An Embedding of ConGolog in 3APL

Koen Hindriks, Yves Lespérance, and Hector Levesque

February 16, 2000

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. ConGolog is a concurrent language for high-level robot programming based on the situation calculus. ConGolog provides a logical perspective on robot programming, but also incorporates a number of imperative programming constructs like sequential composition. 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 concerning the flow of control of agent programs.

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 [16, 14, 7]. Although on first sight these languages may seem quite different, in [4, 5] 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 [2]. In this paper, we present a formal comparison of ConGolog with the agent language 3APL (pronounced "triple-a-p-l").

ConGolog is a language for high-level robot programming. ConGolog, like its predecessor Golog [8], 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 Basic Action Theories in the Situation Calculus

ConGolog is a programming language specified in and based upon the *situation calculus* [11]. ConGolog extends basic action theories in the situation calculus to a real programming language. It allows the construction of more complex program structures built from basic actions. Basic action theories are used to specify the preconditions and effects of basic actions that are used in ConGolog programs. A basic action theory only fixes the basic structure for specifying actions, but leaves the choice of basic actions to the programmer. We first introduce the situation calculus and then define what basic action theories are.

2.1 The Situation Calculus

The situation calculus is a three-sorted, first order logical language, extended with some second order features. The situation calculus is specifically designed for representing dynamically changing worlds. Changes are the result of named, deterministic actions, and a possible world history therefore can be identified with a sequence of actions. Finite action histories are represented by first order terms called *situations* in the situation calculus. The language of the situation calculus $\mathcal{L}_{sitcalc}$ has three sorts: A sort *situation*, a sort *action*, and a sort *object* for everything that is neither a situation nor an action.

Definition 2.1 *(alphabet of $\mathcal{L}_{sitcalc}$)*
The alphabet of $\mathcal{L}_{sitcalc}$ consists of the following sets of symbols:

- Countably infinitely many variables for each sort; we use s to denote variables of sort *situation*, a for variables of sort *action*, and x, y for variables of sort *object*.
- Two function symbols of sort *situation*: (1) the constant S_0 denoting the *initial situation*, and (2) the function *do* of sort : $action \times situation \rightarrow situation$ where $do(a, s)$ denotes the *successor situation* resulting from performing action a in situation s.
- A binary predicate $\sqsubset$ of sort : $situation \times situation$ which is defined as a partial order on situations.
- A binary predicate *Poss* of sort : $action \times situation$. The intended interpretation of $Poss(a, s)$ is that a can be executed in s.
- A finite number of predicate symbols for each sort $(action \cup object)^n$ and function symbols for each sort $(action \cup object)^n \rightarrow (action \cup object)$. These predicate and function symbols are situation independent.
- A finite number of predicate symbols of sort $(action \cup object)^n \times situation$. These predicate symbols are called *relational fluents*.

Note that only two function symbols - S_0 and *do* - are allowed to take values in sort *situation*. Also note that only the binary predicate $\sqsubset$ has more than one argument of sort *situation*. The language $\mathcal{L}_{sitcalc}$ is built from a given alphabet and the usual logical vocabulary, i.e.

equality, negation, conjunction, and the universal quantifier. The other logical connectives like $\vee, \rightarrow, \leftrightarrow$ and the existential quantifier $\exists$ are defined as the usual abbreviations. $\leftrightarrow$ is also written as $\equiv$.

Notice that we did not include *functional fluents* in the language of the situation calculus. Functional fluents are left out because of the particular *call-by-value* mechanism that is used as a parameter mechanism in ConGolog for procedure calls, which would *not* be state based in the presence of functional fluents. For our purposes, we are only interested in eliminating functional fluents; other types of function symbols are allowed.

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

Definition 2.2 *(uniform term, formula)*
Let S be any term of sort situation. Then the set T_S of terms *uniform in* S is inductively defined by:

- $S \in T_S$,
- if a term t does not mention a term of sort situation, then $t \in T_S$,
- 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:

- if $t_1, t_2 \in T_S$ are of the same sort, then $t_1 = t_2 \in \mathcal{L}_S$,
- if P is an n-ary predicate symbol, other than $Poss$ and $\sqsubset$, and $t_1, \ldots, t_n \in T_S$ are of the appropriate sorts, then $P(t_1, \ldots, t_n) \in \mathcal{L}_S$
- if $\varphi_1, \varphi_2 \in \mathcal{L}_S$, then $\neg\varphi_1, \varphi_1 \wedge \varphi_2 \in \mathcal{L}_S$
- if $\varphi \in \mathcal{L}_S$ and x is a variable not of sort situation, then $\forall x(\varphi) \in \mathcal{L}_S$.

A formula that is uniform in S does not mention the predicates $Poss$ or $\sqsubset$, nor does it quantify over situation variables. The only term of sort situation which can occur in a formula that is uniform in S is S itself. In a model M (and valuation ν) for $\mathcal{L}_{sitcalc}$, the true formulas which are uniform in S can be said to *characterise* situation S. In other words, these formulas *completely* specify the state denoted by S.

Notation 2.3 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.

2.2 Foundational Axioms

The basic intuitions associated with the notion of a situation are captured by a set of so called *foundational axioms* (cf. [12]). These axioms are listed below in definition 2.4. The first axiom below states that situations are uniquely identified by situation terms, and implies that situations can be identified with action histories. The second axiom is a second order axiom which captures the intuition that all the situations that exist are the ones reachable by doing a finite number of actions. The third axiom states that S_0 is the initial situation. And finally, the fourth axiom states that a situation s is a predecessor of a situation $do(a, s')$ iff s is a predecessor of s' or s and s' denote the same situation. Although these foundational axioms impose a basic structure upon the set of possible situations, the axioms do not play an important role in the definition of the programming language ConGolog (cf. also [12]).

Definition 2.4 *(foundational axioms)* [1]

$$do(a_1, s_1) = do(a_2, s_2) \rightarrow (a_1 = a_2 \wedge s_1 = s_2) \tag{1}$$
$$\forall P([P(S_0) \wedge \forall a, s(P(s) \rightarrow P(do(a, s))] \rightarrow \forall s(P(s)) \tag{2}$$
$$\neg s \sqsubset S_0 \tag{3}$$
$$s \sqsubset do(a, s') \equiv s \sqsubseteq s' \tag{4}$$

where $s \sqsubseteq s'$ abbreviates $s \sqsubset s' \vee s = s'$.

2.3 Basic Actions

A basic action theory in the situation calculus defines a framework for specifying the pre and postconditions of actions. Three types of axioms are introduced to specify actions. First of all, a set of *unique names axioms for actions* is introduced. These axioms are used to make sure that action names refer to different actions, and that an action symbol supplied with one set of parameters is distinguished from that action symbol supplied with a different set of parameters.

Definition 2.5 *(unique names axioms for actions)*
The set of *unique names axioms for actions* includes the following axioms:

$$A(\vec{x}) \neq B(\vec{y})$$

where $A(\vec{x})$ and $B(\vec{y})$ are expressions of sort *action*, and the set of variables $\vec{x}$ and $\vec{y}$ are disjoint, for each pair of action symbols A and B;
and for any action symbol A:

$$A(\vec{x}) = A(\vec{y}) \rightarrow \vec{x} = \vec{y}.$$

The second type of axioms are called *action precondition axioms*. These axioms specify when an action is *enabled*, i.e. they specify what preconditions must hold in order for an action to be executable in a situation. The uniformity condition on $\Pi_A(\vec{x}, s)$ is used to make sure that the preconditions of an action $A(\vec{t})$ depend only on the current situation s.

[1] The free variables in formulas which occur in definitions throughout this paper are implicitly universally quantified.

Definition 2.6 *(action precondition axiom)*
An *action precondition axiom* is of the form:

$$Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$$

where $A(\vec{x})$ is an expression of sort *action*, and $\Pi_A(\vec{x}, s)$ is a formula that is uniform in s and whose free variables are among $\vec{x}, s$.

The last type of axioms are called *successor state axioms*. Successor state axioms relate the value of a fluent in the situation that results from doing an action to their value in the previous situation, and define the effects of executing an action. Successor state axioms also provide a solution to the frame problem [15]. The uniformity condition on $\Phi_F(\vec{x}, a, s)$ guarantees that the database associated with the successor situation (the database of uniform formulas that hold in that situation) can be computed from that of the previous situation.

Definition 2.7 *(successor state axiom)*
A *successor state axiom* for a *relational* fluent F is 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 and whose free variables are among $\vec{x}, a, s$.

A basic action theory is a collection of the axioms introduced so far. Our definition of a basic action theory slightly differs from the one in [9]. The main difference is that we do not include the initial situation axioms or the *initial database* in the action theory.

Definition 2.8 *(basic action theory)*
A *basic action theory* is a theory $\mathcal{A} = \Sigma \cup \mathcal{A}_{ss} \cup \mathcal{A}_{ap} \cup \mathcal{A}_{una}$ where:

- Σ are the *foundational axioms*,
- $\mathcal{A}_{ss}$ is a set of *successor state axioms* for relational fluents, one for each fluent,
- $\mathcal{A}_{ap}$ is a set of *action precondition axioms*, one for each action symbol,
- $\mathcal{A}_{una}$ is a set of *unique names axioms*, for all pairs of action function symbols.

Definition 2.9 *(initial database)*
An *initial database* is a finite set of (first order) formulas from $\mathcal{L}_{sitcalc}$ that are uniform in S_0.

2.4 Situations, States and Functional Fluents

The reason for introducing the uniformity conditions into basic action theories is to ensure that the evaluation of preconditions and successor state conditions depends only on the current situation. In the presence of functional fluents, however, the uniformity conditions are not enough to guarantee that the preconditions and successor state conditions depend only on the current situation. It is not difficult to give an example in which a functional fluent is substituted for a parameter and results in a violation of a uniformity condition. For example, if we substitute $loc(Ball, do(throw(Ball), S_0))$ for x and 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. In the presence of functional fluents, therefore, we have to be more careful and only substitute terms that do not lead to violations of the uniformity conditions.

The fact that all conditions can be evaluated by inspection of the current situation only implies that during a computation only a database of facts that talk about the current situation has to be maintained. This is a typical feature of a *state based approach*. The main characteristic of a state based approach is that a successor state can be computed from the current state and the action that is performed in that state. Because of the particular form of successor state axioms, basic action theories also support a state based approach, with the proviso that substitution of functional fluents does not lead to violations of the uniformity conditions.

The uniformity conditions thus play an important role in basic action theories. In general, there is an important difference between situations in the situation calculus and states in state based approaches. Whereas a state coincides with a single point in (space-)time, a situation (action history) can be much more complex. A situation s can even refer to would-be situations in a possible history that is different from the actual one referred to by s. For example, consider the situation $do(goto(loc(Ball, do(trow(Ball), S_0)), S_0)$. This situation refers to the situation resulting from going to a particular place in the initial situation S_0. The place referred to needs to be inferred from doing *another* action in situation S_0, namely the action of throwing a ball. For the evaluation of a formula like $corner(loc(Robot, do(goto(loc(Ball, do(trow(Ball), S_0)), S_0))$ we thus have to inspect the would-be situation resulting from throwing the *Ball* in situation S_0, and check if the location of the *Ball* in *that* situation is a corner, assuming that a goto action always succeeds. Due to the possibility of a branching structure of situations we can construct such 'non-linear' situations which depend on other situations in different branches in the possible histories structure.

The example of the previous paragraph used functional fluents to illustrate that situations are different from states. Still another feature in the situation calculus, that of quantification over situations, can give rise to formulas that refer to different situations. An example of such a formula is the following precondition axiom: $Poss(open(d), s) \equiv \exists\, s'(s = do(unlock(d, key), do(get(key), s')))$. This formula states that it is only possible to open a door if a key has been obtained and the door is unlocked with this key in the last *two* situations. Because this precondition refers to the two previous situations, it cannot be evaluated by inspecting the current situation only.

Summarising, in general the situation calculus offers an expressive framework for talking about action histories. The basic action theories that we introduced restrict this expressivity by introducing uniformity conditions, and thus provide for a state based approach. Still, we have to be careful in the presence of functional fluents. Because 3APL is a state based formalism, for the purpose of bisimulating ConGolog, it is important that basic action theories are state based. This is our main reason for excluding functional fluents.

3 The High-Level Programming Language ConGolog

ConGolog is a logic programming language based on the situation calculus. It extends the basic action theories of the previous section with operators for constructing complex actions. In ConGolog it is possible, for example, to specify the sequential composition of two actions, like, $pickup(Block)$; $putaway(Block)$. The set of ConGolog programs is defined below. It is a subset of all the programs as in [2], but includes the main programming constructs. Most of

the programming constructs below are well-known. Tests evaluate a formula in the current situation. The nondeterministic choice of argument construct nondeterministically selects a value for the variable x. In a prioritised parallel program $\delta_1 \rangle\!\rangle \delta_2$ the execution of the left subprogram δ_1 is preferred over that of the right subprogram δ_2; the latter is executed only if δ_1 cannot be executed.

Definition 3.1 *(ConGolog programs)*
The set of *open* programs P and procedures *Proc* 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, [2]
- *parallel composition:* $\delta_1 \| \delta_2 \in P$, if $\delta_1, \delta_2 \in P$,
- *prioritised parallel composition:* $\delta_1 \rangle\!\rangle \delta_2 \in P$, if $\delta_1, \delta_2 \in P$,
- *procedure call:* $P(\vec{t})$,
- *procedure definition:*
  proc $P(\vec{x})$ δ_P **end** $\in$ *Proc*, if $\delta_P \in P$ such that all free variables in δ_P occur in $\vec{x}$.

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

The constructs which are defined in [2] but are not included in the definition above are iteration, synchronised if-then-else, synchronised while, and parallel iteration. As far as iteration is concerned, no expressivity is lost, since it is well-known that this construct can be simulated by recursive procedures which are included in definition 3.1. The synchronised if-then-else and the synchronised while are slight variations on the non-synchronised ones, and require the atomic execution of both the test as well as the first action of one of the branches of the if-then-else or of the body of the while-construct. That is, in both constructs both the test and the first action to be executed next are executed in one single step. 3APL does not have similar constructs that are synchronised in this way. It would not be difficult to extend 3APL with these constructs, but doing so would not lead to any new or interesting results with respect to the simulation of ConGolog in 3APL. A similar remark applies to parallel or concurrent iteration.

[2] Variables of sort action can be simulated if there are only a finite number of actions available. An example of the use of action variables is given in [2].

3.1 Axiomatic Definition of the Semantics for ConGolog

The meaning of the ConGolog programming constructs is specified by using a transition semantics that is presented in a non-standard way. Instead of using a formalism like SOS semantics [13], a new predicate *Trans* is added to the language of the situation calculus and is used to formalise the step semantics of a program. The predicate $Trans(\delta, s, \delta', s')$ expresses that it is possible for program δ to perform a computation step in situation s that results in a new situation s' where δ' is the remaining program that still needs to be executed. The semantics of ConGolog programs is specified by means of a set of axioms for the predicate *Trans*. For each programming construct, there is an axiom that states which computation steps the construct allows. The expression *nil* denotes the 'empty' program, and is used below as an auxiliary construct in the definition of the operational semantics. *nil* is *not* a ConGolog program. Also notice that no transition is associated with the nondeterministic selection of a value in a $\pi x.\delta$ program, but only with the joined action of selecting a value and executing a step of subprogram δ. In the axiomatic definition of *Trans*, the predicate $Final(\delta, s)$ is used to express that program δ may legally terminate in situation s. A formal definition of *Final* is presented after the definition of *Trans*. The definition of the semantics of (recursive) procedures is postponed until the next section.

Definition 3.2 *(axioms for Trans)* [3]
Trans is inductively defined by:

- The Empty Program:

$$Trans(nil, s, \delta', s') \equiv False$$

- Basic Actions:

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

- Tests:

$$Trans(\phi?, s, \delta', s') \equiv \phi[s] \wedge \delta' = nil \wedge s' = s$$

- Sequential Composition:

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

- Nondeterministic Choice:

$$Trans(\delta_1 \mid \delta_2, s, \delta', s') \equiv Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s')$$

[3] Formally, an encoding of ConGolog programs into terms of the first order language $\mathcal{L}_{sitcalc}$ is required, as is done in [2]. However, because the details in this paper - apart from the encoding itself - are almost completely the same, for notational convenience we use the programs as in definition 3.1 and refer the reader to [2] for the details concerning the encoding of programs into terms.

- Nondeterministic Choice of Argument:

$$Trans(\pi x.\delta, s, \delta', s') \equiv \exists\, x.\, Trans(\delta, s, \delta', s')$$

- Parallel Composition:

$$\begin{array}{l} Trans(\delta_1 \| \delta_2, s, \delta', s') \equiv \\ \quad \exists\, \gamma.\delta' = (\gamma \| \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee \\ \quad \exists\, \gamma.\delta' = (\delta_1 \| \gamma) \wedge Trans(\delta_2, s, \gamma, s') \end{array}$$

- Prioritised Parallel Composition:

$$\begin{array}{l} Trans(\delta_1 \rangle\!\rangle \delta_2, s, \delta', s') \equiv \\ \quad \exists\, \gamma.\delta' = (\gamma \rangle\!\rangle \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee \\ \quad \exists\, \gamma.\delta' = (\delta_1 \rangle\!\rangle \gamma) \wedge Trans(\delta_2, s, \gamma, s') \wedge \neg\, \exists\, \eta, s''.\, Trans(\delta_1, s, \eta, s'') \end{array}$$

Definition 3.3 *(axioms for Final)*
The *Final* predicate is defined by the following set of axioms:

- The Empty Program:

$$Final(nil, s) \equiv True$$

- Basic Actions:

$$Final(a, s) \equiv False$$

- Tests:

$$Final(\phi?, s) \equiv False$$

- Sequential Composition:

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

- Nondeterministic Choice:

$$Final(\delta_1 \mid \delta_2, s) \equiv Final(\delta_1, s) \vee Final(\delta_2, s)$$

- Nondeterministic Choice of Argument:

$$Final(\pi x.\delta, s) \equiv \exists\, x.Final(\delta, s)$$

- Parallel Composition:

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

- Prioritised Parallel Composition:

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

3.2 ConGolog Procedures

The semantics of ConGolog procedures is not defined in terms of replacement of the procedure *call* with the procedure *body*, since such steps are not viewed as transitions in the ConGolog semantics (cf. [2]). Instead, a second order definition of the transition predicate is given which abstracts from these steps. A procedure call in the ConGolog semantics involves both body replacement of a (number of) procedure call(s) and the execution of an action or test in a single step. Only actions or tests can give rise to transitions in the ConGolog semantics, and replacement of a procedure call with its associated body or the nondeterministic selection of a value in a $\pi x.\delta$ program are not viewed as transitions.

In ConGolog, *nesting of procedure definitions* is allowed, and it is important to keep track of the scope of a procedure definition. Nesting of procedure definitions, however, can be considered as syntactic sugar (a pre-compiler easily removes all naming conflicts such that global scope can be assumed), and we do not consider this facility here. The implementation of ConGolog also does not include this feature (cf. [2]). In the absence of procedure nestings, the second order definition of *Trans* that also deals with procedures can be defined by

$$Trans(\delta, s, \delta', s') \equiv \forall\, T.(\varphi(T, \delta, s, \delta', s') \rightarrow T(\delta, s, \delta', s'))$$

where $\varphi(T, \delta, s, \delta', s')$ is the conjunction of the set of axioms for *Trans* of the previous section, with T substituted for *Trans*, and the following clause for procedure calls:

$$T(P(\vec{t}), s, \delta', s') \equiv T((\delta_P)^{\vec{x}}_{\vec{t}}, s, \delta', s').$$

In the clause for procedure calls, δ_P is the body of the procedure definition of $P(\vec{x})$ and $(\delta_P)^{\vec{x}}_{\vec{t}}$ is that same body where the formal parameters $\vec{x}$ have been substituted with $\vec{t}$. This second order definition defines *Trans* as the smallest set of transitions closed under the set of clauses for the ConGolog programming constructs defined in the previous section and the clause for procedure calls introduced in this section.

Similarly, *Final* is defined by:

$$Final(\delta, s) \equiv \forall\, F.(\psi(F, \delta, s) \rightarrow F(\delta, s))$$

where $\psi(F, \delta, s)$ is the conjunction of the set of axioms for *Final* of the previous section, with F substituted for *Final*, and the following clause for procedure calls:

$$F(P(\vec{t}), s) \equiv F((\delta_P)^{\vec{x}}_{\vec{t}}, s).$$

The parameter mechanism of ConGolog is a *call-by-value* mechanism, which, due to the assumption that functional fluents are absent, could be somewhat simplified. In the presence of functional fluents, the substitution of the actual parameters is slightly more complex and should be $(\delta_P)^{\vec{x}}_{\vec{t}[s]}$ instead of simply $(\delta_P)^{\vec{x}}_{\vec{t}}$. That is, functional fluents in an actual parameter term t should be evaluated with respect to the current situation s, and to implement this the situation s is substituted for the constant *now*. After substituting s for *now* in parameter t, the parameter $t[s]$ is substituted in the body of the procedure at the appropriated places. We are now in a position to explain in detail our remark above that this type of parameter mechanism may result in an approach that is not state based, if functional fluents are allowed. Consider, for example, the procedure definition **proc** *serve*(n) *go_floor*(n); *turnoff*(n); *open*; *close* **end** and the procedure call

$serve(nearest\_floor(now))$. Notice that the actual parameter in the procedure call is a functional fluent. Now suppose that the current situation is the initial situation S_0. In that case, we get the following transition as a result of the execution of the call:

$$Trans(serve(nearest\_floor(now)), S_0,$$
$$turnoff(nearest\_floor(S_0));\ open, do(go\_floor(nearest\_floor(S_0)), S_0))$$

The remaining program is $turnoff(nearest\_floor(S_0));\ open$ which must be executed in situation $do(go\_floor(nearest\_floor(S_0)), S_0)$. For the evaluation of the argument $nearest\_floor(S_0)$, however, the program has to consult the *previous* situation S_0 instead of the current one. The call-by-value mechanism thus is not compatible with a strictly state based approach in the presence of functional fluents.

The second order semantics abstracts from all steps in which a procedure call is replaced with its body and only actions and tests are viewed as transitions. As a consequence, a procedure call $P(\vec{t})$ that never gives rise to the execution of an action or test, can never give rise to any transition. Formally, $\forall\, \delta', s'. \neg Trans(P(\vec{t}), s, \delta', s')$. This is quite different from a semantics that associates a transition with body replacement as is done in the semantics for 3APL. Consider, for example, the procedure definition **proc** $d(n)$ $(n = 1)? \mid d(n-1);\ go\_down$ **end** and the program $d(0) \mid \text{true}?$. According to the ConGolog semantics, the only transition this program can (always) make is the transition in which the test is executed, because the procedure call $d(0)$ never gives rise to the execution of an action or test. This program thus always successfully executes and terminates after executing the test in the ConGolog semantics. In the 3APL semantics, this is not the case. Since body replacement also is a legal computation step, the left branch may be selected and a body replacement may occur. The selection of the left branch, however, results in a non-terminating computation where in each step a body replacement is performed. In the 3APL semantics, the program thus has a non-terminating computation in contrast with the ConGolog semantics.

Although the behaviour in accordance with the ConGolog semantics may be preferred over that of a semantics which includes steps for body replacement that gives rise to a non-terminating computation, there is a computational problem. To implement the second order semantics for procedure calls, an algorithm which decides if such a call results in the eventual execution of an action or test is required. The problem is that such an algorithm does not exist, since that algorithm would also solve the halting problem. In our example, non-termination may seem easy to detect, but in general it is not possible to decide this type of termination for arbitrary actual parameters (which may involve complex terms). The extension of an operational style semantics in first order logic with second order axioms in this case thus results in a non-computational semantics.

Due to the fact that ConGolog and 3APL assign different semantics to procedure calls, it is not possible to construct an embedding of ConGolog into 3APL. An embedding, however, can be constructed for a large and interesting subclass of ConGolog procedure definitions. In particular, the set of *guarded procedures* is a class of procedures that never does more than a fixed number of procedure calls before executing an action or test. This class thus avoids the problem of detecting termination as in the general case. This subset also can be embedded into 3APL.

A formal definition of a *guarded program* is provided by means of the notion of a *rank*. This notion is similar to that in [2], except for one important difference. In contrast to the definition in [2], our notion of rank is *not* dependent on the current situation, but is completely

syntactic and therefore somewhat simpler. [4]

Definition 3.4 *(rank)*
The *rank* n (n a natural number) of a program δ (possibly containing free variables) is defined by the following axioms:

$$\begin{aligned}
Rank(n, nil) &\equiv True \\
Rank(n, a) &\equiv True \\
Rank(n, \phi?) &\equiv True \\
Rank(n, \delta_1;\ \delta_2) &\equiv Rank(n, \delta_1) \wedge Rank(n, \delta_2) \\
Rank(n, \delta_1 \mid \delta_2) &\equiv Rank(n, \delta_1) \wedge Rank(n, \delta_2) \\
Rank(n, \pi x.\delta) &\equiv Rank(n, \delta) \\
Rank(n, \delta_1 \| \delta_2) &\equiv Rank(n, \delta_1) \wedge Rank(n, \delta_2) \\
Rank(n, \delta_1 \rangle\!\rangle \delta_2) &\equiv Rank(n, \delta_1) \wedge Rank(n, \delta_2) \\
Rank(n, P(\vec{t})) &\equiv Rank(n-1, \delta_P{}^{\vec{x}}_{\vec{t}})
\end{aligned}$$

A ConGolog program δ is *guarded* iff δ is of rank n for some n, i.e. $Guarded(\delta) \stackrel{df}{=} \exists\, n.Rank(n, \delta)$. This notion of guardedness is strictly stronger than the one defined in [2], because it does not depend on the current situation. As a consequence, Theorem 6 in [2] holds, and we can define the semantics of procedures by a set of first order axioms. In the sequel, we will prove an embedding result for *guarded programs* which is based on the first order semantics of the previous section and that of definition 3.5 below. It should be understood that the semantics of definition 3.5 only applies to guarded programs (procedure definitions).

Definition 3.5 *(first order axioms for procedure calls)*

$$\begin{aligned}
&Trans(P(\vec{t}), s, \delta', s') \equiv Trans(\delta_P{}^{\vec{x}}_{\vec{t}}, s, \delta', s') \\
&Final(P(\vec{t}), s) \equiv Final(\delta_P{}^{\vec{x}}_{\vec{t}}, s)
\end{aligned}$$

4 The Agent Programming Language 3APL

3APL is an agent programming language that combines imperative programming and logic programming. From imperative programming the language inherits the full range of regular programming constructs, including recursive procedures, and a notion of state based computation. States of agents, however, are belief or knowledge bases, which are different from the usual variable assignments of imperative programming, and the assignment statement of imperative programming is replaced with updates on the belief base of an agent. From logic programming, the language inherits the proof as computation model as a basic means of computation for querying the belief base of the agent. We want to emphasise that 3APL is a *programming* language, and not a *logical* language like the situation calculus.

The agent language 3APL is based on a rich notion of agents. That is, agents have a *mental state* consisting of *beliefs* and *goals*. Associated with an agent are a number of basic

[4] As a consequence, if a procedure call is guarded for all procedures in a program δ, we also know that a program has a 'guarded evolution'; that is, any program resulting from the execution of any number of steps of δ is guarded again. This is our analogue of Theorem 7 in [2].

capabilities. The basic capabilities of an agent are the basic actions an agent can perform, and can be viewed as defining the *expertise* of the agent. Finally, an agent can have a number of *practical reasoning rules* for planning and modifying its current goals.

The beliefs of an agent are drawn from some knowledge representation language $\mathcal{L}$. In principle, the choice of knowledge representation language is free, and any formalism which allows the derivation of facts from the agent's beliefs can be used to program agents. For the purpose of simulating ConGolog, it is convenient to identify the knowledge representation language $\mathcal{L}$ with the language $\mathcal{L}_{now}$ of situation calculus formulas uniform in *now*. Subsets of these formulas are used to represent the current situation.

The goals of a 3APL agent are plans, or imperative programs like in ConGolog. These goals are built from basic actions, tests, and the same programming constructs as in ConGolog, apart from some minor differences in notation. So called achievement goals correspond to procedure calls of ConGolog.

Definition 4.1 *(goal)*
The set of goals Goal is defined by:

- Basic actions: $A(\vec{t}) \in \mathsf{Goal}$,
- Tests: $\phi? \in \mathsf{Goal}$, if $\phi \in \mathcal{L}_{now}$,
- Achievement Goals: $P(\vec{t}) \in \mathsf{Goal}$,
- Sequential Composition: $\pi_1;\ \pi_2 \in \mathsf{Goal}$, if $\pi_1, \pi_2 \in \mathsf{Goal}$,
- Nondeterministic Choice: $\pi_1 + \pi_2 \in \mathsf{Goal}$, if $\pi_1, \pi_2 \in \mathsf{Goal}$,
- Parallel Composition: $\pi_1 \| \pi_2 \in \mathsf{Goal}$, if $\pi_1, \pi_2 \in \mathsf{Goal}$,

Originally, 3APL does not include a construct for prioritised parallel composition $\rangle\!\rangle$ ([6, 7]), but in the sequel we show how to formally define a semantics for $\rangle\!\rangle$ in a transition style semantics, which is the type of semantics used to specify the operational semantics of 3APL, and extend 3APL with this operator.

One of the more important differences between ConGolog and 3APL is the presence of the π operator in ConGolog and the absence of such a construct in 3APL. Correspondingly, the two languages have quite different parameter mechanisms. Whereas in ConGolog the π operator is used to nondeterministically select a value for a variable that is *bound* by the operator, in 3APL tests are used to compute values for *free* variables as in logic programming. Moreover, whereas the π operator provides for an explicit scoping mechanism, the use of free variables in 3APL is based on implicit scoping and involves the renaming of free variables when procedures calls are replaced with their corresponding body. To facilitate the construction of an embedding of ConGolog into 3APL and to accommodate for these different styles of parameter passing, we introduce an additional construct $random(x)$ which, like the πx operator, nondeterministically selects a value for the variable x. $random(x)$ does not introduce any additional expressivity into 3APL and can be viewed as syntactic sugar.

A practical reasoning rule in 3APL has the form $\pi \leftarrow \phi \mid \pi'$, where π, π' are goals and ϕ is a formula from $\mathcal{L}_{now}$, the knowledge representation language we use here. In a rule $\pi \leftarrow \phi \mid \pi'$, π is called the *head* of the rule, π' the *body* of the rule, and ϕ the *guard* of the rule. When the head π is specialised to an achievement goal $P(\vec{x})$ and the guard is identified

with true, we obtain rules which correspond with the recursive procedures of ConGolog. The head and body may also contain *goal variables*. For example, $X;\ \phi? \leftarrow \psi \mid A$ is a legal rule which replaces anything (matching the goal variable X) that is sequentially composed with a particular test $\phi?$ and this test itself with the simple action A in case the agent believes that ψ holds in the current state. For a more extensive treatment of practical reasoning rules we refer the reader to [6, 7]. For our purposes, we only need simple rules of the form $P(\vec{x}) \leftarrow \pi$ (a guard that is equivalent with true is not mentioned).

A 3APL agent is defined as a tuple $\langle \pi, \sigma, \mathcal{R} \rangle$, where π is a goal, σ is a set of beliefs, and $\mathcal{R}$ is a set of (simple) rules of the form $P(\vec{x}) \leftarrow \pi$.

4.1 Semantics of 3APL

The operational semantics of 3APL is defined by means of a Plotkin-style transition system ([13]). Such a transition system consists of a set of transition rules which define the computation steps associated with individual constructs of the language. Transition rules are a kind of derivation rules, which can be used to derive transitions from a set of given transitions. A transition system specifies a transition relation $\longrightarrow$ and can be viewed as an inductive definition of this relation $\longrightarrow$. The relation $\longrightarrow$ defines the possible *computation steps* a program can perform and is the analogue of the *Trans* predicate of the ConGolog semantics for 3APL.

A Labelled Transition Relation For the simulation of ConGolog, it is important to keep track of the *sequence of basic actions* which are executed during a computation of a program. This information is explicitly represented by the situation arguments of the *Trans* predicate in ConGolog. To represent the same information in the 3APL semantics, we define a *labelled transition relation*. Labels are associated with a transition and indicate whether an action or something else has been performed.

Labels are also used to distinguish between two types of transitions in 3APL. From the point of view of the ConGolog semantics only the execution of basic actions or tests give rise to a transition. As we will see below, 3APL associates a transition with the expansion of a procedure call into its body and with the execution of a *random* action for nondeterministically selecting values. We also use labels to distinguish between 3APL transitions that count as transitions in the ConGolog semantics too and the ones that do not count as such. The latter type of transitions we call *silent steps*. The intuition is that these steps are not 'visible' in the ConGolog semantics and from the ConGolog perspective are considered to be implementation details.

The labelling is derived from these intuitions. A transitions that corresponds with the execution of a basic action is labelled with that same basic action. A transitions that corresponds with the execution of a test is labelled with the special symbol ϵ, the empty sequence. The empty sequence is used to denote that no action has been performed, and the situation has not been changed. Both the execution of a *random* action and the expansion of a procedure call into its body are labelled with i to indicate that an internal or silent step has been performed. The labelling of complex programs are derived from the more basic ones. For example, the label associated with the execution of a sequential composition $\pi_1;\ \pi_2$ is derived from the label that is associated with the execution of π_1.

3APL Configurations The transition relation $\longrightarrow$ is a relation on pairs $\langle \pi, \sigma \rangle$, with π a goal and σ a belief base. These pairs $\langle \pi, \sigma \rangle$ are also called *configurations*. In the transition

rules below, we use E to denote (successful) termination, and identify $E;\ \pi$, $E\|\pi$, and $\pi\|E$ with π.

During a computation a 3APL program may compute bindings for (free) variables in the program by means of tests. These bindings are recorded in substitutions which are associated with the transition relation $\longrightarrow$ for later reference. Substitutions thus provide for a parameter mechanism of 3APL and are used to instantiate free variables in the remaining program (after such a substitution has been computed). $\varnothing$ denotes the empty substitution.

The programming language 3APL does not fix a specific set of basic actions, and as a consequence is an *abstract* programming language. This approach to agent programming assumes that a transition function $\mathcal{T}$ must also be provided to specify the semantics of the basic actions used by agents (similar to specifying successor state axioms for ConGolog programs). The semantics of a basic action in this abstract setting is only fixed in the sense that a basic actions is interpreted as a change to the mental state of the agent, in particular the beliefs of the agent. The execution of a basic action thus amounts to changing this state according to the transition function $\mathcal{T}$.

Definition 4.2 *(transition rule for basic actions)*

$$\frac{\mathcal{T}(A(\vec{t}),\sigma) = \sigma'}{\langle A(\vec{t}),\sigma\rangle_V \xrightarrow{A(\vec{t})}_{\varnothing} \langle E,\sigma'\rangle}$$

A *test* $\phi?$ allows an agent to introspect its beliefs. A test is evaluated relative to the current beliefs of an agent. Their main use, however, is not just to check whether or not the agent believes a particular proposition, but to compute values or bindings for the free variables which occur in the test. In this respect a test is similar to an assignment in imperative programming. The main difference is that a test is evaluated by means of logical proof (as in logic programming), and a test can only be used to *initialise* a variable to some value, not to update the value assigned to a variable. The values that are computed are recorded in a substitution γ, and the bindings computed are used in computation steps in the remaining computation of the program.

Definition 4.3 *(transition rule for tests)*
Let γ be a ground substitution such that $dom(\gamma) = \mathsf{free}(\phi)$.

$$\frac{\sigma \models \phi\gamma}{\langle \phi?,\sigma\rangle_V \xrightarrow{\epsilon}_{\gamma} \langle E,\sigma\rangle}$$

A $random(x)$ action, like a test, computes a binding for the variable x. In contrast with arbitrary tests, however, $random(x)$ always succeeds and nondeterministically returns an arbitrary binding for x. The $random(x)$ action does not increase the expressivity of 3APL, since it can be defined as a special kind of test. For an arbitrary unary predicate symbol P, $random(x)$ can be defined as: $random(x) \stackrel{df}{=} (P(x) \vee \neg P(x))?$. The reason for introducing $random(x)$ as an explicit action is that we want to label this particular action as a silent step. $random(x)$ is used to simulate the pick operator π, which does not give rise to a transition. The nondeterministic selection of a value thus is considered as an implementation detail in the ConGolog semantics, and therefore needs to be modelled as a silent step in the 3APL semantics.

Definition 4.4 *(transition rule for random)*
Let t be a ground term.

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

A sequence of two programs is executed by executing the first program and by updating the results of the computation, that is, updates on the beliefs or bindings for variables, correspondingly, where computed values for variables are passed on to the rest of the program by instantiation.

Definition 4.5 *(transition rule for sequential composition)*

$$\frac{\langle \pi_1, \sigma\rangle_{V \cup \mathsf{free}(\pi_2)} \xrightarrow{l}_\gamma \langle \pi_1', \sigma'\rangle}{\langle \pi_1;\ \pi_2, \sigma\rangle_V \xrightarrow{l}_\gamma \langle \pi_1';\ \pi_2\gamma, \sigma'\rangle}$$

The execution of a nondeterministic choice goal consists in selecting one of the subgoals that is enabled, i.e. can be executed, execute this goal, and drop the other goal.

Definition 4.6 *(transition rule for nondeterministic choice)*

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

Parallel execution is modelled by interleaving. The parallel subgoals may communicate with each other through the belief base (a shared data base) and shared variables. If one of the subgoals retrieves data from the belief base, the substitution so obtained is also applied to the other parallel subgoal.

Definition 4.7 *(transition rule for parallel composition)*

$$\frac{\langle \pi_1, \sigma\rangle_{V \cup \mathsf{free}(\pi_2)} \xrightarrow{l}_\gamma \langle \pi_1', \sigma'\rangle}{\langle \pi_1 \| \pi_2, \sigma\rangle_V \xrightarrow{l}_\gamma \langle \pi_1' \| (\pi_2\gamma), \sigma'\rangle} \qquad \frac{\langle \pi_2, \sigma\rangle_{V \cup \mathsf{free}(\pi_1)} \xrightarrow{l}_\gamma \langle \pi_2', \sigma'\rangle}{\langle \pi_1 \| \pi_2, \sigma\rangle_V \xrightarrow{l}_\gamma \langle (\pi_1\gamma) \| \pi_2', \sigma'\rangle}$$

Practical reasoning rules operate on the goals of the agent. A simple practical reasoning rule of the form $P(\vec{t}) \leftarrow \varphi \mid \pi_b$ is applicable if the head of the rule unifies with a (subgoal of a) current goal of the agent and the guard is entailed by the current beliefs. The notion of a *variant* plays an important role in the semantics of rules. An expression e is a variant of another expression e' in case e can be obtained from e' by renaming of variables and explains the presence of the set of variables V in transition rules. The set V is used to make sure that a variant is chosen such that no interference with free variables in the remaining goal can occur (for details see [6, 7], for a formal definition of variants, cf. [10]).

Definition 4.8 *(transition rule for rule application)*
Let η be a most general substitution such that $P(\vec{t}) = P(\vec{t'})\eta$, and γ be a ground substitution such that $dom(\gamma) = \mathsf{free}(\phi\eta)$.

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

where $P(\vec{t'}) \leftarrow \phi \mid \pi_b$ is a *variant* of a PR-rule in the PR-base $\mathcal{R}$ of the agent such that no free variables in the rule occur in V.

4.2 Silent Steps

As we explained above, in the semantics of ConGolog a transition is associated with a program only if it can perform an action or test. In 3APL, however, a transition is associated also with the expansion of a call into its body and with the execution of a *random* action. For this reason, we introduced a distinction between two types of computation steps in 3APL. Computation steps due to the execution of a basic action or a test are distinguished from other types of computation steps. The latter are called *silent steps*.

For the construction of an embedding of 3APL into ConGolog, we want to abstract from these silent steps. For this reason, we introduce a new transition relation $\Longrightarrow$ for 3APL programs that is derived from the transition relation $\longrightarrow$. The transition relation $\Longrightarrow$ induces a new step relation. $\Longrightarrow$ steps are composed of an arbitrary number of silent steps $\xrightarrow{i}$ followed by a single step that involves the execution of a basic action or a test.

In the definition below, * denotes the transitive closure of a relation. The relation $\longrightarrow^*_\gamma$ is defined as the set of all finite (including empty) sequences of $\longrightarrow$ steps and γ is defined as the subsequent application of the substitutions associated with each of these steps. That is, if $\longrightarrow_{\gamma_1} \longrightarrow_{\gamma_2} \ldots \longrightarrow_{\gamma_n}$ denotes a legal sequence of steps then $\gamma = \gamma_1\gamma_2 \ldots \gamma_n$ is associated with the single step obtained by taking the transitive closure.

Definition 4.9 *(abstracting from internal steps)*
The transition relation $\stackrel{l}{\Longrightarrow}$, where l is either $A(\vec{t})$ or ϵ, is defined by:

$$\stackrel{A(\vec{t})}{\Longrightarrow}_\gamma \quad \stackrel{df}{=} \quad \xrightarrow{i}{}^*_\gamma \cdot \xrightarrow{A(\vec{t})}_\varnothing$$

$$\stackrel{\epsilon}{\Longrightarrow}_{\gamma\theta} \quad \stackrel{df}{=} \quad \xrightarrow{i}{}^*_\gamma \cdot \xrightarrow{\epsilon}_\theta$$

In the sequel, we also just write $\Longrightarrow$ or $\longrightarrow$ instead of $\Longrightarrow_\gamma$ or $\longrightarrow_\gamma$ in case substitutions are not important in the context.

5 Operationalising ConGolog

In this section, we discuss a number of distinguishing features of the ConGolog and 3APL semantics. To be able to construct an embedding of ConGolog into 3APL, a number of issues have to dealt with. First, we discuss the semantics of tests. From this discussion, we derive a requirement on the initial database or belief base of an agent. Secondly, we discuss the semantics of nondeterministic selection of a value by the π operator. We conclude that a domain closure - or similar - assumption is needed to operationalise this operator. These discussions show a difference between ConGolog and 3APL due to the different formalisms used to define their respective semantics. The ConGolog semantics offers a logical definition of an agent system, whereas the 3APL semantics offers a more operational or computational definition of agent systems. The logical semantics of ConGolog raises a number of issues as to how to operationalise or implement it. Next, we proceed to show how to derive an update semantics for 3APL actions from the successor state axioms provided by a basic action theory $\mathcal{A}$. And finally, we define a translation function τ that maps ConGolog programs to equivalent 3APL goals. In the next section, we then prove that the translation function τ defines an embedding of ConGolog into 3APL.

5.1 Operationalising Tests and Complete Theories

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 $\delta = (\phi?;\ A) \mid (\neg\phi?;\ A)$. In 3APL, δ is not always enabled because it is possible that neither ϕ nor its negation $\neg\phi$ is entailed by the current beliefs of the agent. In contrast, δ is enabled in any situation s in ConGolog and, if A is also enabled in situation s, results in a final situation $do(A, s)$. δ is enabled simply because either ϕ or $\neg\phi$ must hold in the current situation (we may assume that ϕ is closed since ConGolog programs are closed).

Program δ of the previous paragraph is a simple example that illustrates the difference between ConGolog and 3APL tests. The program δ, however, is a special kind of program since it does not raise the question whether or not the left or right branch of the program should be executed. Either way, action A must be executed. In case we replace A in both branches with different subprograms, however, we do need to determine which branch to execute. Consider the program $\delta' = \phi?;\ A \mid \neg\phi?;\ B$. The ConGolog semantics implies that δ' is enabled in any situation, and one of the tests in one of the branches should be executed. Due to *incomplete information* about the current situation, however, it may be impossible to *decide* which branch needs to be executed.

To give still another example, in case the initial database does not contain any information about the proposition P, the ConGolog semantics does not specify the behaviour of program $P?$. The logical semantics allows models in which $P?$ can be executed and the program terminates successfully, and models in which $P?$ is not enabled and the program never terminates successfully. In contrast, the 3APL semantics in such a case precisely specifies the behaviour of the program since we always have $\sigma \models P$ or $\sigma \not\models P$ for arbitrary belief bases σ and propositions P.

The ConGolog semantics of tests thus raises the issue how to operationalise or implement it, since, as is illustrated by the examples, the ConGolog semantics does not always completely specify the behaviour of a program with tests. A possible solution for this problem is to require that (initial) databases be *complete*. A complete database σ decides every sentence of a language, and as a consequence if $\sigma \models \phi \vee \psi$ we also have that $\sigma \models \phi$ or $\sigma \models \psi$ for any ϕ, ψ. As a consequence, the two decision problems, evaluating whether ϕ or $\neg\phi$ holds in the current situation, or $\sigma \models \phi$ or $\sigma \models \neg\phi$, coincide. In our case, it is essential to be able to evaluate arbitrary sentences uniform in a particular situation and therefore we require that the current database associated with a particular situation is complete.

Definition 5.1 *(complete theories)*
Let $\sigma \subseteq \mathcal{L}$. σ is called *complete* iff for every sentence φ in $\mathcal{L}$ either $\sigma \models \varphi$ or $\sigma \models \neg\varphi$.

Lemma 5.2 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 $\mathcal{A}$,

$$\mathcal{A} + \sigma[S] \models \varphi[S] \text{ iff } \sigma \models \varphi$$

Proof: Immediate, since $\varphi[S]$ is uniform in S and σ is a complete theory. ■

5.2 Operationalising the π Operator and Domain Closure

Another interesting difference concerns the parameter mechanism in ConGolog and 3APL. Whereas ConGolog has an explicit operator πx which binds variables in a program (only closed programs are ConGolog programs!), the parameter mechanism in 3APL is based on an implicit binding mechanism and the use of tests for computing values for *free* variables in a goal.

The axiomatic definition of the π operator by means of the *logical existential quantifier*, however, again raises the issue of how to operationalise or implement the operator. In the 3APL semantics, computing bindings for (free) variables is specified as finding a suitable *term* to instantiate the variable. The logical semantics for the π operator, however, does not specify any particular mechanism for implementing it. Presumably, any implementation will have to manipulate terms and a similar mechanism as that of 3APL (based on a logic programming like parameter mechanism) is most appropriate.

To illustrate the difference between ConGolog and 3APL, we give a simple example. Suppose the language $\mathcal{L}_{now}$ (the knowledge representation language) only has a single constant a and no other function symbols. Furthermore, assume that the initial database is $\neg P(a)$. Now consider the program $\pi x.P(x)?$ and the question whether this program has a successfully terminating computation. As we will see, the 3APL translation of this ConGolog program is $\pi = \mathit{random}(x);\ P(x)?$ and it is easy to show that in this example the program π has no successfully terminating computation. In contrast, according to the ConGolog semantics the program has a successfully terminating computation in case $\exists\, x.P(x)$, which we cannot exclude given that we only know that $\neg P(a)$ is the case. The point is that there are models which satisfy $\neg P(a)$ and $\exists\, x.P(x)$ for some value in the domain, but this value has no *name*. The logical semantics of ConGolog thus constrains the behaviour of programs less than the 3APL semantics at the cost of not being able to prove certain useful properties concerning, for example, the termination behaviour of a program.

An elegant proposal to operationalise the nondeterministic selection of a value by the π operator is to assume *domain closure*. Domain closure implies that all domain elements have names, which provides for a computational mechanism to implement the π operator by computing bindings. From now on, therefore, we assume that action theories imply domain closure. That is, an action theory $\mathcal{A}$ now also includes a *domain closure axiom* like $\forall\, x(x = t_1 \vee \ldots x = t_n)$. We believe that the assumption of domain closure does not exclude infinite domains (we could use second order axioms to define the set of natural numbers, for example).

5.3 Some Useful Consequences

The fact that databases are required to be complete and that domain closure is assumed has a number of useful consequences. The most important ones are listed in this section and are used in the remainder of the paper.

Lemma 5.3 Let $\mathcal{A}$ be a basic action theory and $\sigma \subseteq \mathcal{L}_{now}$ be a complete theory. Then $\mathit{Final}(\delta, S)$ is decided by $\mathcal{A} + \sigma[S]$ for arbitrary ConGolog programs δ and closed situations S. That is,

$$\mathcal{A} + \sigma[S] \models \mathit{Final}(\delta, S) \text{ or } \mathcal{A} + \sigma[S] \models \neg \mathit{Final}(\delta, S)$$

Proof: Easy induction on the structure of δ. Use domain closure for the case that δ is a nondeterministic choice of argument program. ■

Theorem 5.4 Let $\mathcal{A}$ be a basic action theory and $\sigma \subseteq \mathcal{L}_{now}$ be a complete theory. Then $\exists\, \delta', s'.\, Trans(\delta, S, \delta', s')$ is decided by $\mathcal{A} + \sigma[S]$ for arbitrary ConGolog programs δ and closed situations S. That is,

$$\mathcal{A} + \sigma[S] \models \exists\, \delta', s'.\, Trans(\delta, S, \delta', s') \text{ or } \mathcal{A} + \sigma[S] \models \neg\, \exists\, \delta', s'.\, Trans(\delta, S, \delta', s')$$

Proof: By induction on the rank and structure of ConGolog programs. Induction on the rank of a program is used to deal with the case of Procedure Calls. This case is proven by a simple application of the induction hypothesis. The remaining cases are dealt with below.

- Basic Actions $A(\vec{t})$: Follows from the fact that $\mathcal{A} + \sigma[S] \models Poss(A(\vec{t}), S)$ or $\mathcal{A} + \sigma[S] \models \neg Poss(A(\vec{t}), S)$. The latter is implied by the specific form of precondition axioms, the fact that σ is a complete theory, and lemma 5.2.
- Tests $\phi?$: Follows from the fact (lemma 5.2) that any sentence uniform in S is decided by $\mathcal{A} + \sigma[S]$.
- Sequential Composition $\delta_1;\ \delta_2$: By lemma 5.3, $Final(\delta_1, S)$ is decided. Then apply the induction hypothesis.
- Nondeterministic Choice $\delta_1 \mid \delta_2$: Use induction hypothesis.
- Nondeterministic Choice of Argument $\pi x.\delta$: Use domain closure and induction hypothesis.
- Parallel Composition $\delta_1 \| \delta_2$: Use induction hypothesis.
- Prioritised Parallel Composition $\delta_1 \rangle\!\rangle \delta_2$: Use induction hypothesis.

■

For an arbitrary term S of sort situation, as a notational shorthand, we stipulate that $do(\epsilon, S)$ is identical to S (where ϵ denotes the empty sequence; recall that ϵ is the label associated with the transition for a test in 3APL).

Theorem 5.5 Let $\mathcal{A}$ be a basic action theory, $\sigma \subseteq \mathcal{L}_{now}$ be a complete theory and α be either ϵ or a basic action. Then any closed sentence of the form $Trans(\delta, S, \delta', do(\alpha, S))$ is decided by $\mathcal{A} + \sigma[S]$; that is, either:

$$\mathcal{A} + \sigma[S] \models Trans(\delta, S, \delta', do(\alpha, S)) \text{ or } \mathcal{A} + \sigma[S] \models \neg Trans(\delta, S, \delta', do(\alpha, S))$$

Proof: By induction on the rank and structure of ConGolog programs. The proof is completely analogous to the proof of theorem 5.4, except for the case of prioritised parallel composition. In the latter case, use theorem 5.4. ■

5.4 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, however, is provided by successor state axioms in a given basic action theory.

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 [9]. They define a progression operator for (relatively) complete basic action theories. This progression operator can be used in this context to specify the transition function $\mathcal{T}$ for 3APL basic actions.

Definition 5.6 *(progression operator)*
The progression operator *Prog* is defined by:

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

By theorem 3 in [9], the progression operator as defined in definition 5.6 yields a progression of a complete belief or data base σ since complete data bases are special cases of relatively complete databases and because we assume domain closure. A progression of a database by performing an action thus provides the update semantics of that action. By theorem 1 in [9], we then know that any sentence $\varphi[do(\alpha, S)]$ uniform in $do(\alpha, S)$ is implied by $\mathcal{A} + \sigma[S]$ iff $\mathcal{A} + \sigma[do(\alpha, S)]$ also implies $\varphi[do(\alpha, S)]$.

We can use the progression operator to specify a transition function for basic actions in 3APL. We define an update action in 3APL for every basic action A in the theory $\mathcal{A}$. The semantics of basic actions in 3APL is given by a semantic function $\mathcal{T}$ which defines in which states an action is enabled and what the resulting state of executing the action in that particular state is. $\mathcal{T}$ is defined as a partial function, which incorporates both the information of precondition and successor state axioms.

Definition 5.7 *(semantic function $\mathcal{T}$)*
Let $\sigma \subseteq \mathcal{L}_{now}$ be a complete theory, and S be a closed term of sort *situation*. Define for every action $A(\vec{t})$ its update 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 by lemma 5.2 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.

5.5 Translating ConGolog programs into 3APL agents

Now we have set the stage, we can define a translation function τ from ConGolog programs to 3APL agents. The mapping τ defined below is defined by induction on the structure of programs. One of the more interesting cases is the translation of programs of the form $\pi x.\delta$ which are mapped onto a sequential 3APL program $\mathit{random}(x);\ \tau(\delta)$. The π operator is simulated by the special action *random* in 3APL, and the explicit binding by the π operator is replaced by the implicit binding mechanism in 3APL. For this mapping to work we need to make the following assumption:

Assumption 5.8 In a ConGolog program δ all occurrences of π operators bind *different variables*. That is, πx occurs at most once for a variable x in a ConGolog program. Although this property of programs may be violated if a procedure call is instantiated with its body, in that case we will still assume that the property holds and assume that newly introduced π operators always bind variables which do not already occur in the original program. This is justified by observing that variants of a program δ obtained by renaming bound variables are operationally indistinguishable with respect to belief bases and action histories.

Finally, note that a ConGolog procedure is translated to a 3APL rule without a guard. Also notice that the translation function τ is defined on the set of open programs P, and not just on the set of closed ConGolog programs.

Definition 5.9 *(translation function τ)*
The translation function τ is inductively defined by:

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

6 Embedding ConGolog in 3APL

The embedding of ConGolog in 3APL now proceeds in three stages. First, we show how to embed ConGolog programs without procedure calls or (prioritised) parallel composition in 3APL. In Section 6, we then extend this result to programs with procedure calls (this proof involves induction on the rank of a program). Finally, in section 7 we discuss the special

problems associated with simulating parallel ConGolog programs in 3APL and show how to solve these problems. The main result, however, is established in this section where we prove an embedding result for all the basic constructs of ConGolog. We begin by showing that the concept defined by the *Final* predicate (for arbitrary ConGolog programs) coincides with termination of the corresponding 3APL τ-translation of the program modulo a number of silent steps.

Lemma 6.1 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 we have that:

$$\mathcal{A} + \sigma[S] \models \mathit{Final}(\delta, S) \text{ iff } \langle \tau(\delta), \sigma \rangle \xrightarrow{i}{}^{*} \langle E, \sigma \rangle$$

Proof: Easy proof by induction on the rank and structure of a program. Use domain closure for the case that δ is a nondeterministic choice of argument program. ■

The main embedding result is the following theorem, which shows that the transition relation defined by the *Trans* predicate bisimulates with the step relation $\Longrightarrow$.

Theorem 6.2 *(bisimulation theorem)*
Let δ and δ' be (closed) ConGolog programs, $\sigma \subseteq \mathcal{L}_{now}$ be a complete theory, and S be a closed term of sort *situation*. Then:

$$\begin{gathered} \mathcal{A} + \sigma[S] \models \mathit{Trans}(\delta, S, \delta', \mathit{do}(\alpha, S)) \\ \text{iff} \\ \langle \tau(\delta), \sigma \rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha) \rangle \end{gathered}$$

Proof: We prove the theorem by induction on the structure of programs.

Basic Actions: $\delta = A(\vec{t})$:

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}(A(\vec{t}), S, \delta', \mathit{do}(A(\vec{t}), S)) \quad \text{iff} \quad \langle A(\vec{t}), \sigma \rangle \stackrel{A(\vec{t})}{\Longrightarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, A(\vec{t})) \rangle$$

Proof:

(⇒) By definition of the *Trans* predicate, we have $\mathcal{A} + \sigma[S] \models \mathit{Poss}(A(\vec{t}), S) \wedge \delta' = \mathit{nil}$. By definition of $\mathcal{T}$ and τ, we then have $\mathcal{T}(A(\vec{t}), \sigma) = \mathit{Prog}(\sigma, A(\vec{t}))$ and $\tau(\mathit{nil}) = E$. From this we obtain $\langle A(\vec{t}), \sigma \rangle \stackrel{A(\vec{t})}{\Longrightarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, A(\vec{t})) \rangle$.

(⇐) By the transition rule for basic actions, we must have $\mathcal{T}(A(\vec{t}), \sigma) = \mathit{Prog}(\sigma, A(\vec{t}))$. This implies that $\mathcal{A} + \sigma[S] \models \mathit{Poss}(A(\vec{t}), S)$. From the definition of *Trans* we then conclude that $\mathcal{A} + \sigma[S] \models \mathit{Trans}(A(\vec{t}), S, \mathit{nil}, \mathit{do}(A(\vec{t}), S))$.

Tests: $\delta = \phi?$:

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}(\phi?, S, \delta', \mathit{do}(\epsilon, S)) \text{ iff } \quad \langle \phi?, \sigma \rangle \stackrel{\epsilon}{\Longrightarrow} \langle \tau(\delta'), \sigma \rangle$$

Proof:

(⇒) By definition of *Trans*, we have $\mathcal{A} + \sigma[S] \models \varphi[S] \wedge \delta' = nil$. Since $\varphi[S]$ is closed and uniform in S, and $\tau(nil) = E$, by lemma 5.2 we obtain $\sigma \models \varphi\varnothing$, which is the required premise of the transition rule for tests.

(⇐) By the transition rule for tests, we must have $\sigma \models \varphi$ since φ is closed, and so we also have that $\mathcal{A} + \sigma[S] \models \varphi[S]$ by lemma 5.2. From the definition of *Trans*, we conclude that $\mathcal{A} + \sigma[S] \models Trans(\phi?, S, nil, do(\epsilon, S))$.

Sequential Composition: $\delta = \delta_1;\ \delta_2$:

$$\mathcal{A} + \sigma[S] \models Trans(\delta_1;\ \delta_2, S, \delta', do(\alpha, S)) \quad \text{iff} \quad \langle \tau(\delta_1;\ \delta_2), \sigma\rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta'), Prog(\sigma, \alpha)\rangle$$

Proof: By the induction hypothesis, we may assume that we know that:

$$\mathcal{A} + \sigma[S] \models Trans(\delta_1, S, \delta_1', do(\alpha, S)) \text{ iff } \langle \tau(\delta_1), \sigma\rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta_1'), Prog(\sigma, \alpha)\rangle$$

and

$$\mathcal{A} + \sigma[S] \models Trans(\delta_2, S, \delta_2', do(\alpha, S)) \text{ iff } \langle \tau(\delta_2), \sigma\rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta_2'), Prog(\sigma, \alpha)\rangle$$

(⇒) We prove the implication by reasoning by cases: (which is allowed by theorem 5.5)

1. First, assume that $\mathcal{A} + \sigma[S] \models \exists\gamma.\delta' = (\gamma;\ \delta_2) \wedge Trans(\delta_1, S, \gamma, do(\alpha, S))$. This implies there is a ConGolog program δ_1' such that $\mathcal{A} + \sigma[S] \models \delta' = (\delta_1';\ \delta_2) \wedge Trans(\delta_1, S, \delta_1', do(\alpha, S))$. By the induction hypothesis, we then obtain: $\langle \tau(\delta_1), \sigma\rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta_1'), Prog(\sigma, \alpha)\rangle$. From this and the transition rule for sequential composition, we conclude that $\langle \tau(\delta_1;\ \delta_2), \sigma\rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta_1';\ \delta_2), Prog(\sigma, \alpha)\rangle$. Since $\delta' = (\delta_1';\ \delta_2)$, we obtain: $\langle \tau(\delta_1;\ \delta_2), \sigma\rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta'), Prog(\sigma, \alpha)\rangle$.
2. Second, assume that $\mathcal{A} + \sigma[S] \models \neg\exists\gamma.\delta' = (\gamma;\ \delta_2) \wedge Trans(\delta_1, S, \gamma, do(\alpha, S))$. By the axiomatic definition of *Trans* for ; , we then know that $\mathcal{A} + \sigma[S] \models Final(\delta_1, s) \wedge Trans(\delta_2, S, \delta', do(\alpha, S))$. Since $\mathcal{A} + \sigma[S] \models Final(\delta_1, S)$, by lemma 6.1, we conclude that $\langle \tau(\delta_1), \sigma\rangle \stackrel{i}{\longrightarrow}^{*} \langle E, \sigma\rangle$. By the induction hypothesis, we obtain that $\langle \tau(\delta_2), \sigma) \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta'), Prog(\sigma, \alpha)\rangle$. By the definition of $\Longrightarrow$ we then may add the silent steps of δ_1 in front of the computation involving δ_2 and we obtain: $\langle \tau(\delta_1;\ \delta_2), \sigma\rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta'), Prog(\sigma, \alpha)\rangle$.

(⇐) From $\langle \tau(\delta_1;\ \delta_2), \sigma\rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta'), Prog(\sigma, \alpha)\rangle$ it follows that: (i) $\langle \tau(\delta_1), \sigma\rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta_1'), Prog(\sigma, \alpha)\rangle$ for some δ_1', or (ii) $\langle \tau(\delta_2), \sigma\rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta'), Prog(\sigma, \alpha)\rangle$ and $\langle \tau(\delta_1), \sigma\rangle \stackrel{i}{\longrightarrow}^{*} \langle E, \sigma\rangle$. In the first case, simply apply the induction hypothesis and note that δ' must be of the form $\delta_1';\ \delta_2$ to conclude that $\mathcal{A} + \sigma[S] \models Trans(\delta_1;\ \delta_2, S, \delta', do(\alpha, S))$. In the second case, use lemma 6.1 to obtain $\mathcal{A} + \sigma[S] \models Final(\delta_1, S)$ and apply the induction hypothesis.

Nondeterministic Choice: $\delta = \delta_1 + \delta_2$:

$$\mathcal{A} + \sigma[S] \models Trans(\delta_1 \mid \delta_2, S, \delta', do(\alpha, S)) \quad \text{iff} \quad \langle \tau(\delta_1) + \tau(\delta_2), \sigma\rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta'), \sigma'\rangle$$

Proof:

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}(\delta_1 \mid \delta_2, S, \delta', \mathit{do}(\alpha, S))$$

By the axiomatic definition of *Trans* this is equivalent to:

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}(\delta_1, S, \delta', \mathit{do}(\alpha, S)) \vee \mathit{Trans}(\delta_2, S, \delta', \mathit{do}(\alpha, S))$$

Now, by theorem 5.5 and the fact that σ is complete, the latter is equivalent to:

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}(\delta_1, S, \delta', \mathit{do}(\alpha, S)) \text{ or } \mathcal{A} + \sigma[S] \models \mathit{Trans}(\delta_2, S, \delta', \mathit{do}(\alpha, S))$$

This in turn, by the induction hypothesis, is equivalent to:

$$\langle \tau(\delta_1), \sigma \rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha) \rangle \text{ or } \langle \tau(\delta_2), \sigma \rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha) \rangle$$

Finally, this is equivalent to:

$$\langle \tau(\delta_1) + \tau(\delta_2), \sigma \rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha) \rangle$$

Nondeterministic Choice of Argument: $\delta = \pi x.\delta'$:

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}((\pi x.\delta'), S, \delta'', \mathit{do}(\alpha, S)) \quad \text{iff} \quad \langle \tau(\pi x.\delta'), \sigma \rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta''), \sigma' \rangle$$

Proof:

(⇒)

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}((\pi x.\delta'), S, \delta'', \mathit{do}(\alpha, S))$$

By the axiomatic definition of *Trans* this is equivalent to:

$$\mathcal{A} + \sigma[S] \models \exists\, x.\mathit{Trans}(\delta', S, \delta'', \mathit{do}(\alpha, S)))$$

Because of domain closure, there is a ground witness t for $\exists\, x$ and we obtain:

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}(\delta'\{x = t\}, S, \delta'', \mathit{do}(\alpha, S))$$

By the induction hypothesis, we then have that

$$\langle \tau(\delta'\{x = t\}), \sigma \rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta''), \mathit{Prog}(\alpha, \sigma) \rangle$$

From this, the fact that all choice operators πx in δ bind different variables by assumption 5.8, and because $\mathit{random}(x)$ gives rise to an internal step, we can then derive:

$$\langle \mathit{random}(x);\ \tau(\delta'), \sigma \rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta''), \sigma' \rangle$$

By definition of the translation function τ, we obtain:

$$\langle \tau(\pi x.\delta'), \sigma \rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta''), \sigma' \rangle$$

($\Leftarrow$) From

$$\langle random(x);\ \tau(\delta'), \sigma\rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta''), Prog(\sigma, \alpha)\rangle$$

it follows that:

$$\langle \tau(\delta\{x = t\}), \sigma\rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta''), Prog(\sigma, \alpha)\rangle$$

for some t, since $random(x)$ gives rise to an internal step. By the induction hypothesis, we then obtain:

$$\mathcal{A} + \sigma[S] \models Trans(\delta'\{x = t\}, S, \delta'', do(\alpha, S))$$

By the semantic definition of $\exists\, x$ we then can derive:

$$\mathcal{A} + \sigma[S] \models \exists\, x. Trans(\delta', S, \delta'', do(\alpha, S))$$

And finally, by the axiomatic definition of *Trans* this is equivalent to:

$$\mathcal{A} + \sigma[S] \models Trans((\pi x.\delta'), S, \delta'', do(\alpha, S))$$

As a corollary, we obtain that ConGolog programs (without procedures or parallelism) and the translations of these programs in 3APL compute the same belief or data bases and compute these belief bases by executing the same sequence of actions:

Corollary 6.3 Let $\mathcal{A}$ be a basic action theory with initial database $\sigma[S_0]$, and α be a sequence of basic actions.

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

where $Trans^*$ denotes the transitive closure of *Trans*.

7 Procedures

In this section, we extend the bisimulation result to include programs with procedures. The proof proceeds by induction on the rank and structure of a program. The proofs for programs without procedure calls are essentially the same as in the previous section. The only interesting new case is that of simulating a procedure call.

Procedure Call: $\delta = P(\vec{t})$:

$$\mathcal{A} + \sigma[S] \models Trans(P(\vec{t}), S, \delta', do(\alpha, S)) \quad \text{iff} \quad \langle \tau(P(\vec{t})), \sigma\rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta'), \sigma'\rangle$$

Proof: We proceed by induction on the rank of a program. The base case, programs with rank 0, coincides with programs without procedure calls, and is proven in the previous section. Assuming that we know the simulation result holds for all programs of rank n, we now show that it also holds for programs with rank $n+1$:

(⇒) Suppose $P(\vec{t})$ is of rank $n+1$ and $\mathcal{A}+\sigma[S] \models \mathit{Trans}(P(\vec{t}), S, \delta', \mathit{do}(\alpha, S))$. Then we also have $\mathcal{A}+\sigma[S] \models \mathit{Trans}({\delta_P}^{\vec{v}}_{\vec{t}}, S, \delta', \mathit{do}(\alpha, S))$ where ${\delta_P}^{\vec{v}}_{\vec{t}}$ is the body of procedure P with formal parameters replaced with actual parameters. Moreover, if P is of rank $n+1$, then δ_P is of rank n. By assumption 5.8 we may assume the body δ_P introduces only πv such that v does not already occur in the program making the procedure call. Then, by the induction hypothesis, we obtain: $\langle \tau({\delta_P}^{\vec{v}}_{\vec{t}}), \sigma\rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha)\rangle$. By the transition rule for rule application, we then derive: $\langle \tau(P(\vec{t})), \sigma\rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha)\rangle$, because application of a rule is an internal step.

(⇐) Suppose $\langle \tau(P(\vec{t})), \sigma\rangle \stackrel{\alpha}{\Longrightarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha)\rangle$. Then there must be some rule such that: $\langle \tau(P(\vec{t})), \sigma\rangle \longrightarrow \langle \tau({\delta_P}^{\vec{v}}_{\vec{t}}), \sigma\rangle$. Because $P(\vec{t})$ is of rank $n+1$, ${\delta_P}^{\vec{v}}_{\vec{t}}$ must be of rank n. By the induction hypothesis, we then have: $\mathcal{A}+\sigma[S] \models \mathit{Trans}({\delta_P}^{\vec{v}}_{\vec{t}}, S, \delta', \mathit{do}(\alpha, S))$. From this, by using assumption 5.8, we immediately obtain: $\mathcal{A}+\sigma[S] \models \mathit{Trans}(P(\vec{t}), S, \delta', \mathit{do}(\alpha, S))$.

■

8 Parallel Composition

In this section, we extend the bisimulation result to *arbitrary ConGolog programs* including parallel as well as prioritised parallel programs. To simulate prioritised parallel programs 3APL is extended with the $\rangle\!\rangle$ operator and we show how to define the semantics of this operator in a transition style semantics below. Originally, the parallel composition of goals was included in 3APL. The semantics of the parallel operator $\|$ in 3APL, however, differs from that of the ConGolog semantics. Because of this difference, the main issue in extending the simulation result to parallel programs is to prove that the two semantics are equivalent with respect to some appropriate observation criterion. That is, we must show that parallel programs in 3APL and ConGolog compute the same things.

The difference in the semantic definitions of the parallel operator concerns the ordering of computation steps of a 3APL (parallel) program. A 3APL program can perform silent steps which do not have a counterpart in the ConGolog semantics. The problem concerns the order in which these silent steps are performed. This can be illustrated as follows: given a parallel program $\delta_1 \| \delta_2$, in the ConGolog semantics only δ_1 *or* δ_2 can be transformed in a single step but *not* both, while according to the 3APL semantics as defined by $\Longrightarrow$ both subprograms may perform silent steps *before* an action or a test (a 'real' ConGolog step) is performed. For example, the ConGolog program

proc $P()$
  A
end
proc $Q()$
  B
end
$P() \| Q()$

can execute either the left or right branch of the parallel composition resulting in respectively $Q()$ and $P()$ as the remaining programs for execution. The program is translated to $P()\|Q()$ in 3APL (plus translations of the procedure definitions to rules). According to the $\Longrightarrow$ semantics which abstracts from silent steps, this program, however, can result in either $P()$, $Q()$, A or B after performing one $\Longrightarrow$ step. The latter two new possibilities result from the fact that with respect to the $\Longrightarrow$ semantics a silent step may have been performed in which the procedure is expanded into its body before an actual step (not a silent step) is performed. The $\Longrightarrow$ transition semantics thus allows silent steps of *both* subprograms to be performed before an actual step is performed in either one of them, whereas we would like to make sure that only computation steps associated with one of the subprograms are performed.

To solve this problem we show that the order of performing silent steps does not matter. For this purpose, we introduce a new transition relation $\rightsquigarrow$ which imposes a restriction on the order in which silent steps are performed in a parallel program and show that $\rightsquigarrow$ and $\longrightarrow$ are equivalent in the sense that they compute the same belief bases (our observation criterion).

The transition relation $\rightsquigarrow$ is derived from $\Longrightarrow$. In the definition of the $\rightsquigarrow$ transition relation also a specification of the semantics of the prioritised parallel operator is given. The transition rule for prioritised parallel composition $\pi_1 \rangle\!\rangle \pi_2$ uses a negative premise to specify that the execution of π_2 is only allowed if π_1 cannot perform an action or test after a finite number of silent steps. A justification for this type of transition rule can be found in [3].

Definition 8.1 *(transition relation $\rightsquigarrow$)*
Let *Parfree* denote the set of 3APL programs without occurrences of parallel operators.

$$\frac{\langle \pi, \sigma\rangle \stackrel{l}{\Longrightarrow}_\gamma \langle \pi', \sigma'\rangle, \pi \in \mathit{Parfree}}{\langle \pi, \sigma\rangle \stackrel{l}{\rightsquigarrow}_\gamma \langle \pi', \sigma'\rangle}$$

$$\frac{\langle \pi_1, \sigma\rangle \stackrel{l}{\rightsquigarrow}_\gamma \langle \pi_1', \sigma'\rangle}{\langle \pi_1;\ \pi_2, \sigma\rangle \stackrel{l}{\rightsquigarrow}_\gamma \langle \pi_1';\ \pi_2\gamma, \sigma'\rangle}$$

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

$$\frac{\langle \pi_1, \sigma\rangle \stackrel{l}{\rightsquigarrow}_\gamma \langle \pi_1', \sigma'\rangle}{\langle \pi_1 \| \pi_2, \sigma\rangle \stackrel{l}{\rightsquigarrow}_\gamma \langle \pi_1' \| \pi_2\gamma, \sigma'\rangle} \qquad \frac{\langle \pi_2, \sigma\rangle \stackrel{l}{\rightsquigarrow}_\gamma \langle \pi_2', \sigma'\rangle}{\langle \pi_1 \| \pi_2, \sigma\rangle \stackrel{l}{\rightsquigarrow}_\gamma \langle \pi_1\gamma \| \pi_2', \sigma'\rangle}$$

$$\frac{\langle \pi_1, \sigma\rangle \stackrel{l}{\rightsquigarrow}_\gamma \langle \pi_1', \sigma'\rangle}{\langle \pi_1 \rangle\!\rangle \pi_2, \sigma\rangle \stackrel{l}{\rightsquigarrow}_\gamma \langle \pi_1' \rangle\!\rangle \pi_2\gamma, \sigma'\rangle}$$

$$\frac{\forall\, \pi_1', \sigma', \theta, m(\langle \pi_1, \sigma\rangle \stackrel{m}{\not\rightsquigarrow}_\theta \langle \pi_1', \sigma'\rangle) \text{ and } \langle \pi_2, \sigma\rangle \stackrel{l}{\rightsquigarrow}_\gamma \langle \pi_2', \sigma''\rangle}{\langle \pi_1 \rangle\!\rangle \pi_2, \sigma\rangle \stackrel{l}{\rightsquigarrow} \langle \pi_1\gamma \rangle\!\rangle \pi_2', \sigma''\rangle}$$

The following theorem provides the basis for our simulation result. It proves that the transition relation $\rightsquigarrow$ is equivalent to $\Longrightarrow$ with respect to computed belief bases, which is used as the observation criterion here for programs without prioritised parallel composition (which is not defined for $\Longrightarrow$).

Theorem 8.2 Let π be the translation of a ConGolog program δ without prioritised parallel composition, i.e. $\pi = \tau(\delta)$. Then:

$$\langle \pi, \sigma \rangle \Longrightarrow^* \sigma' \text{ iff } \langle \pi, \sigma \rangle \rightsquigarrow^* \sigma'$$

Proof:

(⇒) By induction on the length of the computation and the structure of π. The base case, a computation of length 1, is trivial, since in that case π must be $A(\vec{t})$ or $\phi?$. So, suppose for all computations of length n the theorem holds. We must prove it for computations of length $n+1$.

- Basic Actions, Tests: Easy.
- Random Action: Is not a translation of a ConGolog program.
- Sequential Composition: Easy; partition computation into $\stackrel{\alpha}{\Longrightarrow}$ steps.
- Nondeterministic Choice: Easy.
- Parallel Composition: $\pi = \pi_1 \| \pi_2$.
  Take the first m steps of the computation such that the mth step is a $A(\vec{t})$ or $\phi?$ step and all previous steps are internal steps. The mth step is either performed by π_1 or by π_2. Suppose it is performed by π_1 (the other case is analogous). By rearranging the first m steps such that all steps performed by π_1 are performed first - in the same order - and then performing the (internal) steps performed by π_2, we still have a legal computation which does not change the computed result (since the steps from π_2 only expand procedure calls or randomly guess values). Now, the sequence of π_1 steps correspond to one $\Longrightarrow$ step by definition. The remaining computation is at least one computation step shorter, and thus we are done.
- Procedure Call: Use induction hypothesis.

(⇐) Trivial.

■

Now we are able to extend the simulation result to (prioritised) parallel programs. To prove this extended simulation result, we use the new transition relation $\rightsquigarrow$. The proof for all cases except for parallel and prioritised parallel composition are analogous to the proofs of previous sections and are omitted for this reason. Theorem 8.3 shows that the transition relation defined by *Trans* for arbitrary ConGolog programs bisimulates with the step relation $\rightsquigarrow$.

Theorem 8.3 *(bisimulation which includes parallel and prioritised parallel composition)*

Parallel Composition: $\delta = \delta_1 \| \delta_2$:

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}(\delta_1 \| \delta_2, S, \delta', do(\alpha, S)) \quad \text{iff} \quad \langle \tau(\delta_1) \| \tau(\delta_2), \sigma \rangle \stackrel{\alpha}{\rightsquigarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha) \rangle$$

Proof: Note that the base case, i.e. $\langle \pi, \sigma \rangle \rightsquigarrow \langle \pi', \sigma' \rangle$ because $\langle \pi, \sigma \rangle \Longrightarrow \langle \pi', \sigma' \rangle$, has been proven in previous sections. We now deal with the remaining cases.

(⇒) Assume $\mathcal{A} + \sigma[S] \models \mathit{Trans}(\delta_1 \| \delta_2, S, \delta', \mathit{do}(\alpha, S))$. We need to distinguish two cases, the case where δ_1 is executed and the case where δ_2 is executed. Because of symmetry, we only give the details for one of these cases. So, assume: $\mathcal{A} + \sigma[S] \models \exists \gamma.\delta' = (\gamma \| \delta_2) \wedge \mathit{Trans}(\delta_1, S, \gamma, \mathit{do}(\alpha, S))$. As a consequence, there is a δ_1' such that: $\mathcal{A} + \sigma[S] \models \delta' = (\delta_1' \| \delta_2) \wedge \mathit{Trans}(\delta_1, S, \delta_1', \mathit{do}(\alpha, S))$. Then, by the induction hypothesis, we obtain: $\langle \tau(\delta_1), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta_1'), \mathit{Prog}(\sigma, \alpha) \rangle$. By definition, we then have: $\langle \tau(\delta_1) \| \tau(\delta_2), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha) \rangle$.

(⇐) Assume $\langle \tau(\delta_1) \| \tau(\delta_2), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha) \rangle$. In that case, by definition 8.1 we know that either $\langle \tau(\delta_1), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta_1'), \mathit{Prog}(\sigma, \alpha) \rangle$ for some δ_1' such that $\delta' = \delta_1' \| \delta_2$, or $\langle \tau(\delta_2), \sigma \rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta_2'), \mathit{Prog}(\sigma, \alpha) \rangle$ for some δ_2' such that $\delta' = \delta_1 \| \delta_2'$. Then apply the induction hypothesis.

Prioritised Parallel Composition: $\delta = \delta_1 \rangle\!\rangle \delta_2$:

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}(\delta_1 \rangle\!\rangle \delta_2, S, \delta', \mathit{do}(\alpha, S)) \quad \text{iff} \quad \langle \tau(\delta_1 \rangle\!\rangle \delta_2), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha) \rangle$$

Proof:

(⇒) We prove the implication by reasoning by cases: (which is allowed by theorem 5.5)

1. First, assume that $\mathcal{A} + \sigma[S] \models \exists \gamma.\delta' = (\gamma \rangle\!\rangle \delta_2) \wedge \mathit{Trans}(\delta_1, S, \gamma, \mathit{do}(\alpha, S))$. Suppose δ_1' is a ConGolog program which satisfies this equation. By the induction hypothesis, we then obtain that $\langle \tau(\delta_1), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta_1'), \mathit{Prog}(\sigma, \alpha) \rangle$. By the transition rule for prioritised parallel composition of definition 8.1, this implies that $\langle \tau(\delta_1 \rangle\!\rangle \delta_2), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha) \rangle$.
2. Secondly, assume that $\mathcal{A} + \sigma[S] \models \neg \exists \gamma.\delta' = (\gamma \rangle\!\rangle \delta_2) \wedge \mathit{Trans}(\delta_1, S, \gamma, \mathit{do}(\alpha, S))$. By the axiomatic definition of *Trans*, we then have that $\mathcal{A} + \sigma[S] \models \exists \gamma.\delta' = (\delta_1 \rangle\!\rangle \gamma) \wedge \mathit{Trans}(\delta_2, S, \gamma, \mathit{do}(\alpha, S)) \wedge \neg \exists \eta, s''.\mathit{Trans}(\delta_1, S, \eta, s'')$. By the induction hypothesis, we then have that $\langle \tau(\delta_2), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta_2'), \mathit{Prog}(\sigma, \alpha) \rangle$ for some δ_2' such that $\delta' = \delta_1 \rangle\!\rangle \delta_2'$. Moreover, there is no transition $\rightsquigarrow$ corresponding to δ_1. By the transition rule for prioritised parallel composition, we then have that $\langle \tau(\delta_1 \rangle\!\rangle \delta_2), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha) \rangle$.

(⇐) From $\langle \tau(\delta_1 \rangle\!\rangle \delta_2), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha) \rangle$ it follows that: (i) $\langle \tau(\delta_1), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta_1'), \mathit{Prog}(\sigma, \alpha) \rangle$ for some δ_1' or (ii) $\langle \tau(\delta_2), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta_2'), \mathit{Prog}(\sigma, \alpha) \rangle$ for some δ_2' and there is no transition associated with δ_1. In both cases, use the induction hypothesis to conclude that $\mathcal{A} + \sigma[S] \models \mathit{Trans}(\delta_1 \rangle\!\rangle \delta_2, S, \delta', \mathit{do}(\alpha, S))$.

■

Finally, we obtain that *arbitrary* ConGolog programs and the translations of these programs in 3APL compute the same belief or data bases and compute these belief bases by executing the same sequence of actions. The extended version of corollary 6.3 now includes all ConGolog programs.

Corollary 8.4 Let $\mathcal{A}$ be a basic action theory with initial database $\sigma[S_0]$, and α be a sequence of basic actions.

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

■

9 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 contrast 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 inherited from logic programming, which is used to implement ConGolog [2].

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' [7]. 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 dictate the use of one or the other model of execution (cf. also [1]).

Finally, the construction of an embedding of ConGolog into 3APL also identified several common and distinguishing features of the formalisms used to specify a semantics. Basically, the embedding result indicates that an axiomatic definition in an extended predicate logic like the situation calculus and a Plotkin-style transition semantics result in more or less equivalent semantics. The use of a logical semantics specified in the situation calculus, however, requires careful consideration of a number of issues, like the incompleteness of databases and domain closure. Moreover, the logical semantics seems suitable for proving partial correctness, but

raises some doubts as to its usefulness for proving termination properties (even more so with respect to the second order semantics).

References

[1] Guiseppe de Giacomo and Hector J. Levesque. An incremental interpreter for high-level programs with sensing. Technical report, Department of Computer Science, Univ. of Toronto, 1998.

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

[3] Jan Friso Groote. Transition System Specifications with Negative Premises. *Theoretical Computer Science*, 118(2):263–299, 1993.

[4] 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.

[5] 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 G. Antoniou and J. Slaney, editors, *Advanced Topics in Artificial Intelligence (LNAI 1502)*, pages 155–166. Springer-Verlag, 1998.

[6] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Formal Semantics for an Abstract Agent Programming Language. In Munindar P. Singh, Anand Rao, and Michael J. Wooldridge, editors, *Intelligent Agents IV (LNAI 1365)*, pages 215–229. Springer-Verlag, 1998.

[7] 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.

[8] 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.

[9] Fangzhen Lin and Ray Reiter. How to Progress a Database. *Artificial Intelligence*, 92:131–167, 1997.

[10] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[11] J. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In Meltzer and Michie, editors, *Machine Intelligence*, pages 463–502. Edinburgh University Press, 1969.

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

[13] G. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, Computer Science Department, 1981.

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

[15] Ray Reiter. The Frame Problem in the Situation Calculus: A Simple Solution (Sometimes) and a Completeness Result for Goal Regression. In *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, 1991.

[16] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.