolog

ConGolog is a logic programming language based on the situation calculus. It extends basic action theories of the previous section with operators for constructing complex actions. We discuss a subset of all the programming constructs in [1]. Parallel and prioritised parallel composition are dealt with in [3]. Most of the programming constructs listed below are well-known. The pick operator $\pi x.\delta$ is an operator that nondeterministically selects a value for x and then continues with the execution of δ.

Definition 4.1 The set of *open* programs P is inductively defined by:

- *primitive actions* $a \in P$,
- *tests* $\phi? \in P$, for $\phi \in \mathcal{L}_{now}$,
- *sequential composition* $(\delta_1;\ \delta_2) \in P$ if $\delta_1, \delta_2 \in P$,
- *nondeterministic choice* $(\delta_1 \mid \delta_2) \in P$ if $\delta_1, \delta_2 \in P$,
- *nondeterministic choice of arguments* $\pi x.\delta \in P$ if $\delta \in P$ and x is a variable of sort object,
- *procedure calls* $P(\vec{t})$,
- *procedure definitions* **proc** $P(\vec{x})\ \delta_P$ **end**.

By definition, the set of *ConGolog programs* is the set of *closed* programs in P.

The meaning of these constructs is specified using a transition semantics presented in a non-standard way. A new predicate *Trans* is added to the language of the situation calculus. The predicate $Trans(\delta, s, \delta', s')$ expresses that program δ can perform a computation step in situation s resulting in a new situation s' where δ' is the remaining program for execution. The semantics of ConGolog programs is specified by means of a set of axioms for the predicate *Trans*. The expression *nil* below denotes the 'empty' program, and is used as an auxiliary construct in the definition of the semantics. In the definition of the semantics, the predicate $Final(\delta, s)$ is used to express that program δ may legally terminate in situation s.

Definition 4.2 *Trans* and *Final* are axiomatised by:[4]

The Empty Program:
$Trans(nil, s, \delta', s') \equiv False$,
$Final(nil, s) \equiv True$
Basic Actions:
$Trans(a, s, \delta', s') \equiv Poss(a, s) \wedge \delta' = nil \wedge s' = do(a, s)$,
$Final(a, s) \equiv False$
Tests:
$Trans(\phi?, s, \delta', s') \equiv \phi[s] \wedge \delta' = nil \wedge s' = s$,
$Final(\phi?, s) \equiv False$
Sequential Composition:
$Trans(\delta_1;\ \delta_2, s, \delta', s') \equiv$
$\exists \gamma.\delta' = (\gamma;\ \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee$
$Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s')$,
$Final(\delta_1;\ \delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)$
Nondeterministic Choice:
$Trans(\delta_1 \mid \delta_2, s, \delta', s') \equiv$
$Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s')$,
$Final(\delta_1 \mid \delta_2, s) \equiv Final(\delta_1, s) \vee Final(\delta_2, s)$
Nondeterministic Choice of Argument:
$Trans(\pi x.\delta, s, \delta', s') \equiv \exists x.Trans(\delta, s, \delta', s')$,
$Final(\pi x.\delta, s) \equiv \exists x.Final(\delta, s)$,
Procedure Calls:
$Trans(P(\vec{t}), s, \delta', s') \equiv Trans((\delta_P)^{\vec{x}}_{\vec{t}}, s, \delta', s')$,
$Final(P(\vec{t}), s) \equiv Final((\delta_P)^{\vec{x}}_{\vec{t}}, s)$.

[4] Formally, an encoding of ConGolog programs into terms of the first-order language $\mathcal{L}_{sitcalc}$ is required, as is done in [1]. However, since the details in this paper are almost completely the same, for notational convenience we use the programs themselves in the definitions below.

where δ_P is the body of the procedure P and $(\delta_P)^{\vec{x}}_{\vec{t}}$ is the program δ_P with variables $\vec{x}$ simultaneously substituted with $\vec{t}$.

The definition of *Trans* with respect to procedure calls requires some explanation. The definition differs from the second-order definition given in [1], but is based on the first-order version presented in that same paper. Procedure definitions are assumed to be global, and we do not allow nesting of such definitions in this paper. A first-order definition of the procedure semantics only works for the restricted type of procedure definitions that are *guarded*. A formal definition of this notion is presented in [1]. Informally, a guarded procedure is a procedure that never does more than a fixed number of procedure calls before executing an action or test. From now on, we assume that all procedure definitions are guarded. Also note that *no* transition is associated with the expansion of a procedure call into its body, a feature of both the second-order and the first-order definition.

5 The Agent Programming Language 3APL

3APL (pronounced "triple-a-p-l") is an agent programming language and is a combination of imperative programming and logic programming. We want to emphasise that 3APL is a *programming* language, and not a *logical* language like the situation calculus.

3APL agents or programs are built from similar constructs as the program constructs used in ConGolog programs, apart from some minor differences in the naming of these constructs. 3APL has facilities corresponding to basic actions $A(\vec{t})$, tests $\phi?$, sequential composition ; , nondeterministic choice $+$ and parallel composition. 3APL has similar facilities for recursive procedures as ConGolog, but also allows for more general rules to modify programs or plans of an agent in arbitrary ways.

One of the more important differences between ConGolog and 3APL is the presence of the πx operator in ConGolog and the absence of such a construct in 3APL which gives rise to quite different parameter mechanisms in the two programming languages. Whereas in ConGolog the πx operator is used to nondeterministically 'guess' values for variables *bound* by the operator, in 3APL tests are used to compute values for *free* variables. Moreover, whereas the π operator provides for an explicit scoping mechanism, the use of free variables in 3APL is based on implicit scoping. To facilitate the construction of an embedding of ConGolog in 3APL and to accommodate for this difference, below we introduce a construct $random(x)$ which corresponds to the nondeterministic guessing of a value for variable x.

An agent in the language 3APL is defined as a tuple $\langle \pi, \sigma, \mathcal{R} \rangle$, where π is a *goal*, σ is a set of beliefs drawn from some knowledge representation language $\mathcal{L}$, and $\mathcal{R}$ is a set of rules of the form $\pi \leftarrow \varphi \mid \pi'$ where π, π' are goals and φ is a belief. A goal π is an imperative-like program built from basic actions and tests just like a ConGolog program. 3APL does not make any commitment to a specific knowledge representation language $\mathcal{L}$, but only assumes a consequence relation $\models$ is associated with $\mathcal{L}$ for deriving facts from a belief base. For the purpose of simulating ConGolog, the knowledge representation language $\mathcal{L}$ is specialised to $\mathcal{L}_{now}$, the language of formulas uniform in *now* of the situation calculus, and $\models$ is the usual consequence relation of first-order logic.

5.1 Semantics of 3APL

The operational semantics of 3APL is defined by means of a Plotkin-style *transition system*. A transition system inductively defines a *transition relation* $\longrightarrow$ which is the analogue of the *Trans* predicate of the ConGolog semantics for 3APL. It consists of a set of *transition rules* which specify possible computation steps, denoted by $\longrightarrow$, that are associated with each of the constructs of the language.

One of the differences between ConGolog and 3APL, as we will see below, is that in ConGolog only the execution of basic actions or tests give rise to a transition, whereas in 3APL the expansion of a procedure into its body also corresponds to a transition. To distinguish these types of transitions in 3APL, we define a *labelled* transition relation $\xrightarrow{l}$, where the label l is $A(\vec{t})$ in case a basic action is executed, l is ϵ (the empty sequence) in case a test is executed, and l is i in all other cases. The label i is associated with the execution of a *random* action and the expansion of a procedure into its body. From the perspective of ConGolog, these transitions are not visible but are considered implementation details and therefore labelled with i to indicate that a so called *internal or silent step* has been performed. The labelling of transitions of complex programs is derived from the basic ones. Labels are also used to keep track of the *sequence of basic actions* that is executed during a computation of a 3APL program.

The transition relation $\longrightarrow$ is a relation on pairs $\langle \pi, \sigma \rangle$ where π is the current program or plan of the agent and σ is the current belief base. In the transition rules below, we use E to denote *successful termination*, and we identify $E;\ \pi$ with π.

The programming language 3APL does not fix a specific set of basic actions, but allows a programmer to define its own set of actions. In the semantics, only the *type* of actions is specified by a function $\mathcal{T}$. $\mathcal{T}$ is a partial function mapping basic actions and belief bases into belief bases and defines basic actions as *belief updates*. The semantics thus abstracts from any particular specification of actions.

A *test* $\phi?$ allows an agent to introspect its beliefs. A test is evaluated relative to the current beliefs of an agent and can also be used to compute bindings for free variables in the test as in logic programming. These bindings are recorded in a substitution γ, and used to instantiate variables for which a binding has been computed in the remaining program. Technically this is implemented by associating γ with $\longrightarrow$ (notation: $\longrightarrow_\gamma$). Substitutions implement the parameter mechanism of 3APL. $\varnothing$ is used to denote the empty substitution.

Definition 5.1

$$\frac{\mathcal{T}(A(\vec{t}), \sigma) = \sigma'}{\langle A(\vec{t}), \sigma \rangle \xrightarrow{A(\vec{t})}_{\varnothing} \langle E, \sigma' \rangle} \qquad \frac{\sigma \models \phi\gamma, dom(\gamma) = \mathsf{free}(\phi)}{\langle \phi?, \sigma \rangle \xrightarrow{\epsilon}_{\gamma} \langle E, \sigma \rangle}$$

A $random(x)$ action always succeeds and nondeterministically returns an arbitrary binding for x. $random(x)$ can be defined by the test $(P(x) \vee \neg P(x))?$. The *random* action is labelled as a silent step, because we want to use it to simulate the pick operator πx, which in the ConGolog semantics does not give rise to a transition.

Definition 5.2 Let t be a ground term.

$$\frac{x \text{ is a variable}}{\langle random(x), \sigma \rangle \xrightarrow{i}_{\{x=t\}} \langle E, \sigma \rangle} \qquad \frac{}{\langle random(t), \sigma \rangle \xrightarrow{i}_{\varnothing} \langle E, \sigma \rangle}$$

A sequence of two programs is executed by executing the first program and passing bindings γ computed by this program on to the remaining program by applying γ to it. The execution of a nondeterministic choice goal consists in selecting one of the subgoals that is enabled, executing this goal, and dropping the other. Only the rule for the selection of the left subgoal is given.

Definition 5.3

$$\frac{\langle \pi_1, \sigma \rangle \xrightarrow{l}_{\gamma} \langle \pi_1', \sigma' \rangle}{\langle \pi_1;\ \pi_2, \sigma \rangle \xrightarrow{l}_{\gamma} \langle \pi_1';\ \pi_2\gamma, \sigma' \rangle} \qquad \frac{\langle \pi_1, \sigma \rangle \xrightarrow{l}_{\gamma} \langle \pi_1', \sigma' \rangle}{\langle \pi_1 + \pi_2, \sigma \rangle \xrightarrow{l}_{\gamma} \langle \pi_1', \sigma' \rangle}$$

3APL practical reasoning rules operate on the goals of an agent. In this paper, we only consider rules of the form $P(\vec{t'}) \leftarrow \phi \mid \pi_b$ which are similar to the recursive procedure definitions of ConGolog. For an explanation of more general rules which allow arbitrary modification of a program and not just body replacement as with procedure calls, we refer the reader to [5]. A computation step due to an achievement goal $P(\vec{t})$ consists in the replacement of $P(\vec{t})$ with the body of the rule and the correct instantiation of the formal parameters with the actual parameters $\vec{t}$. Technically, this is implemented by means of a substitution for unifying the procedure call with the head of the rule. In contrast to ConGolog procedures definitions, rules are guarded and may compute appropriate values given the current context of computation to further instantiate the body of the rule.

Definition 5.4 Let η be a most general substitution s.t $P(\vec{t}) = P(\vec{t'})\eta$

$$\frac{\sigma \models \phi\eta\gamma, dom(\gamma) = \mathsf{free}(\phi)}{\langle P(\vec{t}), \sigma\rangle \xrightarrow{i}_{\eta\gamma} \langle \pi_b\eta\gamma, \sigma\rangle}$$

where $P(\vec{t'}) \leftarrow \phi \mid \pi_b$ is a rule in the rule base $\mathcal{R}$ of the agent.

5.2 Silent Steps

Because the semantics for ConGolog abstracts from silent steps, which are present in the semantics of 3APL, we introduce a new computation step relation $\Longrightarrow$ for 3APL which also abstracts from these steps. The relation $\Longrightarrow$ is derived from $\longrightarrow$ and used to construct an embedding of ConGolog into 3APL. $\Longrightarrow$ steps are complex steps, composed of an arbitrary number of silent steps $\xrightarrow{i}$ and a single step that involves the execution of a basic action or a test. From now on, we just write $\longrightarrow$ without mentioning substitutions anymore.

Definition 5.5 *(abstracting from silent steps)*
The transition relation $\overset{l}{\Longrightarrow}$, where l is either $A(\vec{t})$ or ϵ, is defined by: $\overset{l}{\Longrightarrow} \overset{df}{=} \xrightarrow{i}{}^{*} \cdot \xrightarrow{l}$ where * denotes the transitive closure of a relation.

6 Issues in Embedding ConGolog in 3APL

In this section, we discuss some distinguishing features of ConGolog and 3APL. To be able to construct an embedding of ConGolog into 3APL, a number of issues have to be dealt with. These include (1) requirements on the database or belief base, (2) a domain closure assumption, and (3) deriving an update semantics for 3APL actions from a basic action theory $\mathcal{A}$. A translation function τ is then defined which maps ConGolog programs to equivalent 3APL programs.

Operationalising Tests A particularly interesting difference between ConGolog and 3APL concerns the semantics of tests. Whereas the semantics of a test in 3APL is defined in terms of entailment by the current beliefs, in ConGolog a test is defined in terms of truth in the current situation. The difference can be illustrated with the program $\delta = \phi?;\ \delta_1 \mid \neg\phi?;\ \delta_2$. In 3APL, δ is not always enabled since neither ϕ nor $\neg\phi$ need to be entailed by the current beliefs. In ConGolog, however, δ is *always* enabled simply because either ϕ or $\neg\phi$ must hold in the current situation. The axiomatic definition of *Trans* thus implies that δ can perform a computation step.

A problem with the ConGolog semantics of tests, however, is how to implement it. As illustrated by the example program δ, it is in general not possible *to decide* which branch to execute for arbitrary tests ϕ due to incomplete databases. It is possible to avoid this problem by requiring that the (initial) database be *complete*. A database $\sigma \subseteq \mathcal{L}_{now}$ is called *complete* iff for every sentence φ in $\mathcal{L}_{now}$ either $\sigma \models \varphi$ or $\sigma \models \neg\varphi$. For complete databases σ, the difference between the ConGolog and 3APL semantics of tests also disappears, since the problem of evaluating $\phi \vee \neg\phi$ in the current situation and that of evaluating $\sigma \models \phi \vee \sigma \models \neg\phi$ then coincide.

Operationalising Nondeterministic Choice of Argument Another interesting difference concerns the specification of the parameter mechanisms in ConGolog and 3APL. Whereas ConGolog has an explicit operator πx for binding variables in a program (only closed programs are ConGolog programs!), the parameter mechanism in 3APL is based on an implicit binding mechanism where tests are used to compute bindings for *free* variables in a program. The axiomatic definition of the πx operator by means of the *logical existential quantifier*, however, again poses a problem concerning the implementation of this operator. To operationalise the guessing of a value by the operator πx we need a name for every possible value since computations proceed by manipulating terms. Therefore, we assume that action theories together with the current database satisfy domain closure, that is, $\mathcal{A} + \sigma[S] \models \forall x(x = t_1 \vee \ldots x = t_n)$ for $\sigma \subseteq \mathcal{L}_{now}$ and S a situation term.

Basic Actions, Progression and Belief Bases A third difference between ConGolog and 3APL is that the operational semantics of 3APL explicitly refers to states called *belief bases* whereas the axiomatic definition of the predicate *Trans* only mentions situations which *denote* such a state. In the operational semantics for 3APL, belief bases are updated by basic actions. The semantics of basic actions in ConGolog is provided by successor state axioms. For our purposes, we need a way to link successor state axioms to an update semantics for actions. This link is provided by the work of Lin and Reiter on the progression of databases [7]. They define a progression operator for (relatively) complete basic action theories which can be used to define a transition function $\mathcal{T}$ for 3APL basic actions.

Definition 6.1 The progression operator *Prog* is defined by:

$$\begin{array}{l} Prog(\sigma, \epsilon) = \sigma, \\ Prog(\sigma, A(\vec{t})) = \\ \{P(\vec{t}) \mid \sigma \models P(\vec{t}) \text{ and } P(\vec{t}) \text{ is situation independent}\} \cup \\ \{\neg P(\vec{t}) \mid \sigma \models \neg P(\vec{t}) \text{ and } P(\vec{t}) \text{ is situation independent}\} \cup \\ \{F(\vec{t'}, now) \mid \sigma \models \Phi_F(\vec{t'}, A(\vec{t}), now)\} \cup \\ \{\neg F(\vec{t'}, now) \mid \sigma \models \neg\Phi_F(\vec{t'}, A(\vec{t}), now)\} \end{array}$$

By theorem 3 in [7], the progression operator as defined in definition 6.1 yields a progression of a complete belief or data base σ. By theorem 1 in [7], we then know that any sentence $\varphi[do(\alpha, S)]$ uniform in $do(\alpha, S)$ is entailed by $\mathcal{A} + \sigma[S]$ iff $\mathcal{A} + \sigma[do(\alpha, S)]$ also entails $\varphi[do(\alpha, S)]$, which shows that the progression operator can be used to define an update semantics for actions.

The transition function $\mathcal{T}$ which assigns meaning to 3APL actions is derived from the progression operator. $\mathcal{T}$ incorporates both the information of precondition and successor state axioms.

Definition 6.2 Let $\sigma \subseteq \mathcal{L}_{now}$ be a complete theory, and S a closed term of sort *situation*. Define for every action $A(\vec{t})$ its semantics by:

$$\begin{array}{ll} \mathcal{T}(A(\vec{t}), \sigma) = Prog(\sigma, A(\vec{t})) & \text{if } \mathcal{A} + \sigma[S] \models Poss(A(\vec{t}), S), \\ \mathcal{T}(A(\vec{t}), \sigma) \text{ is undefined} & \text{otherwise.} \end{array}$$

Because instances of action precondition axioms must be uniform in a situation S, it follows that $\mathcal{T}$ is well-defined. The main point of this definition is that it shows how to reduce situations (action histories) to *states* for complete databases.

Translation Function τ Now that we have set the stage, we define a translation function τ from ConGolog programs to 3APL agents. The mapping τ is defined by induction on the structure of programs. One of the more interesting cases is the translation of a $\pi x.\delta$ program which is mapped to a sequential 3APL program $\mathit{random}(x);\ \tau(\delta)$. The πx operator is simulated by the special action *random*, and the explicit binding by the πx operator is replaced by the implicit binding mechanism in 3APL. Also note that a ConGolog procedure is translated to a 3APL rule with empty guard.

Definition 6.3 The translation function τ is inductively defined by:

- $\tau(\mathit{nil}) = E$, $\tau(A(\vec{t})) = A(\vec{t})$, $\tau(\phi?) = \phi?$,
- $\tau(\delta_1;\ \delta_2) = \tau(\delta_1);\ \tau(\delta_2)$, $\tau(\delta_1 \mid \delta_2) = \tau(\delta_1) + \tau(\delta_2)$,
- $\tau(\pi x.\delta) = \mathit{random}(x);\ \tau(\delta)$,
- $\tau(P(\vec{t}) = P(\vec{t})$, $\tau(\textbf{proc}\ P(\vec{x})\delta_P\ \textbf{end}) = P(\vec{x}) \leftarrow \tau(\delta_P)$.

7 An Embedding of ConGolog into 3APL

The basic idea of the embedding is to show that a ConGolog program can be *bisimulated* by a corresponding 3APL program. A program bisimulates another program in case every possible computation of the former can be matched with a computation of the latter, and vice versa. The concept of matching computations is defined in terms of the matching of single computation steps. Computation steps match if their behaviour is similar. In our case, we define this similarity in terms of the actions that are performed and the databases that are computed.

The main result of the paper is Theorem 7.2, which shows that a ConGolog program δ bisimulates with its τ-translation $\tau(\delta)$. The theorem shows that δ and $\tau(\delta)$ generate the same action sequences and produce the same databases. We first state a lemma that shows that the concept defined by the *Final* predicate for ConGolog programs δ coincides with successful termination of the corresponding 3APL τ-translations $\tau(\delta)$ modulo some internal steps. Both the lemma and the theorem are proven by induction on the structure of programs , and by making use of the completeness of databases and the domain closure assumption. The proofs can be found in [3].

Lemma 7.1 Let $\sigma \subseteq \mathcal{L}_{now}$ be a complete theory, $\varphi \in \mathcal{L}_{now}$, and S be a closed situation term. Then, for any action theory $\mathcal{A}$ and ConGolog program δ: $\mathcal{A} + \sigma[S] \models \mathit{Final}(\delta, S)$ iff $\langle \tau(\delta), \sigma\rangle \xrightarrow{i}{}^{*} \langle E, \sigma\rangle$.

The main embedding result is the following bisimulation theorem:

Theorem 7.2 Let $\mathcal{A}$ be a basic action theory with complete initial database $\sigma \subseteq \mathcal{L}_{now}$, δ, δ' be (closed) ConGolog programs, and α be an action sequence. Then:

$$\mathcal{A} + \sigma[S_0] \models \mathit{Trans}^*(\delta, S_0, \mathit{nil}, \mathit{do}(\alpha, S))$$
$$\text{iff}$$
$$\langle \tau(\delta), \sigma\rangle \stackrel{\alpha}{\Longrightarrow}{}^{*} \langle E, \mathit{Prog}(\sigma, \alpha)\rangle$$

8 Discussion

ConGolog is a logic programming language which extends basic action theories in the situation calculus with operators for building complex programs. The logical perspective of the situation calculus offers a very expressive framework for specifying agents. Basic action theories provide a framework for specifying actions and offer a solution to the frame problem. The logical semantics of ConGolog, however, does not straightforwardly provide an implementation language, in contrasts with the operational semantics of 3APL. The embedding result of this paper shows that one option to implement (a restricted version of) ConGolog is to embed the language into 3APL. Another important feature of the logical semantics is that in the presence of functional fluents, situations cannot be identified with states.

3APL is an agent programming language based on the agent-oriented approach. Its operational semantics is state-based and specified by means of a transition semantics. A clear distinction is made between the programming language and a programming logic for proving properties of 3APL agents. The agent language 3APL abstracts both from the knowledge representation that agents use and a concrete specification of actions. The embedding result shows that basic action theories in the situation calculus can be used to specify actions and to derive an update semantics for 3APL actions.

Both languages emphasise different aspects of agent computing. ConGolog is presented as a high-level programming alternative to planning. The focus is on extracting a legal action sequence from a nondeterministic program. A ConGolog program thus is seen as a vehicle for computing a situation (action history). As in planning, finding a legal action sequence requires search and this explains the use of a backtracking model of execution. The backtracking model is inhereted from logic programming, which is used to implement ConGolog [1].

With respect to 3APL, the focus is on computing belief bases. Upon termination a 3APL program returns a belief base. The execution model that is proposed is that of the 'imperative flow of control' [5]. The basic feature of this model is that a commitment to a choice is made as soon as an action has been executed. Because of the embedding result, it is clear, however, that neither the semantics of ConGolog nor that of 3APL dictates the use of one or the other model of execution.

REFERENCES

[1] Giuseppe De Giacomo, Yves Lespérance, and Hector Levesque, '*ConGolog*, a Concurrent Programming Language Based on the Situation Calculus', *Artificial Intelligence, accepted for publication*.

[2] Koen Hindriks, F.S. de Boer, Wiebe van der Hoek, and John-Jules Meyer, 'An Operational Semantics for the Single Agent Core of AGENT-0', Technical Report UU-CS-1999-30, Department of Computer Science, University Utrecht, (1999).

[3] Koen Hindriks, Yves Lespérance, and Hector J. Levesque, 'An Embedding of ConGolog in 3APL', Technical Report UU-CS-2000-13, Department of Computer Science, University Utrecht, (2000).

[4] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer, 'A Formal Embedding of AgentSpeak(L) in 3APL', in *Advanced Topics in Artificial Intelligence (LNAI 1502)*, eds., G. Antoniou and J. Slaney, 155–166, Springer-Verlag, (1998).

[5] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer, 'Agent Programming in 3APL', *Autonomous Agents and Multi-Agent Systems*, **2**(4), 357–401, (1999).

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

[7] Fangzhen Lin and Ray Reiter, 'How to Progress a Database', *Artificial Intelligence*, **92**, 131–167, (1997).

[8] J. McCarthy and P.J. Hayes, 'Some philosophical problems from the standpoint of artificial intelligence', in *Machine Intelligence*, eds., Meltzer and Michie, 463–502, Edinburgh University Press, (1969).

[9] Fiora Pirri and Ray Reiter, 'Some Contributions to the Metatheory of the Situation Calculus', *JACM, accepted for publication*, (1999).

[10] Anand S. Rao, 'AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language', in *Agents Breaking Away (LNAI 1038)*, eds., W. van der Velde and J.W. Perram, pp. 42–55. Springer-Verlag, (1996).

[11] Yoav Shoham, 'Agent-oriented programming', *Artificial Intelligence*, **60**, 51–92, (1993).