The Cognitive Agents Specification Language and Verification Environment for Multiagent Systems

Steven Shapiro
Dept. of Computer Science
University of Toronto
Toronto, ON M5S 3G4
Canada
steven@ai.toronto.edu

Yves Lespérance
Dept. of Computer Science
York University
Toronto, ON M3J 1P3
Canada
lesperan@cs.yorku.ca

Hector J. Levesque
Dept. of Computer Science
University of Toronto
Toronto, ON M5S 3G4
Canada
hector@ai.toronto.edu

ABSTRACT

The Cognitive Agents Specification Language (CASL) is a framework for specifying multiagent systems. It has a mix of declarative and procedural components to facilitate the specification and verification of *complex* multiagent systems. In this paper, we describe CASL and a verification environment (CASLve) for it based on the PVS verification system. We give an example of a multiagent meeting scheduler application specified with CASL. To illustrate the verification system, we discuss a proof we carried out in it, namely, that all bounded-loop CASL specifications terminate.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*multiagent systems*; D.2.1 [**Software Engineering**]: Requirements/Specifications—*languages,tools*; D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods*

General Terms

Verification, Languages, Reliability

Keywords

Agent specification languages, verification tools, theorem proving, proof assistants

1. INTRODUCTION

The Cognitive Agents Specification Language (CASL) is a framework for specifying multiagent systems, which allows the specifier to view agents as entities with mental states, such as knowledge, beliefs, and goals, and to define the behavior of the agents in terms of their mental states. It combines a declarative action theory defined in the situation calculus [11, 12, 13] — which allows the specifier to methodically and concisely describe the effects of actions on the world and the mental states of agents — with a rich programming/process language with constructs for concurrency and non-determinism to facilitate the specification and verification of *complex* multiagent systems. In the next section, we will describe the different aspects of CASL, and in Sec. 3, we will illustrate its use on a multiagent system application.

We are also developing a verification environment (CASLve) for CASL based on the Prototype Verification System (PVS) [10] to make it easier to verify properties of CASL specifications. CASLve uses a representation of the CASL formalism within PVS that is also described in the next section. The verification environment should provide the user with a comprehensive library of proof methods, or lemmas, to facilitate the proof of various types of results (safety, liveness, termination, etc.). We are in the process of building such a library. For establishing termination, one important such lemma is that bounded-loop programs or subprograms terminate. We have proven such a lemma and discuss it in detail in Sec. 4, as an example of what is involved in the process of verification and building a library for this purpose. We show part of the proof in Sec. 5 to illustrate how CASLve is used. The environment should also provide specialized proof strategies for reasoning about CASL multiagent system specifications (e.g., regression) and customized tools for displaying CASL specifications and proofs, and these are planned for future work.

2. SPECIFICATION

In this section, we discuss the CASL specification language and how we represent it in PVS.

2.1 PVS

PVS [10] is a typed, higher-order logic together with a proof system to facilitate theorem proving. The language has useful features, such as abstract datatypes and recursive definitions of functions and relations. PVS also has an extensive library of theories of mathematics, and datatypes such as lists, infinite sequences, arrays, records, etc. The proof system has built-in proof strategies, including ones for inductive proofs, and facilities for adding new strategies. PVS also features a convenient Emacs-based user-interface, a facility for displaying proof trees graphically, and proof-management functionality.

A PVS specification is a collection of *theories* that are similar to logical theories except that they contain extra syntax for such purposes as declaring new types and declaring types of variables and constants. Theories can be parameterized, yielding a limited form of polymorphism. We will, as much as possible, omit the extra-logical syntax of the PVS language to make the theories look more like classical higher-order logic.

AAMAS'02, July 15-19, 2002, Bologna, Italy.

The PVS proof system is a standard sequent calculus for higher-order logic with high-level proof strategies and decision procedures to facilitate equational and mathematical reasoning. As mentioned above, PVS comes with an extensive library of theories, some of which are built in to the standard proof strategies and decision procedures. We will use $\Gamma \vdash_{\mathrm{pvs}} \alpha$ to denote that the sentence α can be derived from the theory Γ and the built-in library of theories using the PVS proof system. In the remainder of this section, we will discuss the different components of CASL, and how they were represented in PVS.

2.2 Action Theory

The situation calculus is a predicate-calculus language for representing dynamically changing domains. A situation represents a snapshot of the domain. There is a set of initial situations corresponding to the ways an agent thinks the domain might be initially. The actual initial state of the domain is represented by the distinguished initial situation constant, S_0. The term $do(a, s)$ denotes the unique situation that results from an agent performing action a in situation s. Thus, the situations can be structured into a set of trees, where the root of each tree is an initial situation and the arcs are actions.

Predicates and functions that have a situation argument (which by convention is placed last) are called *fluents*. Fluents are used to talk about the dynamic aspects of the domain. For example, $\mathrm{IN}(agt, r, s)$ could be used to specify that agt is in room r in situation s. The effects of actions on fluents are defined using successor state axioms [11], which provide a succinct representation for both effect axioms and frame axioms [9].

To completely specify the dynamics of an application domain, we use a theory with the following kinds of axioms: (1) successor state axioms for the fluents, which describe how they are affected by actions (2) action precondition axioms, which specify the circumstances under which an action can be performed, (3) initial state axioms, which describe the initial state of the domain and the initial mental state of the agent, (4) unique names axioms for the actions, and (5) domain-independent foundational axioms (discussed below).

The axioms which define the structure of the situations (including an induction axiom) are called the *foundational axioms*. Reiter [11] formulated foundational axioms for the case where there is only a single initial situation, S_0. Since we will need multiple initial situations to model knowledge and goals, we use the axiomatization provided by Lakemeyer and Levesque (L&L)[7].

L&L first define the initial situations to be those that have no predecessors: $Init_{LL}(s') \stackrel{\text{def}}{=} \neg\exists a, s.s' = do(a, s)$. Then, they define a relation on situations $s \preceq_{LL} s'$ that holds if s' can be reached from s by a (possibly empty) sequence of actions. $\preceq_{LL}$ is defined to be the smallest relation that is reflexive and transitive and contains $(s, do(a, s))$ for any s and a:

$$
\begin{aligned}
s \preceq_{LL} s' \stackrel{\text{def}}{=} \forall P[&(\forall s_1.P(s_1, s_1)) \wedge (\forall a, s_1.P(s_1, do(a, s_1))) \wedge \\
&(\forall s_1, s_2, s_3.P(s_1, s_2) \wedge P(s_2, s_3) \supset P(s_1, s_3)) \supset \\
&\quad P(s, s')]
\end{aligned}
$$

The foundational axioms are as follows:

F1. $Init_{LL}(S_0)$.

F2. $\forall a_1, a_2, s_1, s_2.do(a_1, s_1) = do(a_2, s_2) \supset (a_1 = a_2 \wedge s_1 = s_2)$.

F3. $\forall P(\forall s, s'.Init_{LL}(s) \wedge s \preceq_{LL} s' \supset P(s')) \supset \forall s.P(s)$.

F1 declares that S_0 is an initial situation. F2 states that performing different actions yields different situations, i.e., the *do* function is 1-1. F3 is an induction axiom which says that if a property holds for any situation that can be reached from an initial situation, then that property holds for all situations.

We represent situations in PVS as an abstract datatype:

$$
\begin{aligned}
&\mathit{Sit} : \mathrm{DATATYPE} \\
&\quad \mathrm{BEGIN} \\
&\quad\quad \mathit{addinit}(\mathit{getroot} : \mathit{Rootset}) : \mathit{Init} \\
&\quad\quad \mathit{do}(\mathit{lastact} : \mathit{Action}, \mathit{undo} : \mathit{Sit}) : \mathit{Noninit} \\
&\quad \mathrm{END}\mathit{Sit}
\end{aligned}
$$

This datatype is called *Sit*. It uses two types, *Rootset* and *Action*, so we assume that these have been previously declared as types. The *Rootset* type is a set of objects that will become the initial situations in the situation datatype. The *Action* type is the set of actions. More information about the objects in these types will be given by applications that use this datatype, but we do not need any more details about the types for the declaration.

Datatypes have three main elements: constructors, accessors, and recognizers. Constructors form the elements of datatypes, i.e., they are functions whose values are objects in the datatype. Accessors map elements of the datatype back into the objects that were used to construct them. Recognizers are predicates that identify which constructor was used to construct an element of the datatype. The *Sit* datatype has two constructors. The first one, *addinit*, maps objects of the type *Rootset* into initial situations. The second one, *do*, maps an action and a situation into a non-initial situation. There are also two recognizers. $Init(s)$ ($Noninit(s)$, resp.) will be true iff s is an initial (non-initial, resp.) situation. *getroot* is an accessor that maps an initial situation into the element of *Rootset* that was used to construct it. *lastact* (*undo*, resp.) is an accessor that maps a non-initial situation $do(a, s)$ into a (s, resp.). Note that datatype declarations can be recursive, i.e., one can use the datatype as a type in its own declaration.

The datatype declaration generates a theory that formalizes the datatype. We can infer L&L's definitions and the last two foundational axioms from the theory generated by the datatype declaration for *Sit*. The first foundational axiom has to be explicitly added to our theory. The axioms generated for the *Init* recognizer imply that for any s, $Init(s)$ holds iff $Init_{LL}(s)$ holds. Also, among the definitions generated by the datatype declaration is a relation, $Subterm(s, s')$, which holds if s is a subterm of s' (i.e., s' can be reached from s with a finite number of applications of *do*). We will substitute the infix operator $\preceq$ for *Subterm* to make it better fit its intuitive meaning for situations. $\preceq$ is equivalent to $\preceq_{LL}$.

Let Sit denote the theory that is generated by the *Sit* datatype augmented with the axiom: $Init(S_0)$. We can show that it correctly represents the theory obtained from L&L's foundational axioms:[1]

Theorem 1

1. $\mathrm{Sit} \vdash_{\mathrm{pvs}} \forall s.Init(s) \equiv Init_{LL}(s)$
2. $\mathrm{Sit} \vdash_{\mathrm{pvs}} \forall s, s'.s \preceq s' \equiv s \preceq_{LL} s'$
3. $\mathrm{Sit} \vdash_{\mathrm{pvs}} \forall a_1, a_2, s_1, s_2.do(a_1, s_1) = do(a_2, s_2) \supset a_1 = a_2 \wedge s_1 = s_2$
4. $\mathrm{Sit} \vdash_{\mathrm{pvs}} \forall P(\forall s, s'.Init(s) \wedge s \preceq s' \supset P(s')) \supset \forall s P(s)$

[1] All the theorems appearing in this paper were verified with CASLve [13].

2.3 Knowledge and Goals

In CASL, we want to be able to model agents in terms of their mental states. Specifically, we include operators to specify agents' information (what they know or believe) and their motivation (what their goals are). Scherl and Levesque [12] showed how to model (single-agent) knowledge and sensing actions in the situation calculus, using a possible-worlds semantics with an accessibility relation K, where the possible worlds are situations. We adapt their approach to handle multiple agents and communicative actions between agents. $K(agt, s', s)$ will be used to denote that in situation s, agt thinks that situation s' might be the actual situation. An agent knows a formula ϕ[2] in s if ϕ holds in all situations K-accessible from s, i.e., **Know**$(agt, \phi, s) \stackrel{\text{def}}{=} \forall s'.K(agt, s', s) \supset \phi[s']$.

Scherl and Levesque [12] show how to obtain a successor state axiom for K that completely specifies how knowledge is affected by actions. In their framework, the knowledge-producing actions were performed by the agent itself, i.e., sensing actions. In CASL, we are interested in communication actions, which are actions that affect the mental state of the agent to whom they are addressed rather than that of the agent who performs the action. However, Scherl and Levesque's successor state axiom for K is easily adapted to handle communicative actions. In [15], we give a successor state axiom for K that handles knowledge expansion as a result of INFORM actions. Knowledge expansion is a special case of belief change where an agent adds to its existing knowledge but never discovers that it was mistaken about something it believed. In [16], we show how to generalize Scherl & Levesque's framework to handle *belief* change more generally (for the single agent case with sensing actions), where the agent can discover that it was mistaken about its beliefs and be forced to revise them.

We formalize the goals of an agent using an accessibility relation, $W(agt, s', s)$ which holds if in situation s, agt considers that in s' everything that it wants to be true is actually true [14]. For example, if agt wants to become a millionaire in s, then in all situations W-related to s, agt is a millionaire, but these situations can be arbitrarily far in the future. Goals will be evaluated relative to two situations now and $then$, where $now \preceq then$. We can think of $then$ as defining a path of situations, namely, the sequence of situations in the history of $then$. Intuitively, now is the situation along that path that occurs at the current time, i.e., the situations that come before now are considered to be in the past, and the situations that come after now are considered to be in the future. Thus, goal formulae are evaluated relative to a path of situations and the current 'time'. For example, we could represent the goal of increasing one's wealth as: $\text{TOTALASSETS}(then) > \text{TOTALASSETS}(now)$.

We use the K accessibility relation to pick out the current situation along a path, since the K-accessible situations are the ones that the agent thinks might be the current situation. We say that an agent has ψ[3] as a goal in s if ψ holds on the path defined by now and $then$, where $then$ is W-related to s and now is K-related to s and $now \preceq then$:

$$\begin{aligned}&\textbf{Goal}(agt, \psi, s) \stackrel{\text{def}}{=}\\ &\quad \forall now, then. W(agt, then, s) \wedge K(agt, now, s) \wedge\\ &\qquad now \preceq then \supset \psi[now, then].\end{aligned}$$

[2] ϕ is a formula that can contain a situation variable, now, in the place of situation terms. $\phi[s]$ denotes the formula that results from substituting s for now in ϕ. We often suppress now when the intent is clear from the context.

[3] ψ is a formula that can contain two situation variables, now and $then$, in the place of situation terms. $\psi[s, s']$ denotes the formula that results from substituting s for now and s' for $then$ in ψ. We often suppress now and $then$ when the intent is clear from the context.

In [13], we formulate a successor-state axiom for W that handles goal expansion as a result of REQUEST actions and goal contraction as a result of CANCELREQUEST actions.

We can encode the mental state operators in PVS quite straightforwardly from the logical characterizations given above. The only difficulty is that we need to represent formulae as terms since the **Know** and **Goal** operators take formulae as arguments, and we need to be able to obtain the value of formula at a situation. One solution would be to encode formulae and variables as first-order terms and encode the substitution using a (long) set of axioms, such as the one given by by De Giacomo et. al. in [3]. Instead, since PVS is a higher-order logic, we use predicates on situations to represent formulae. This has the advantage that no axiomatization is needed to encode formulae as terms, nor to represent substitution. For example, the formula $\forall x, y.\text{ON}(x, y, now) \supset \neg\text{ON}(y, x, now)$ is represented by: $\lambda s.\forall x, y.On(x, y, s) \supset \neg On(y, x, s)$. To obtain the value of a formula at a situation, we simply apply the corresponding predicate to the situation. The mental state operators will take such predicates as arguments, e.g., **Know**$(\lambda s.\forall x, y.On(x, y, s) \supset \neg On(y, x, s), S_0)$.

2.4 Agent Behavior

We have just presented a framework in which one can systematically and concisely describe the effects of actions on the world and on the mental states of multiple, communicating agents. In order to describe a multi-agent system, we must also specify what actions the agents perform.

We specify the behavior of agents with the notation of the programming language ConGolog [2], the concurrent version of Golog [8]. While versions of both Golog and ConGolog have been implemented, we are mainly interested here in the potential for using ConGolog as a specification language. The language contains the following constructs:[4]

a,	primitive action
$\phi?$,	wait for a condition
$\delta_1 ; \delta_2$,	sequence
$\delta_1 \mid \delta_2$,	nondeterministic choice of programs
$\pi x.\delta$,	nondeterministic choice of arguments
δ^*,	nondeterministic iteration
if ϕ **then** δ_1 **else** δ_2,	conditional
for $x \in L$ **do** δ,	for loop
while ϕ **do** δ,	while loop
$\delta_1 \parallel \delta_2$,	concurrency with equal priority
$\delta_1 \rangle\!\rangle \delta_2$,	concurrency with δ_1 at a higher priority
$\langle\, \vec{x} : \phi \rightarrow \delta \,\rangle$,	interrupt

In the above, a denotes a situation calculus action; ϕ denotes a formula as described in footnote 2; δ, δ_1, and δ_2 stand for complex actions; L is a finite list; and $\vec{x}$ is a sequence of variables. These constructs are mostly self-explanatory. Intuitively, the interrupts work as follows. Whenever $\exists\vec{x}.\phi$ becomes true, δ is executed with the bindings of $\vec{x}$ that satisfied ϕ; once δ has finished executing, the interrupt can trigger again.

The semantics of ConGolog programs are defined by De Giacomo et. al. [2] using a kind of semantics called *structural operational semantics* [6], which is based on "single steps" of computation, or *transitions*. A step here is either a primitive action or testing whether a condition holds in the current situation. They introduce two special predicates, $Final_{DG}$ and $Trans_{DG}$, where

[4] De Giacomo et. al. [2] allow recursive procedures in the language. To simplify matters, we omit them here. We only allow nonrecursive procedures and treat them as definitions.

$Final_{DG}(\delta, s)$ denotes that program δ may legally terminate in situation s, and where $Trans_{DG}(\delta, s, \delta', s')$ means that program δ in situation s may legally execute one step, ending in situation s' with program δ' remaining. They then define $Do_{DG}(\delta, s, s')$ to mean that s' is a terminating situation of program δ starting in situation s:

$$Do_{DG}(\delta, s, s') \stackrel{\text{def}}{=} \exists \delta'.Trans^*_{DG}(\delta, s, \delta', s') \wedge Final_{DG}(\delta', s'),$$

where $Trans^*_{DG}$ is the reflexive, transitive closure of $Trans_{DG}$.

In other words, $Do_{DG}(\delta, s, s')$ holds if it is possible to repeatedly single-step the program δ, obtaining a program δ' and a situation s' such that δ' can legally terminate in s'. The semantics does not handle the for-loop construct, since De Giacomo et. al. did not have this construct. However, it is straightforward to extend their semantics to handle for-loops.

As seen above, De Giacomo et. al. quantify over programs when defining Do_{DG}. To encode this semantics in PVS, we define a type for programs. We do this using the datatype declaration shown in Fig. 1. The declaration depends on some predefined types. As before, *Action* is the type of primitive actions. *Fluent* is the type of formulae that contain fluents, which we encode as predicates on situations, as discussed above. The π, for-loop, and interrupt operators are similar to quantifiers in that they bind variables. *QuantDom* corresponds to the domain of quantification for these operators. However, we want to allow quantification over any non-empty subtype of *QuantDom*, therefore in the datatype we use the type *NP*, which is the type of non-empty predicates on *QuantDom*. In PVS, types cannot be passed to functions, but predicates can be. A predicate can be converted to a type (i.e., the type of objects that satisfy the predicate) by enclosing it in parenthesis. For example, the arguments to the *pick* constructor are a non-empty predicate on *QuantDom*, *piPred*, and a function from (*piPred*) to programs. This is called *dependent subtyping*, since the type of an argument of a function depends on an earlier argument. It is a very useful feature of PVS. $[D \to R]$ is the type of functions from *D* to *R*. *list*[*T*] is the type of lists with elements of type *T*.

For each program construct in the language, we define a constructor, accessors, and a recognizer. They are mostly self-explanatory. *nil* is the null program that is used in defining *Trans* and *Final*. It is a constant constructor, so it has no accessors, but it has a recognizer *Null*. There is no constructor for interrupts because in the ConGolog semantics [2], they are defined in terms of other constructs.

As seen above, the π and for-loop operators bind variables. For example, in $\pi x.\delta$, x is introduced as a variable and δ is a program in which x can occur as a free variable. In the axioms for $Trans_{DG}$ and $Final_{DG}$, δ always appears in the scope of an existential quantifier that binds x. In order to directly represent variable introduction in programs, we would have to do much of the work that we avoided by modeling formulae as predicates instead of as first-order terms. Instead, we saw that the *pick* operator (and the for-loop operator) takes a predicate and a function from objects that satisfy the predicate to programs. In other words, we are representing programs with a free variable as functions from the domain of the free variable to programs (we will refer to these functions as *program functions*). We can achieve the effect of existentially quantifying over the free variable using: $\exists y.(\lambda x.\delta)(y)$, where $(\lambda x.\delta)(y)$ denotes the application of $\lambda x.\delta$ to y and yields a program.

The axioms for $Trans_{DG}$ and $Final_{DG}$ (which we do not present here due to lack of space) can easily be defined in PVS using the CASES statement which handles pattern matching over datatypes. Their encodings in PVS will be denoted *Trans* and *Final*, respectively. We encode them as recursive definitions, instead of axioms.

$Trans^*_{DG}$ is the reflexive, transitive closure of $Trans_{DG}$. Since PVS is a higher-order logic, the definition for $Trans^*_{DG}$, which is second-order, could be directly encoded in PVS. However, in order to use this definition in a proof, we would have to define a predicate in PVS to instantiate the second-order quantifier. We would then have to prove that this predicate is the reflexive, transitive closure of $Trans_{DG}$ as a lemma, and invoke this lemma each time we use the predicate to instantiate the second-order quantifier. It is more convenient to use this predicate in place of $Trans^*_{DG}$. In PVS, an infinite sequence of elements of type T can be represented by a function from *Nat* to T. We will represent a program execution by an infinite sequence of *program states*. A program state is a pair consisting of a program and a situation. We define $Trans^*$ as follows:

$$\begin{aligned}&Trans^*(\delta, s, \delta', s') \stackrel{\text{def}}{=}\\ &\quad \exists seq, n.seq(0) = (\delta, s) \wedge seq(n) = (\delta', s') \wedge\\ &\qquad\quad \forall i.i < n \supset Trans(seq(i), seq(i+1)).\end{aligned}$$

In other words, $Trans^*(\delta, s, \delta', s')$ holds if there exists an infinite sequence, seq, and a natural number, n, such that (δ, s) is the 0th element of seq, (δ', s') is the n-th element of seq, and for all $i < n$, *Trans* takes the i-th element of seq to the $i+1$-th element. Note that we are using *Trans* as a binary predicate here over pairs of program states.

This definition is equivalent to $Trans^*_{DG}$. Let $\mathrm{ConGolog}$ denote the theory generated by the *Program* datatype.

Theorem 2

$$\begin{aligned}&\mathrm{Sit} \cup \mathrm{ConGolog} \vdash_{\mathrm{pvs}}\\ &\quad \forall \delta, s, \delta', s'.Trans^*_{DG}(\delta, s, \delta', s') \equiv Trans^*(\delta, s, \delta', s')\end{aligned}$$

3. A MEETING SCHEDULER EXAMPLE

We illustrate the use of CASL by briefly describing a specification of a meeting scheduler multi-agent system that is more fully described in [15]. In this example, there are meeting organizer agents, which are trying to schedule meetings with personal agents, which manage the schedules of their (human) owners. To schedule a meeting, an organizer agent requests of each of the personal agents of the participants in the meeting to adopt the goal that its owner attend the meeting during a given time period. If a personal agent does not have any goals that conflict with its owner attending the meeting (i.e., it has not previously scheduled a conflicting meeting), it adopts the goal that its owner attend this meeting and informs the meeting organizer that it has adopted this goal, i.e., that it accepts the meeting request. Otherwise, the personal agent informs the meeting organizer that it has not adopted the goal that its owner attend the meeting, i.e., that it declines the meeting request.

The specification of the behavior of the personal agents is shown in Fig. 2. Its arguments are the personal agent and its owner. In the specification, we use the fluent $\mathrm{ATMEETING}(user, chair, s)$, which means that $user$ is at a meeting chaired by $chair$ in situation s. **During**$(period, \phi)$ means that ϕ holds throughout the time period specified by $period$. **KWhether**(agt, ϕ) holds if agt knows ϕ or knows $\neg\phi$. $\mathrm{INFORMWHETHER}(agt_1, agt_2, \phi)$ is a complex action in which agt_1 informs agt_2 whether ϕ holds.

There are two interrupts, the first running at higher priority than the second. The first interrupt fires when the agent has the goal that the user be at a meeting that starts in less than fifteen minutes, and the agent knows that the user does not yet know that it has this goal. The agent asks the user to go the meeting (actually, the agent informs the user that it wants him to go to the meeting). The second interrupt handles meeting requests; it fires when the agent knows that an organizer agent has requested a meeting, and it knows that it has not yet replied to the request. The action taken is to inform the

```
Program: DATATYPE
  BEGIN
    nil: Null                                                                  % null program used in semantics
    prim(action: Action): Primitive                                            % primitive action
    test(testPred: Fluent): Test                                               % wait for a condition
    seqn(seqFirst:Program, seqSecond:Program): Sequence                        % sequence
    nondet(ndFirst:Program, ndSecond:Program): NonDet                          % nondet. choice of actions
    pick(piPred : NP, piProg : [(piPred) → Program]): Pick                     % nondet. choice of argument
    star(starProg:Program): Star                                               % nondet. iteration
    if(ifPred:Fluent, thenProg:Program, elseProg:Program): If                  % conditional
    while(whilePred:Fluent, whileProg:Program): While                          % while loop
    for(forType: NP, objlist: list[(forPred)], forProg:[forPred → Program]): For   % for loop
    conc(concFirst:Program, concSecond:Program): Conc                          % concurrency with equal priority
    priconc(priConcFirst:Program, priConcSecond:Program): PriConc              % prioritized concurrency
  end Program
```

Figure 1: Dataype declaration for ConGolog programs.

$$
\begin{aligned}
&\textsc{manageSchedule}(pa, user) \overset{\text{def}}{=} \\
&\quad \langle period, chair\ : \\
&\qquad \mathbf{Goal}(pa, \mathbf{During}(period, \text{AtMeeting}(user, chair))) \wedge \\
&\qquad \mathbf{Know}[pa, \neg\mathbf{Know}(user, \mathbf{Goal}(pa, \mathbf{During}(period, \text{AtMeeting}(user, chair)))) \wedge \\
&\qquad\qquad\qquad \mathit{start}(period) - {:}15 \leq \textsc{time} \leq \mathit{start}(period)] \rightarrow \\
&\qquad\quad \textsc{inform}(pa, user, \mathbf{Goal}(pa, \mathbf{During}(period, \text{AtMeeting}(user, chair)))) \ \rangle \\
&\quad \rangle\!\rangle \\
&\quad \langle oa, period, chair\ : \\
&\qquad \mathbf{Know}(pa, \mathbf{Goal}(oa, \mathbf{During}(period, \text{AtMeeting}(user, chair))) \wedge \\
&\qquad\qquad\qquad \neg\mathbf{KWhether}(oa, \mathbf{Goal}(pa, \mathbf{During}(period, \text{AtMeeting}(user, chair))))) \rightarrow \\
&\qquad\quad \textsc{informWhether}(pa, oa, \mathbf{Goal}(pa, \mathbf{During}(period, \text{AtMeeting}(user, chair)))) \ \rangle
\end{aligned}
$$

Figure 2: Specification of the personal agents.

organizer whether it accepts or declines the meeting request. Due to space constraints, we omit the axiomatization of the fluents and actions for this example, and the code that specifies the behavior of the organizer agents.

A complete meeting scheduler system is defined by composing instances of the personal agents and the meeting organizer agents in parallel, thereby modeling the behavior of several agents acting independently. We also need to compose the nondeterministic iteration of a *tick* action concurrently (at a lower priority) to allow time to pass when the agents are not acting. Here is an example of such a system:

$$\begin{array}{l}[\text{MANAGESCHEDULE}(\text{PA}_1, \text{USER}_1) \parallel \\ \text{MANAGESCHEDULE}(\text{PA}_2, \text{USER}_2) \parallel \\ \text{MANAGESCHEDULE}(\text{PA}_3, \text{USER}_3) \parallel \\ \text{ORGANIZEMEETING}(\text{OA}_1, \text{USER}_1, \{\text{USER}_1, \text{USER}_3\}, \\ \qquad \text{12:00–2:00}) \parallel \\ \text{ORGANIZEMEETING}(\text{OA}_2, \text{USER}_2, \{\text{USER}_2, \text{USER}_3\}, \\ \qquad \text{1:30–2:45})] \rangle\!\rangle \text{ TICK}^*\end{array}$$

In this example, OA_1 is trying to schedule a meeting between USER_1 and USER_3 from 12 to 2. OA_2 is trying to schedule a meeting between USER_2 and USER_3 from 1:30 to 2:45. Since both meeting organizers will try to obtain USER_3's agreement for meetings that overlap, there will be two types of execution sequences, depending on who obtains this agreement.

The meeting scheduler system is easy to represent in CASLve. The only complication is in defining the domain of quantification (*QuantDom*). In the example, we need to quantify over periods of time and agents. We represent periods of time as pairs of natural numbers, while agents are declared as a primitive type. In PVS, one cannot directly define a type to be the union of other types, so we had to define a datatype with constructors for both types. This means that when defining the system, we have to use constructors and accessors to map into the *QuantDom* and back to the original types.

4. VERIFICATION

Let us now discuss the PVS-based verification environment for CASL that we have developed, which we call CASLve. PVS has high-level proof-strategies and decision procedures that take care of many of the low-level details associated with computer-aided theorem proving. Some simple proofs (including some inductive proofs) can be handled using a single application of a proof strategy. Many proofs can be accomplished using only the following steps: lemma introduction (here lemma is a general term for any proposition: axioms, lemmas, theorems, etc., and includes induction axioms), definition expansion, quantifier instantiation, and simplification. In addition, PVS has useful proof-management facilities, including a graphical display of the proof tree, and proof stepping and editing.

We have used CASLve to prove many lemmas that are useful in verifying properties of programs. In the remainder of this section, we will discuss one such lemma, namely that all bounded-loop programs terminate. For the purposes of this paper, we consider a bounded-loop program to be one without while-loops and nondeterministic iteration (but for-loops are allowed): $\mathit{Bounded}(\delta) \stackrel{\text{def}}{=} \forall \delta'.\mathit{Subterm}(\delta', \delta) \supset \neg \mathit{Star}(\delta') \wedge \neg \mathit{While}(\delta')$. If we want to prove that this property holds for some program, we run into a problem involving the use of program functions as a way of handling program operators that bind variables. We intend to use functions such as: $\lambda x.\mathit{seqn}(\mathit{test}(\text{ONTABLE}(x)), \text{REMOVE}(x))$, which when applied to an object such as BLOCK1 simulates the substitution of BLOCK1 for x. However, there is nothing to stop us from defining a function $f : \text{NAT} \rightarrow \text{PROGRAM}$ such that $f(i) = a^i$, where a^i denotes the sequential composition of a with itself i times. Then, $\mathit{pick}(\mathit{Nat}, f)$ is essentially an unbounded program because we cannot bound in advance the number of primitive actions that it will execute.

Our solution to this problem is to restrict the program functions to functions that always return programs that have the same structure. We first define a congruence relation $\mathit{Congruent}(\delta, \delta')$ that holds if two programs have the same structure. This relation is defined recursively and it checks that the outermost operator is the same for each program and then recursively checks that the rest of the programs are congruent. In addition, if δ and δ' are of the form $\mathit{pick}(NP_1, pf_1)$, and $\mathit{pick}(NP_2, pf_2)$, respectively, then we require that $NP_1 = NP_2$ and $\forall x : (NP_1).\mathit{Congruent}(pf_1(x), pf_2(x))$. We have a similar requirement for for-loops, but we also require that the two lists be of the same length. We omit the formal definition of this relation here. We say that a program δ is *suitable* ($\mathit{SProg}(\delta)$), if for any subprogram of δ that is of the form $\mathit{pick}(NP, pf)$ or $\mathit{for}(NP, l, pf)$ all the instantiations of pf are congruent, i.e., $\forall x, y : (NP).\mathit{Congruent}(pf(x), pf(y))$. Again, we omit the formal definition. We will limit our attention to suitable programs. We can do this because they are closed under transitions:

Theorem 3

$$\begin{array}{l}\mathrm{Sit} \cup \mathrm{ConGolog} \vdash_{\mathrm{pvs}} \\ \quad \forall \delta, \delta', s, s'.\mathit{SProg}(\delta) \wedge \mathit{Trans}(\delta, s, \delta', s') \supset \mathit{SProg}(\delta')\end{array}$$

All future quantifications over programs will be assumed to be restricted to suitable programs.

Following Francez [5], we say that a program δ *terminates* starting in situation s if it has no infinite executions starting in s. We can use the notion of a sequence of program states introduced for our definition of Trans^* in Sec. 2.4 to talk about infinite executions. We say that δ has an infinite execution starting in s, if there is a infinite sequence of program states that starts with (δ, s), such that *Trans* holds of each adjacent pair of states: $\mathit{InfExec}(\delta, s) \stackrel{\text{def}}{=} \exists \mathit{seq}.\mathit{seq}(0) = (\delta, s) \wedge \forall i.\mathit{Trans}(\mathit{seq}, i)$, where *seq* ranges over functions from the natural numbers to program states, and $\mathit{Trans}(\mathit{seq}, i)$ holds if there is a transition from the i-th to the $i+1$-th element of *seq*. Note that this is an overloading of the previously defined binary *Trans* predicate. Now, we can define termination as the absence of an infinite execution: $\mathit{Terminates}(\delta, s) \stackrel{\text{def}}{=} \neg \mathit{InfExec}(\delta, s)$.

We adapt a technique from Francez [5], to assist in proving termination of a program δ starting in situation s. The idea is to find a predicate $P_{seq} \subseteq \text{NAT} \times \text{NAT}$ that is intuitively a measure on an execution seq of δ (where the sequence is understood from the context, we drop the subscript). Intuitively, $P(i, j)$ holds if the i-th step of seq has measure j. We can infer that $\mathit{Terminates}(\delta, s)$, if for any sequence seq such that $seq(0) = (\delta, s)$, there is a P such that:

1. $P(0, j)$ holds for some j,
2. the value of the measure strictly decreases with each transition step of the sequence, and
3. when the measure reaches 0, i.e., $P(i, 0)$ for some i, then there is no legal transition from $seq(i)$ to $seq(i+1)$,

We can state this formally as follows:

Theorem 4

$$\begin{array}{l}\mathrm{Sit} \cup \mathrm{ConGolog} \vdash_{\mathrm{pvs}} \\ \quad \forall \delta, s[\forall seq.seq(0) = (\delta, s) \supset \\ \quad\quad (\exists P.(\exists j.P(0,j)) \wedge \\ \quad\quad\quad (\forall i, j.j > 0 \wedge P(i,j) \wedge \mathit{Trans}(seq, i) \supset \\ \quad\quad\quad\quad \exists k.k < j \wedge P(i+1,k)) \wedge \\ \quad\quad\quad (\forall i.P(i,0) \supset \neg \mathit{Trans}(seq,i)))] \supset \mathit{Terminates}(\delta, s)\end{array}$$

Note that we use a relation for the measure because we do not want to require that the measure be defined over all natural numbers. In particular, once the measure reaches 0, we want to allow it to be undefined from then on. If we can find such a measure for any bounded program δ and situation s, we can show that all bounded programs terminate. The measure that we use is based on the length of δ, where the length of a program is the maximum number of primitive actions and tests that will be generated in any execution of the program. We informally describe the *proglen* function which maps a bounded program to its length, but omit its formal definition. It is defined recursively. *nil* has length 0. Primitive actions and tests have length 1. The length of sequential, concurrent, and prioritized concurrent compositions are the sum of the lengths of their arguments. The length of nondeterministic choice of programs and if-then-else statements are the maximum of the lengths their (program) arguments. Since we are only considering bounded, suitable programs, the length of the program that results from applying the program function argument of a *pick* statement to an object will be the same for all objects. Therefore, the length of a *pick* statement is the length of the program that results from applying the program function to an arbitrary object. Similarly, the length of a for-loop is the (list) length of its list argument multiplied by the (program) length of its program function argument applied to an arbitrary object.

We will now define a relational measure, P_{seq}, that uses the program length. We want the measure to be defined initially and as long as the sequence continues to be a valid execution of the program. Where it is defined, $P_{seq}(i,j)$ will hold if j is the program length of the program component of $seq(i)$: $P_{seq} \stackrel{\text{def}}{=} \lambda i, j.(\forall k.k < i \supset \mathit{Trans}(seq,k)) \wedge j = \mathit{proglen}(\mathit{pj}_1(seq(i)))$, where pj_1 projects out the first element of a pair. We use this measure to show that all bounded programs terminate:

Theorem 5

$$\mathrm{Sit} \cup \mathrm{ConGolog} \vdash_{\mathrm{pvs}} \forall \delta, s.\mathit{Bounded}(\delta) \supset \mathit{Terminates}(\delta, s)$$

We did not show the specification of the meeting organizer agents here, but they are specified with bounded-loop programs, so it is easy to show that they terminate. To show that the entire meeting scheduler system terminates,[5] we also need to show that the personal agent processes terminate. Although the personal agents are not specified with bounded-loop programs, they only perform actions in response to actions performed by the organizer agents, and each such response consists of a finite set of actions. Since the organizer agent processes terminate, it should follow that the entire system terminates. In future work, we plan to complete a formal proof in CASLve that the meeting scheduler system terminates following this line of reasoning.

[5] Note that the instance of the meeting scheduler system specified above does *not* terminate, since the TICK* process has an infinite execution. This can easily be fixed by replacing this process by one that has a time limit.

5. EXAMPLE PROOF

To illustrate CASLve, we will run through part of a proof. Since we are presenting parts of a PVS proof, we will use PVS notation, i.e., a sequent calculus with a typed, higher-order language. The proof we illustrate is a lemma that is used in the proof of Theorem 5, i.e., that all bounded programs terminate. The lemma says that all legal transitions of bounded programs result in programs of smaller length; it is stated formally in the first sequent below.

When the PVS prover is invoked, one enters the proof mode with a single sequent that contains only the proposition to be proved in the consequent. The antecedent formulae are numbered with negative integers, while the consequent are numbered with positive integers.

$$\begin{array}{|ll}\hline \{1\} & \forall(\delta : \mathit{Bounded}), (\delta' : \mathit{Program}), (s, s' : \mathit{Sit}) : \\ & \quad \mathit{Trans}(\delta, s, \delta', s') \supset \mathit{proglen}(\delta') < \mathit{proglen}(\delta)\end{array}$$

The proof proceeds by induction over δ. The PVS command for this is: (INDUCT δ 1), which is a strategy that sets up a proof of formula $\{1\}$ by induction over δ. Since δ is of type *Program*, PVS sets up the induction by creating a new sequent to prove for each program construct, and possibly some sequents to prove type correctness conditions. Since we do not have much space, we will only show the proof of one of the cases. We will show the proof for tests. Recall from Fig. 1 that the program construct for tests is $\mathit{test}(\phi)$, where ϕ is a fluent, i.e., a predicate on situations. The sequent that PVS generates for this case is as follows.

$$\begin{array}{|ll}\hline \{1\} & \forall(\phi : \mathit{Fluent}) : \mathit{Bounded}(\mathit{test}(\phi)) \supset \\ & \quad \forall(\delta' : \mathit{Program}), (s, s' : \mathit{Sit}) : \\ & \quad\quad \mathit{Trans}(\mathit{test}(\phi), s, \delta', s') \supset \\ & \quad\quad\quad \mathit{proglen}(\delta') < \mathit{proglen}(\mathit{test}(\phi))\end{array}$$

Next, we simplify the sequent with the PVS command (REDUCE NIL). REDUCE is a strategy that performs various simplifications, including skolemization, propositional simplification, applying decision procedures, and equality replacement. The NIL parameter is used to prevent heuristic quantifier instantiation. Skolem constants are formed by adding subscripted numerals to variable names. The following sequent is the result of the simplification.

$$\begin{array}{|ll}\{\text{-}1\} & \mathit{Bounded}(\mathit{test}(\phi_1)) \\ \{\text{-}2\} & \mathit{Trans}(\mathit{test}(\phi_1), s_1, \delta'_1, s'_1) \\ \hline \{1\} & \mathit{proglen}(\delta'_1) < \mathit{proglen}(\mathit{test}(\phi_1))\end{array}$$

In our encoding of ConGolog, we made *Trans* a PVS definition. The definition states that $\mathit{Trans}(\mathit{test}(\phi_1), s_1, \delta'_1, s'_1)$ holds iff $\phi_1(s_1)$ holds and $s_1 = s'_1$ and $\delta'_1 = \mathit{nil}$. The next step of the proof is to expand the definition of *Trans*, which yields the following sequent.

$$\begin{array}{|ll}[\text{-}1] & \mathit{Bounded}(\mathit{test}(\phi_1)) \\ \{\text{-}2\} & \phi_1(s_1) \wedge \delta'_1 = \mathit{nil} \wedge s'_1 = s_1 \\ \hline [1] & \mathit{proglen}(\delta'_1) < \mathit{proglen}(\mathit{test}(\phi_1))\end{array}$$

If a formula's number is enclosed in square brackets, it means that the formula has not changed from the previous sequent. In the definition of *proglen*, the *nil* program is given length 0 and a program consisting of only a test is given length 1. Therefore, after simplifying and expanding the definition of *proglen*, we obtain the following sequent.

$$\begin{array}{|ll}[\text{-}1] & \mathit{Bounded}(\mathit{test}(\phi_1)) \\ [\text{-}2] & \phi_1(s_1) \\ [\text{-}3] & \delta'_1 = \mathit{nil} \\ [\text{-}4] & s'_1 = s_1 \\ \hline \{1\} & 0 < 1\end{array}$$

It is easy to see that this sequent is true, and we can use the (GROUND) command, which simplifies using decision procedures, to complete the proof. We have illustrated some of the main steps used in CASLve. The other ones that are used most often are quantifier instantiation, lemma introduction, and GRIND, which is a strategy that repeatedly performs heuristic instantiation of quantifiers and simplification and can complete many simple proofs. PVS also has a facility for creating user-defined strategies, which we would like to use to develop strategies specifically for CASL to further facilitate the verification of CASL specifications.

6. CONCLUSION AND FUTURE WORK

We have presented the different aspects of the CASL specification language and how we encoded them in PVS to form the basis of a verification environment. We briefly described the specification of a meeting scheduler multi-agent system in CASL, and we showed that all bounded-loop programs terminate, which is a useful lemma for proving termination of CASL programs. To our knowledge, CASL is the only multiagent specification language with a verification environment that uses theorem proving. There are attempts to use model-checking to automatically verify properties of multi-agent systems[1], but that approach limits the expressiveness of the specification language, which can make it more difficult to specify and verify complex multiagent systems.

We plan to complete the termination proof for the meeting scheduler system as discussed above. We would also like to prove partial correctness properties of this system. The techniques for compositional verification of multi-agent systems developed by Engelfriet et. al. [4] may prove useful here. In addition, we would like to handle recursive procedures in CASL using the framework of De Giacomo et. al. [2].

7. REFERENCES

[1] M. Benerecetti, F. Giunchiglia, and L. Serafini. Model checking multiagent systems. *Journal of Logic and Computation*, 8(3):401–423, 1998.

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

[3] G. De Giacomo and H. J. Levesque. Progression using regression and sensors. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 160–165, Stockholm, Sweden, 1999.

[4] J. Engelfriet, C. M. Jonker, and J. Treur. Compositional verification of multi-agent systems in temporal multi-epistemic logic. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V: Proceedings of the Fifth International Workshop on Agent Theories, Architectures and languages (ATAL'98)*, volume 1555 of *LNAI*, pages 177–194. Springer-Verlag, 1999.

[5] N. Francez. *Fairness*. Springer-Verlag, New York, 1986.

[6] M. Hennessy. *The Semantics of Programming Languages*. John Wiley & Sons, 1990.

[7] G. Lakemeyer and H. J. Levesque. AOL: a logic of acting, sensing, knowing, and only knowing. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR-98)*, pages 316–327, 1998.

[8] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.

[9] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*. Edinburgh University Press, 1969.

[10] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

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

[12] R. B. Scherl and H. J. Levesque. The frame problem and knowledge-producing actions. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 689–695, Washington, DC, July 1993. AAAI Press/The MIT Press.

[13] S. Shapiro. PhD thesis. In preparation.

[14] S. Shapiro and Y. Lespérance. Modeling multiagent systems with the cognitive agents specification language — a feature interaction resolution application. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents Volume VII — Proceedings of the 2000 Workshop on Agent Theories, Architectures, and Languages (ATAL-2000)*, LNAI. Springer-Verlag, Berlin, 2001. To appear.

[15] S. Shapiro, Y. Lespérance, and H. J. Levesque. Specifying communicative multi-agent systems. In W. Wobcke, M. Pagnucco, and C. Zhang, editors, *Agents and Multi-Agent Systems — Formalisms, Methodologies, and Applications*, volume 1441 of *LNAI*, pages 1–14. Springer-Verlag, Berlin, 1998.

[16] S. Shapiro, M. Pagnucco, Y. Lespérance, and H. J. Levesque. Iterated belief change in the situation calculus. In A. G. Cohn, F. Giunchiglia, and B. Selman, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Seventh International Conference (KR2000)*, pages 527–538, San Francisco, CA, 2000. Morgan Kaufmann Publishers.