Situation Calculus-based Programs for Representing and Reasoning about Game Structures

Giuseppe De Giacomo
Dip. Informatica e Sistemistica
Sapienza Università di Roma
Roma, Italy
degiacomo@dis.uniroma1.it

Yves Lespérance
Dept. of Computer Sci. & Eng.
York University
Toronto, Canada
lesperan@cse.yorku.ca

Adrian R. Pearce
Dept. of Computer Sci. & Software Eng.
University of Melbourne
Victoria, Australia
adrianrp@unimelb.edu.au

Abstract

A wide range of problems, from contingent and multiagent planning to process/service orchestration, can be viewed as games. In many of these, it is natural to specify the possible behaviors procedurally. In this paper, we develop a logical framework for specifying these types of problems/games based on the situation calculus and ConGolog. The framework incorporates game-theoretic path quantifiers as in ATL. We show that the framework can be used to model such problems in a natural way. We also show how verification/synthesis techniques can be used to solve problems expressed in the framework. In particular, we develop a method for dealing with infinite state settings using fixpoint approximation and "characteristic graphs".

Introduction

Many types of problems, from contingent and multiagent planning to process/service orchestration, can be viewed as games, where one or more agents try to ensure that certain objectives hold no matter how the environment and other agents behave. There has been much work recently on developing logical formalisms for specifying such game structures and the properties that coalitions of agents can ensure in them, e.g., (Ruan, van der Hoek, and Wooldridge 2009), mostly based on Alternating-Time Temporal Logic (ATL) (Alur, Henzinger, and Kupferman 2002). It has also been shown that model checking techniques can be used to verify that such properties hold in a game structure. As well, these techniques can be used to synthesize strategies for the agents in the coalition to ensure that the objectives hold (Lomuscio, Qu, and Raimondi 2009).

Logics like ATL, ATL*, and the alternating-time μ-calculus (AMC) (Alur, Henzinger, and Kupferman 2002) provide elegant and rather expressive languages for specifying the properties that one wants to verify. However, these logics do not address how one specifies the model/game structure over which the property is to be verified, or the strategies that agents may follow. The languages in common use for this arose with model checking technology and tend to be quite low level and, more importantly, restricted to finite states structures. Also, in many applications, it is natural to specify the possible behaviors of the agents/players by combining declarative and procedural elements. This suggests incorporating an action/programming language into the logical framework.

In this paper, we develop a logical framework for specifying and solving game theoretic problems based on the situation calculus and the ConGolog agent programming language (De Giacomo, Lespérance, and Levesque 2000). We adapt ConGolog to precisely specify which agent can make a choice in any given situation. A program in the resulting language, called *GameGolog*, can be used to conveniently specify a game structure, making use of a background situation calculus action theory in doing this. For specifying properties to be verified, we use a very rich language that combines the μ-calculus, game-theoretic path quantifiers, and first-order quantification.

We show how our logical framework can be used to model in a natural way and solve a range of problems from simple games, to contingent planning, and process/service orchestration. To do this, we show how existing formalizations of contingent planning (Lespérance, De Giacomo, and Ozgovde 2008) and process/service orchestration (De Giacomo and Sardina 2007) can be translated into our framework. We also discuss how constraints such as forms of fairness can be expressed and used in verifying properties.

The verification method we propose is inspired from ATL symbolic model checking approaches (Alur, Henzinger, and Kupferman 2002; Lomuscio, Qu, and Raimondi 2009). However, our method has to deal with incomplete specification of the game structure, which in our case is a theory and not a single model. Moreover, as usual in the situation calculus, we allow for first-order quantification and infinite states settings. To deal with this, our method for verification in infinite states settings uses fixpoint approximation and "characteristic graphs" (Claßen and Lakemeyer 2008).

We stress that the aim of this work is to support reasoning about game structures, not just games in the conventional sense, but any type of multiagent problem that requires strategic thinking. Our framework is not based on classical game theory and does not use probabilities or utilities. Our focus is instead on multiagent interaction problems that are naturally specified in a language that combines declarative and procedural elements.

The rest of the paper is organized as follows. First, we review the essentials of the situation calculus and ConGolog. Then, we present our approach to modeling and verifying game structures in the situation calculus and propose an ATL-like language for specifying properties to be verified. We illustrate the approach by specifying the game tic-tac-toe and expressing and verifying some of its properties. After this, we define GameGolog and show how it can be used to specify and verify game structures conveniently. Then we discuss how our framework can be used to conveniently model contingent planning and process orchestration problems. We conclude by discussing related work, considering how one can do synthesis in our framework, and pointing out some issues for future work.

Preliminaries

The Situation Calculus and Basic Action Theories. The *situation calculus* is a logical language specifically designed for representing and reasoning about dynamically changing worlds (Reiter 2001). All changes to the world are the result of *actions*, which are terms in the language. We denote action variables by lower case letters a, action types by capital letters A, and action terms by α, possibly with subscripts. A possible world history is represented by a term called a *situation*. The constant S_0 is used to denote the initial situation where no actions have yet been performed. Sequences of actions are built using the function symbol do, such that $do(a, s)$ denotes the successor situation resulting from performing action a in situation s. Predicates and functions whose value varies from situation to situation are called *fluents*, and are denoted by symbols taking a situation term as their last argument (e.g., $Holding(x, s)$).

Within the language, one can formulate action theories that describe how the world changes as the result of the available actions. Here, we concentrate on *basic action theories* as proposed in (Pirri and Reiter 1999; Reiter 2001). We also assume that there is a *finite number of action types* in the domains that we consider. As a result a basic action theory $\mathcal{D}$ is the union of the following disjoint sets: the foundational, domain independent, axioms of the situation calculus (Σ); precondition axioms stating when actions can be legally performed ($\mathcal{D}_{poss}$); successor state axioms describing how fluents change between situations ($\mathcal{D}_{ssa}$); unique name axioms for actions and domain closure on action types ($\mathcal{D}_{ca}$); and axioms describing the initial configuration of the world ($\mathcal{D}_{S_0}$). A special predicate *Poss*(a, s) is used to state that action a is executable in situation s; precondition axioms in $\mathcal{D}_{poss}$ characterize this predicate. In turn, successor state axioms encode the causal laws of the world being modeled; they take the place of the so-called effect axioms and provide a solution to the frame problem.

High-Level Programs. To represent and reason about complex actions or processes obtained by suitably executing atomic actions, various so-called *high-level programming languages* have been defined. Here we concentrate on a fragment of ConGolog, which includes most constructs of the language, except for (recursive) procedures:

α	atomic action
$\varphi?$	test for a condition
$\delta_1; \delta_2$	sequence
if φ **then** δ_1 **else** δ_2	conditional
while φ **do** δ	while loop
$\delta_1 \vert \delta_2$	nondeterministic branch
$\pi x.\delta$	nondeterministic choice of argument
δ^*	nondeterministic iteration
$\delta_1 \Vert \delta_2$	concurrency

In the above, α is an action term, possibly with parameters, and φ is situation-suppressed formula, that is, a formula in the language with all situation arguments in fluents suppressed. We denote by $\varphi[s]$ the situation calculus formula obtained from φ by restoring the situation argument s into all fluents in φ. Program $\delta_1|\delta_2$ allows for the nondeterministic choice between programs δ_1 and δ_2, while $\pi x.\delta$ executes program δ for *some* nondeterministic choice of a legal binding for variable x (observe that such a choice is, in general, unbounded). δ^* performs δ zero or more times. Program $\delta_1\|\delta_2$ expresses the concurrent execution (interpreted as interleaving) of programs δ_1 and δ_2.

Formally, the semantics of ConGolog is specified in terms of single-step transitions, using the following two predicates (De Giacomo, Lespérance, and Levesque 2000): *(i)* $Trans(\delta, s, \delta', s')$, which holds if one step of program δ in situation s may lead to situation s' with δ' remaining to be executed; and *(ii)* $Final(\delta, s)$, which holds if program δ may legally terminate in situation s. The definitions of $Trans$ and $Final$ for the constructs used in this paper are shown below:

$$
\begin{array}{l}
Trans(\alpha, s, \delta', s') \equiv \\
\qquad s' = do(\alpha, s) \wedge Poss(\alpha, s) \wedge \delta' = True? \\
Trans(\varphi?, s, \delta', s') \equiv False \\
Trans(\delta_1; \delta_2, s, \delta', s') \equiv \\
\qquad Trans(\delta_1, s, \delta_1', s') \wedge \delta' = \delta_1'; \delta_2 \vee \\
\qquad Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s') \\
Trans(\textbf{if } \varphi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s, \delta', s') \equiv \\
\qquad \varphi[s] \wedge Trans(\delta_1, s, \delta', s') \vee \\
\qquad \neg\varphi[s] \wedge Trans(\delta_2, s, \delta', s') \\
Trans(\textbf{while } \varphi \textbf{ do } \delta, s, \delta', s') \equiv \\
\qquad \varphi[s] \wedge Trans(\delta, s, \delta'', s') \wedge \delta' = \delta''; (\textbf{while } \varphi \textbf{ do } \delta) \\
Trans(\delta_1|\delta_2, s, \delta', s') \equiv \\
\qquad Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s') \\
Trans(\pi x.\delta, s, \delta', s') \equiv \exists x.Trans(\delta, s, \delta', s') \\
Trans(\delta^*, s, \delta', s') \equiv Trans(\delta, s, \delta'', s') \wedge \delta' = \delta''; \delta^* \\
Trans(\delta_1\|\delta_2, s, \delta', s') \equiv \\
\qquad Trans(\delta_1, s, \delta_1', s') \wedge \delta' = \delta_1'\|\delta_2 \vee \\
\qquad Trans(\delta_2, s, \delta_2', s') \wedge \delta' = \delta_1\|\delta_2'
\end{array}
$$

$$
\begin{array}{l}
Final(\alpha, s) \equiv False \\
Final(\varphi?, s) \equiv \varphi[s] \\
Final(\delta_1; \delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s) \\
Final(\textbf{if } \varphi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s) \equiv \\
\qquad \varphi[s] \wedge Final(\delta_1, s) \vee \neg\varphi[s] \wedge Final(\delta_2) \\
Final(\textbf{while } \varphi \textbf{ do } \delta, s) \equiv \varphi[s] \wedge Final(\delta, s') \vee \neg\varphi[s] \\
Final(\delta_1|\delta_2, s) \equiv Final(\delta_1, s) \vee Final(\delta_2, s) \\
Final(\pi x.\delta, s) \equiv \exists x.Final(\delta, s) \\
Final(\delta^*, s) \equiv True \\
Final(\delta_1\|\delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)
\end{array}
$$

The definitions of $Trans$ and $Final$ we use are as in (Sardina and De Giacomo 2009); these are in fact the usual ones (De Giacomo, Lespérance, and Levesque 2000), except that, following (Claßen and Lakemeyer 2008), the test construct $\varphi?$ does not yield any transition, but is final when satisfied. Thus, it is a *synchronous* version of the original test construct (it does not allow interleaving). Also, as in (Sardina and De Giacomo 2009), we require that in programs of the form $\pi x.\delta$, the variable x occurs in some non-variable action term in δ; we disallow cases where x occurs *only* in tests or as an action itself. In this way, $\pi x.\delta$ acts as a construct for making nondeterministic choices of action parameters (possibly constrained by tests). Finally, we assume without loss of generality that each occurrence of the construct $\pi x.\delta$ in a program uses a unique fresh variable x—no two occurrences of such a construct use the same variable.

Situation Calculus Game Structures

To model games, we use a specialization of the situation calculus in which every action has an agent parameter. The function *agent()* takes an action, a, and returns the agent of the action; we provide axioms specifying it for every action type. By convention, the agent will usually be the first argument of the action type. We assume that that there is a finite set of agents *Agents*, which are denoted by a set of unique names.

Actions are partitioned into two classes: choice actions and standard actions. *Choice actions* are special actions that are used to model the decisions of agents. We assume that choice actions have no effects on any fluents, other than those defined below. $Poss(a, s)$ corresponds to the notion of an action a being physically possible (i.e. executable) in situation s. We take choice actions to be always physically possible. However, we introduce a stronger version of possibility/legality that is used to model the structure of the game setting of interest. We specify such a notion using a special predicate *Legal*, modeling the ability of agents to perform actions and take decisions *according to the rules of the game*. *Legal* must be axiomatized on a case by case basis according to the game being modeled. But, we require that the axiomatization of *Legal* entail the following properties:

1. $Legal$ implies physically possible/executable:

$$Legal(s) \supset s = S_0 \vee \exists a, s'.\, s' = do(a, s) \wedge Poss(s').$$

2. If we are in a legal situation after an action, then before the action we were in a legal situation as well:

$$Legal(s) \supset s = S_0 \vee \exists a, s'.\, s' = do(a, s) \wedge Legal(s').$$

3. In a legal situation, only one agent can act:

$$Legal(do(a, s) \wedge Legal(do(a', s) \supset agent(a) = agent(a').$$

For convenience, we also introduce the predicate *Control* that given a legal situation returns the agent that can act in that situation:

$$Control(agt, s) \doteq \exists a.\, Legal(do(a, s)) \wedge agent(a) = agt.$$

Note that $Control(agt, s) \wedge Control(agt', s) \supset agt = agt'$ is entailed, since the constraints on *Legal* imply that only one agent can act in a legal situation.

We note that there are clearly games where several agents can act simultaneously. We can model such games using a sort of round robin of choice actions among the agents involved. Also, there are games where several agents may try to act and nondeterministically one player will succeed in performing his action. We can model this case by adding to the game an extra player, a sort of game master, who is in charge of making the "nondeterministic" decision, i.e., of deciding which agent will actually act among all those that may act (and we record such decisions in the situation).

We call the resulting variant of basic action theories *situation calculus game structures*. A situation calculus game structure $\mathcal{D}_{GS} = \Sigma \cup \mathcal{D}_{poss} \cup \mathcal{D}_{ssa} \cup \mathcal{D}_{ca} \cup \mathcal{D}_{S_0} \cup \mathcal{D}_{legal}$, where $\Sigma \cup \mathcal{D}_{poss} \cup \mathcal{D}_{ssa} \cup \mathcal{D}_{ca} \cup \mathcal{D}_{S_0}$ are as for standard basic action theories, cf. Preliminaries, and $\mathcal{D}_{legal}$ denotes the axioms for $Legal$ and $Control$, as well as the definition of the function $agent()$. Observe that in the literature, the term game structure usually refers to a single model; here instead, it stands for a type of situation calculus theory.

Example 1 To illustrate, we use the well-known tic-tac-toe game as a running example. The state of the game, in situation s, is captured by fluents, $Cell(m, r, c, s)$ that represent the mark, m, either nought, O, cross X, or blank, B; one for each of the row, $1 \leq r \leq 3$, and column, $1 \leq c \leq 3$, positions. Initially, the whole board is blank:

$$\forall r, c.InRange(r, c) \supset Cell(B, r, c, S_0)$$

where $InRange(r, c) \doteq (1 \leq r \leq 3) \wedge (1 \leq c \leq 3)$.

Players perform actions $move(O, r, c)$ or $move(X, r, c)$ at a particular cell. They can move on any blank cell position, so we define possible moves as follows:

$$\begin{array}{l} Poss(move(m, r, c), s) \equiv \\ \quad (m = X \vee m = O) \wedge Cell(B, r, c, s) \wedge InRange(r, c). \end{array}$$

We specify the successor state axiom

$$\begin{array}{l} Cell(m, r, c, do(a, s)) \equiv \\ \quad a = move(m, r, c) \vee Cell(m, r, c, s), \end{array}$$

capturing how fluent $Cell(m, r, c, s)$ becomes true either if a player performs an action to move into that cell or remains true if that cell was in the same state in the previous situation, s, and the player (implicitly) moved into a different cell. The function $agent()$ is specified as follows:

$$agent(move(m, r, c)) = m.$$

Given the precondition axiom, m can only be X or O.

For defining legality it is convenient to introduce the following abbreviation:

$$\begin{array}{l} Completed(s) \doteq \\ \quad \forall r, c.InRange(r, c) \wedge Cell(m, r, c, s) \supset m \neq B. \end{array}$$

Then, we can write:

$$\begin{array}{l} Legal(s) \equiv S_0 \vee \\ \quad \exists a.(s = do(a, S_0) \wedge Poss(a, S_0) \wedge agent(a) = X) \vee \\ \quad \exists a, b, s'.s = do(b, do(a, s') \wedge Poss(b, do(a, s')) \wedge \\ \qquad Poss(a, s') \wedge \neg Completed(do(a, s')) \wedge \\ \qquad (agent(a) = X \wedge agent(b) = O \vee \\ \qquad\; agent(a) = O \wedge agent(b) = X). \end{array}$$

capturing that a situation s is legal if it is either the initial situation or a situation reached by alternating X and O moves starting with X, and stopping the alternation as soon as the board does not contain any more blanks ($Completed$ holds).

Finally, we define a convenient abbreviation for denoting the winning condition for the tic-tac-toe game:

$$\begin{array}{l} Wins(m, s) \doteq \\ \quad \exists r \bigwedge_{1 \leq c \leq 3} Cell(m, r, c) \vee \\ \quad \exists c \bigwedge_{1 \leq r \leq 3} Cell(m, r, c) \vee \\ \quad \bigwedge_{1 \leq i \leq 3} Cell(m, i, i) \vee \\ \quad \bigwedge_{1 \leq i \leq 3} Cell(m, i, 4 - i) \end{array}$$

We also model the game being finished by: $Finished(s) \doteq Completed(s) \lor Wins(X,s) \lor Wins(O,s)$. □

To express properties about these kinds of game structures, we introduce a specific logic, inspired by ATL (Alur, Henzinger, and Kupferman 2002), and more generally by the μ-calculus (Bradfield and Stirling 2007) over the game structures (de Alfaro, Henzinger, and Majumdar 2001) used, e.g., in LTL synthesis by model checking (Piterman, Pnueli, and Sa'ar 2006). The key building block of our logic is the following operator:

$$
\begin{array}{l}
\langle\langle G\rangle\rangle \bigcirc \varphi \doteq \\
\quad (\exists agt \in G.\, Control(agt, now) \land \\
\qquad \exists a.\, agent(a) = agt \land \\
\qquad\quad Legal(do(a, now)) \land \varphi[do(a, now)]) \lor \\
\quad (\exists agt \notin G.\, Control(agt, now) \land \\
\qquad \forall a.\, agent(a) = agt \land \\
\qquad\quad Legal(do(a, now)) \supset \varphi[do(a, now)])
\end{array}
$$

In the above, φ is a (possibly open) situation suppressed formula, and now is a placeholder for the suppressed situation in the formula. Note that here, we quantify differently according to whether or not the agent controlling the situation is in the coalition G. In the first case, for an agent $agt \in G$, the quantification is existential: we look for an action that makes φ true. In the second case, i.e. for $agt \notin G$, we require that all actions that the controlling agent can do make φ true. In this way, we allow an agent $agt \in G$ to select a play against all possible moves of other agents not in G.

With this operator at hand, we develop the whole logic $\mathcal{L}$ based on the μ-calculus (Park 1976; Bradfield and Stirling 2007). $\mathcal{L}$ is defined as follows:

$$
\begin{array}{l}
\Psi \leftarrow \varphi \mid Z(\vec{x}) \mid \Psi_1 \land \Psi_2 \mid \Psi_1 \lor \Psi_2 \mid \exists x.\Psi \mid \forall x.\Psi \mid \\
\langle\langle G\rangle\rangle \bigcirc \Psi \mid [[G]] \bigcirc \Psi \mid \mu Z(\vec{x}).\Psi(Z(\vec{x})) \mid \nu Z(\vec{x}).\Psi(Z(\vec{x}))
\end{array}
$$

where φ is an arbitrary, possibly open, situation-suppressed situation calculus uniform formula, Z is a predicate variable of a given arity, $\langle\langle G\rangle\rangle \bigcirc \Psi$ is as defined above, $[[G]] \bigcirc \Psi$ is the dual of $\langle\langle G\rangle\rangle \bigcirc \Psi$ (i.e., $[[G]] \bigcirc \Psi \equiv \neg\langle\langle G\rangle\rangle \bigcirc \neg\Psi$[1]), μ (resp. ν) is the least (resp. greatest) fixpoint operator from the μ-calculus, and $\Psi(Z(\vec{x}))$ is a notation used to emphasize that $Z(\vec{x})$ may occur free, i.e., not quantified by μ or ν in Ψ.

We can express arbitrary temporal/dynamic properties using least and greatest fixpoint constructions. For instance, to say that group G has a strategy to achieve $\varphi(\vec{x})$, where $\varphi(\vec{x})$ is a situation suppressed formula with free variables $\vec{x}$, we use the following least fixpoint construction:

$$\langle\langle G\rangle\rangle\Diamond\varphi(\vec{x}) \doteq \mu Z(\vec{x}).\, \varphi(\vec{x}) \lor \langle\langle G\rangle\rangle \bigcirc Z(\vec{x})$$

Similarly, we use a greatest fixpoint construction to express the ability of a coalition G to maintain a property φ:

$$\langle\langle G\rangle\rangle\Box\varphi(\vec{x}) \doteq \nu Z(\vec{x}).\varphi(\vec{x}) \land \langle\langle G\rangle\rangle \bigcirc Z(\vec{x})$$

Example 2 In the tic-tac-toe game, an interesting property to check is the existence of a strategy to win the game for player m. This can be formalized in our logic as:

$$\mu Z.\, Wins(m) \lor \langle\langle m\rangle\rangle \bigcirc Z$$

i.e., $\langle\langle m\rangle\rangle\Diamond Wins(m)$. Now it is well known that starting from the bank board neither of the players has a strategy to ensure a win, and indeed, we have that $\mathcal{D}_{TTT} \not\models \langle\langle m\rangle\rangle\Diamond Wins(m)[S_0]$ for both $m = X$ and $m = O$, where $\mathcal{D}_{TTT}$ is the game structure theory for tic-tac-toe specified above. However, if we start, for example, from the board resulting from executing $move(X, 2, 2)$ followed by $move(O, 1, 2)$, then X does have a strategy to ensure she wins, and indeed we have $\mathcal{D}_{TTT} \models \langle\langle X\rangle\rangle\Diamond Wins(X)[S]$ for $S = do(move(O, 1, 2), do(move(X, 2, 2), S_0))$. □

Verifying Properties of Game Structures

Next, we will show how one can verify that a formula of our logic is satisfied in a situation calculus-based game structure specified as explained in the previous section under certain assumptions, spelled out below. First, note that since we assume finite sets of action types and agents, we can rewrite the formula $\langle\langle G\rangle\rangle \bigcirc \varphi$ as follows:

$$
\begin{array}{l}
\langle\langle G\rangle\rangle \bigcirc \varphi \doteq \\
\quad (\bigvee_{agt \in G} Control(agt, now) \land \\
\qquad \bigvee_{a \in Agents} \exists\vec{x}.\, agent(a(\vec{x})) = agt \land \\
\qquad\quad Legal(do(a(\vec{x}), now)) \land \varphi(do(a(\vec{x}), now))) \lor \\
\quad (\bigvee_{agt \notin G} Control(agt, now) \land \\
\qquad \bigwedge_{a \in Agents} \forall\vec{x}.\, agent(a(\vec{x})) = agt \land \\
\qquad\quad Legal(do(a(\vec{x}), now)) \supset \varphi(do(a(\vec{x}), now)))
\end{array}
$$

Our verification method is based on two main ingredients: *(i) regression* (Pirri and Reiter 1999; Reiter 2001), and *(ii) fixpoint approximates* and the classical Knaster and Tarski results (Tarski 1955).

Regarding regression, we will assume that $Legal$ is regressible. Given this, if φ is regressible, then $\langle\langle G\rangle\rangle \bigcirc \varphi$ is also regressible, and in fact its regression is:

$$
\begin{array}{l}
\mathcal{R}(\langle\langle G\rangle\rangle \bigcirc \varphi) \doteq \\
\quad (\bigvee_{agt \in G} \mathcal{R}(Control(agt, now)) \land \\
\qquad \bigvee_{a \in A} \exists\vec{x}.\, agent(a(\vec{x})) = agt \land \\
\qquad\quad \mathcal{R}(Legal(do(a(\vec{x}), now))) \land \mathcal{R}(\varphi(do(a(\vec{x}), now)))) \lor \\
\quad (\bigvee_{agt \notin G} \mathcal{R}(Control(agt, now)) \land \\
\qquad \bigwedge_{a \in A} \forall\vec{x}.\, agent(a(\vec{x})) = agt \land \\
\qquad\quad \mathcal{R}(Legal(do(a(\vec{x}), now))) \supset \mathcal{R}(\varphi(do(a(\vec{x}), now))))
\end{array}
$$

This observation is the first key element of our method.

The second element is the ability, in some cases, to compute fixpoint approximates. Suppose that we want to verify a least fixpoint formula $\mu Z.\Psi(Z)$, where Z occurs free in Ψ. We can attempt to evaluate such a formula using the general technique of fixpoint approximates (Tarski 1955). The technique goes as follows. The approximates for a least fixpoint of the form $\mu Z.\Psi(Z)$ are as follows:

$$
\begin{array}{l}
Z_0 \doteq \Psi(False) \\
Z_1 \doteq \Psi(Z_0) \\
Z_2 \doteq \Psi(Z_1) \\
\cdots
\end{array}
$$

Observe that all of these formulas Z_i are situation suppressed which means that they all talk about the same situation, say now. As a direct consequence of the classical Knaster and Tarski results, we have:

[1] Although $\neg\langle\langle G\rangle\rangle \bigcirc \neg\Psi$ is not in $\mathcal{L}$ according to the syntax, the equivalent formula in negation normal form is.

Proposition 1 *Let $\mathcal{D}_{GS}$ be a situation calculus game structure and let S be a situation. If for some i, $\mathcal{D}_{GS} \models Z_{i+1}[S] \equiv Z_i[S]$, then $\mathcal{D}_{GS} \models Z_i[S] \equiv \mu Z.\Phi(Z)[S]$.*

Analogously, the approximates for a greatest fixpoint of the form $\nu Z.\Psi(Z)$ are as follows:

$$\begin{array}{l} Z_0 \doteq \Psi(True) \\ Z_1 \doteq \Psi(Z_0) \\ Z_2 \doteq \Psi(Z_1) \\ \ldots \end{array}$$

As a consequence of the Knaster and Tarski results, we have:

Proposition 2 *Let $\mathcal{D}_{GS}$ be a situation calculus game structure and let S be a situation. If for some i, $\mathcal{D}_{GS} \models Z_{i+1}[S] \equiv Z_i[S]$, then $\mathcal{D}_{GS} \models Z_i[S] \equiv \nu Z.\Psi(Z)[S]$.*

Based on this, we define a procedure $\tau(\cdot)$ that given an $\mathcal{L}$ formula tries to compute a first-order formula uniform in the current situation now that is equivalent to Ψ. We proceed by induction on structure of the $\mathcal{L}$ formula:

$$\begin{array}{l} \tau(\varphi) = \varphi \\ \tau(Z) = Z \\ \tau(\Psi_1 \wedge \Psi_2) = \tau(\Psi_1) \wedge \tau(\Psi_2) \\ \tau(\Psi_1 \vee \Psi_2) = \tau(\Psi_1) \vee \tau(\Psi_2) \\ \tau(\exists x.\Psi) = \exists x.\tau(\Psi) \\ \tau(\forall x.\Psi) = \forall x.\tau(\Psi) \\ \tau(\langle\langle G\rangle\rangle \bigcirc \Psi) = \mathcal{R}(\langle\langle G\rangle\rangle \bigcirc \tau(\Psi)) \\ \tau([[G]] \bigcirc \Psi) = \neg\mathcal{R}(\langle\langle G\rangle\rangle \bigcirc \tau(\mathrm{NNF}(\neg\Psi))) \\ \tau(\mu Z.\Psi) = \mathit{lfp}Z.\tau(\Psi) \\ \tau(\nu Z.\Psi) = \mathit{gfp}Z.\tau(\Psi) \end{array}$$

where NNF$(\neg\Psi)$ stands for the negation normal form of $\neg\Psi$ with the proviso that for variables NNF$(Z) \doteq Z$, and

- $\mathit{lfp}Z.\Psi$ is the formula R resulting from the least fixpoint procedure

```
$R := False$;
$R_{new} := \Psi(False)$;
while ($\mathcal{D}_{ca} \not\models R \equiv R_{new}$){
    $R := R_{new}$;
    $R_{new} := \Psi(R)$ }
```

- $\mathit{gfp}Z.\Psi$ is the formula R resulting from the greatest fixpoint procedure

```
$R := True$;
$R_{new} := \Psi(True)]$;
while ($\mathcal{D}_{ca} \not\models R \equiv R_{new}$){
    $R := R_{new}$;
    $R_{new} := \Psi(R)$ }
```

Notice that in computing such fixpoints we need to test whether $\mathcal{D}_{ca} \not\models R \equiv R_{new}$, i.e., check the validity of $R \equiv R_{new}$ under the unique name and domain closure assumptions for actions in $\mathcal{D}_{GS}$. Note that such a check is purely first-order (Reiter 1982).

For the least fixpoint formula $\langle\langle G\rangle\rangle\Diamond\varphi$, i.e. that coalition G can achieve φ, the fixpoint approximates are:

$$\begin{array}{l} Z_0 \doteq \varphi \vee \langle\langle G\rangle\rangle \bigcirc False \text{ i.e., } Z_0 = \varphi \\ Z_1 \doteq \varphi \vee \langle\langle G\rangle\rangle \bigcirc Z_0 \\ Z_2 \doteq \varphi \vee \langle\langle G\rangle\rangle \bigcirc Z_1 \\ \ldots \end{array}$$

By computing $\tau(\langle\langle G\rangle\rangle\Diamond\varphi)$, we apply regression at each step of the computation of the approximate so as to get a formula uniform in s:

$$\begin{array}{l} R_0 \doteq \varphi \\ R_1 \doteq \varphi \vee \mathcal{R}[\langle\langle G\rangle\rangle \bigcirc R_0] \\ R_2 \doteq \varphi \vee \mathcal{R}[\langle\langle G\rangle\rangle \bigcirc R_1] \\ \ldots \end{array}$$

Observe that by the regression theorem (Reiter 2001), given a situation S, $R_i[S]$ is equivalent to $Z_i[S]$, the difference between the two being that in $R_i[S]$ the only situation term that appears is S, while in $Z_i[S]$ we have S and perhaps other situation terms that may be up to i steps in the future.

Notice that $\tau(\langle\langle G\rangle\rangle\Diamond\varphi)$ stops if $\mathcal{D}_{ca} \models R_i \equiv R_{i+1}$. Obviously, there are no guarantees in general that such a condition is ever met. However, if it does eventually hold, i.e. $\mathcal{D}_{GS} \models Z_i[S] \equiv Z_{i+1}[S]$, then $\mathcal{D}_{GS} \models R_i[S] \equiv \mu Z.\Psi(Z)[S]$. Moreover, $R_i[S]$ is not only first-order but uniform in S, and this means that if $S = S_0$, then we have $\mathcal{D}_{GS} \models R_i[S_0]$ iff $\mathcal{D}_{S_0} \cup \mathcal{D}_{ca} \models R_i[S_0]$. That is, in order to check whether $\mathcal{D}_{GS} \models \mu Z.\Psi(Z)[S_0]$, we only need to check whether $\mathcal{D}_{S_0} \cup \mathcal{D}_{ca} \models R_i[S_0]$. In other words, we have reduced the task of verifying a fixpoint formula in the situation calculus (including the second-order axioms for situations) into that of verifying a first-order formula, by some iterated syntactic manipulation and checks of first order formulas (needed to compute R_i). We stress again here that there are obviously no guarantees that the procedure to compute R_i terminates in general. An analogous line of reasoning would allow us to verify that $\mathcal{D}_{GS} \models \nu P.\langle\langle G\rangle\rangle\Box\varphi[S_0]$ by checking $\mathcal{D}_{S_0} \cup \mathcal{D}_{ca} \models \tau(\nu P.\langle\langle G\rangle\rangle\Box\varphi)[S_0]$. More generally, we have the following theorem:

Theorem 1 *Let $\mathcal{D}_{GS}$ be a situation calculus game structure and let Ψ be an $\mathcal{L}$-formula. If the algorithm above terminates, then $\mathcal{D}_{GS} \models \Psi[S_0]$ iff $\mathcal{D}_{S_o} \cup \mathcal{D}_{ca} \models \tau(\Psi)[S_0]$.*

GameGolog

As an alternative, more procedural, way of specifying game structures, we introduce a variant of the ConGolog programming language that we call *Game Structure ConGolog*, or simply *GameGolog*. In this variant, all nondeterministic choices are made by some agent that has control in the situation, and *are recorded in the situation*. For example, consider the program $[agt\ a \mid b]$. The agent can choose to go left and continue with the execution of a, ending up in $do(a, do(left(agt), s))$, or she can choose to go right and continue with the execution of b, ending up in $do(b, do(right(agt), s))$. Thus the important thing about GameGolog programs is that which agent gets to act next is always specified, and that the history of nondeterministic choices is recorded in the situation.

Formally, GameGolog is obtained from ConGolog by replacing the three nondeterministic constructs and the concurrency construct with new versions where we specify explicitly which agent is responsible for the nondeterministic choice (we denote GameGolog programs by ρ, possibly with sub/superscripts):

$[agt\ \rho_1 \vert \rho_2]$	nondeterministic branch
$[agt\ \pi x.\rho]$	nondeterministic choice of argument
$[agt\ \rho^*]$	nondeterministic iteration
$[agt\ \rho_1 \Vert \rho_2]$	concurrency

Intuitively, $[agt\ \rho_1|\rho_2]$ states that agent agt chooses whether to continue with ρ_1 or with ρ_2; $[agt\ \pi x.\rho]$ states that agent agt chooses a binding for the variable x to continue with ρ; $[agt\ \rho^*]$ states that agent agt chooses when to stop the iteration of ρ; $[agt\ \rho_1\|\rho_2]$ states that agent agt chooses how to interleave the execution of ρ_1 and ρ_2.

The definitions of $Trans$ and $Final$ for the new nondeterministic constructs are the following (for the deterministic constructs they are as before):

$$\begin{array}{l}
Trans([agt\ \rho_1|\rho_2], s, \rho', s') \equiv \\
\quad s' = do(left(agt), s) \wedge \rho' = \rho_1 \vee \\
\quad s' = do(right(agt), s) \wedge \rho' = \rho_2 \\
Trans([agt\ \pi x.\rho], s, \rho', s') \equiv \\
\quad \exists x.s' = pick(agt, x) \wedge \rho' = \rho \\
Trans([agt\ \rho^*], s, \rho', s') \equiv \\
\quad s' = do(continue(agt), s) \wedge \rho' = \rho; [agt\ \rho^*] \vee \\
\quad s' = do(stop(agt), s) \wedge \rho' = True? \\
Trans([agt\ \rho_1\|\rho_2], s, \rho', s') \equiv \\
\quad s' = do(left(agt), s) \wedge \rho' = [agt\ \rho_1\langle\|\ \rho_2] \vee \\
\quad s' = do(right(agt), s) \wedge \rho' = [agt\ \rho_1\|\rangle\ \rho_2] \\
Trans([agt\ \rho_1\ \langle\|\ \rho_2], s, \rho', s') \equiv \\
\quad Trans(\rho_1, s, \rho'_1, s) \wedge \rho' = [agt\ \rho'_1\|\rho_2] \\
Trans([agt\ \rho_1\ \|\rangle\ \rho_2], s, \rho', s') \equiv \\
\quad Trans(\rho_2, s, \rho'_2, s) \wedge \rho' = [agt\ \rho_1\|\rho'_2]
\end{array}$$

$$\begin{array}{l}
Final([agt\ \rho_1|\rho_2], s) \equiv False \\
Final([agt\ \pi x.\rho], s) \equiv False \\
Final([agt\ \rho^*], s) \equiv False \\
Final([agt\ \rho_1\|\rho_2], s) \equiv False \\
Final([agt\ \rho_1\ \langle\|\ \rho_2], s) \equiv \\
\quad Final(\rho_1, s) \wedge Final(\rho_2, s) \\
Final([agt\ \rho_1\ \|\rangle\ \rho_2], s) \equiv \\
\quad Final(\rho_1, s) \wedge Final(\rho_2, s)
\end{array}$$

In the above, for the nondeterministic constructs, we introduce explicit choice actions as necessary to record how the nondeterministic choice was resolved into the situation. Thus, for a nondeterministic branch $[agt\ \rho_1\ |\ \rho_2]$, either agt chooses to go left, performing the $left(agt)$ choice action, and then executes the left branch ρ_1, or chooses to go right, doing the $right(agt)$ choice action, and then executes the right branch ρ_2. For a nondeterministic choice of argument $[agt\ \pi x.\rho]$, agt first picks a binding for x, doing the $pick(agt, x)$ choice action, and then executes the body ρ for this binding of x. For a nondeterministic iteration, the agent either chooses to continue iterating or to stop; in either case, the choice is recorded by doing a choice action, $continue(agt)$ in the former case, and $stop(agt)$ in the latter. Concurrency requires some more technical machinery: consider the program $[agt\ \rho_1\ \|\ \rho_2]$ and suppose that agt chooses to perform a step of ρ_1 first. Then, we must ensure that the first transition comes from ρ_1, after which agent agt can decide on the interleaving of the remainder of ρ_1, ρ'_1, and ρ_2, i.e. $[agt\ \rho'_1\ \|\ \rho_2]$. To handle this, we introduce the new "auxiliary" construct $[agt\rho_1\ \langle\|\ \rho_2]$ to model the state of the computation after the agent has chosen/committed to go left, but before it has performed any further transition. Similarly, we introduce $[agt\rho_1\ \|\rangle\ \rho_2]$ to represent the state after agt has chosen to go right.

Example 3 To model the tic-tac-toe game, we can use the following GameGolog program ρ_{TTT}:

$$\begin{array}{l}
\textbf{while}\ \neg Finished()\ \textbf{do}\ (\\
\quad [X\ \pi r, c.\ move(X, r, c)]; \\
\quad \textbf{if}\ \neg Finished()\ \textbf{then} \\
\qquad [O\ \pi r, c.\ move(O, r, c)] \\
\quad \textbf{else}\ True?\ )
\end{array}$$

ρ_{TTT} simply alternates the moves of X and O, starting from X, until a player has won or the board no longer has blanks and hence the fluent $Finished()$ holds. □

Interestingly, GameGolog programs that do not involve the new concurrency constructs, can be expressed directly in ConGolog. Indeed let us define a translation function ∂ by induction on the structure of GameGolog programs (without concurrency):

$$\begin{array}{l}
\partial(\alpha) = \alpha \\
\partial(\varphi?) = \varphi? \\
\partial(\rho_1; \rho_2) = \partial(\rho_1); \partial(\rho_2) \\
\partial(\textbf{if}\ \varphi\ \textbf{then}\ \rho_1\ \textbf{else}\ \rho_2) = \textbf{if}\ \varphi\ \textbf{then}\ \partial(\rho_1)\ \textbf{else}\ \partial(\rho_2) \\
\partial(\textbf{while}\ \varphi\ \textbf{do}\ \rho) = \textbf{while}\ \varphi\ \textbf{do}\ \partial(\rho) \\
\partial([agt\ \rho_1\ |\ \rho_2]) = (left(agt); \partial(\rho_1))\ |\ (right(agt); \partial(\rho_2)) \\
\partial([agt\ \pi x.\rho]) = \pi x.pick(agt, x); \partial(\rho) \\
\partial([agt\ \rho^*]) = (continue(agt); \partial(\rho))^*; stop(agt)
\end{array}$$

Then, one can show the following result by induction on GameGolog programs:

Theorem 2 *For every GameGolog program ρ not involving concurrency, we have that*

$$\begin{array}{l}
\mathcal{D}_{TF} \models Trans_{GG}(\rho, s, \rho', s') \equiv Trans_{CG}(\partial(\rho), s, \partial(\rho'), s'), \\
\mathcal{D}_{TF} \models Final_{GG}(\rho, s) \equiv Final_{CG}(\partial(\rho), s),
\end{array}$$

where we have distinguished the $Trans$ and $Final$ predicates of GameGolog and ConGolog by the subscripts GG and CG respectively, and denoted by D_{TF} the union of the axioms for $Trans$ and $Final$ for the two languages.

Strictly speaking, in GameGolog one must specify explicitly which agent controls each nondeterministic choice. But often we would like to say that a given agent controls all the nondeterministic choices in some program. Thus for convenience, we will often write $[agt\ \varrho]$, where ϱ is a program that may mix constructs of both GameGolog and ConGolog, to refer to the GameGolog program $\iota([agt\ \varrho])$, where agt controls all the nondeterministic choices in ϱ that are not already controlled by other agents. To get the standard GameGolog program from $[agt\ \varrho]$, we inductively define the syntactic transformation ι as follows:

$$\begin{array}{l}
\iota([agt\ \rho]) = \rho \text{ where } \rho \text{ is a (pure) GameGolog program} \\
\iota([agt\ \varrho_1; \varrho_2]) = \iota([agt\ \varrho_1]); \iota([agt\ \varrho_2]) \\
\iota([agt\ \textbf{if}\ \varphi\ \textbf{then}\ \varrho_1\ \textbf{else}\ \varrho_2]) = \\
\qquad \textbf{if}\ \varphi\ \textbf{then}\ \iota([agt\ \varrho_1])\ \textbf{else}\ \iota([agt\ \varrho_2]) \\
\iota([agt\ \textbf{while}\ \varphi\ \textbf{do}\ \varrho]) = \textbf{while}\ \varphi\ \textbf{do}\ \iota([agt\ \varrho]) \\
\iota([agt_1\ [agt_2\ \varrho]]) = \iota([agt_2\ \varrho]) \\
\iota([agt\ \varrho_1|\varrho_2]) = [agt\ \iota([agt\ \varrho_1])|\iota([agt\ \varrho_2])] \\
\iota([agt\ \pi x.\varrho]) = [agt\ \pi x.\iota([agt\ \varrho])] \\
\iota([agt\ \varrho^*]) = [agt\ \iota([agt\varrho])^*] \\
\iota([agt\ \varrho_1\|\varrho_2]) = [agt\ \iota([agt\ \varrho_1])\ \|\ \iota([agt\ \varrho_2])] \\
\iota([agt\ \varrho_1\langle\|\varrho_2]) = [agt\ \iota([agt\ \varrho_1])\langle\|\ \iota([agt\ \varrho_2])] \\
\iota([agt\ \varrho_1\|\rangle\ \varrho_2]) = [agt\ \iota([agt\ \varrho_1])\ \|\rangle\ \iota([agt\ \varrho_2])]
\end{array}$$

This is used in the larger examples later in the paper.

Let's now examine some of the semantic properties of GameGolog. We denote by $\mathcal{D}_{GG}$ the union of the axioms for $Trans$ and $Final$ for GameGolog. In GameGolog (as well as in the version of ConGolog presented in the Preliminaries), we have that any program transition adds one action to the situation (as there are no test transitions):

$$\Sigma \cup \mathcal{D}_{GG} \models Trans(\rho, s, \rho', s') \supset \exists a. s' = do(a, s).$$

Thus the number of transitions in $Trans^*(\rho, s, \rho', s')$ is equal to the number of actions that lead from s to s'. GameGolog has another interesting property: that all decisions are represented in the situation. This means that if we make a transition to arrive to a given situation, the remaining program is unique for this situation. Formally, we can prove, by induction on the GameGolog structure of ρ, that

$$\begin{aligned} &\Sigma \cup \mathcal{D}_{GG} \models \\ &\quad Trans(\rho, s, \rho', s') \wedge Trans(\rho, s, \rho'', s') \supset \rho' = \rho'' \end{aligned}$$

Furthermore, by induction on the number of *Trans* steps and observing that such a number is determined by the final situation, it can be shown that this uniqueness property is preserved over sequences of transitions:

$$\begin{aligned} &\Sigma \cup \mathcal{D}_{GG} \models \\ &\quad Trans^*(\rho, s, \rho', s') \wedge Trans^*(\rho, s, \rho'', s') \supset \rho' = \rho'' \end{aligned}$$

Now that we have defined this GameGolog language, where all decisions are recorded in the situation, we can use it to specify $Legal$ in a game structure. Note that this use of a program is quite different from the standard one: instead of executing it, we use the space of possible computations specified by the program to define $Legal$ in the game structure, i.e., to define the possible moves/legal states of the game.

Concretely, we do this by axiomatizing *Legal* to specify that the legal situations are exactly those that can be reached from the initial situation by performing transitions on the program ρ_0 modeling the game structure of interest:

$$Legal(s) \equiv \exists \rho'. Trans^*(\rho_0, S_0, \rho', s).$$

Note that $Legal$ defined in this way is more than simply an invariant for the program, it is the *strongest invariant* (Cousot 1990), i.e. it completely characterizes the reachable configurations of the program.

In some cases, is it also useful to specify which situations correspond to the *Final* configuration of the program. We do this as follows:

$$Final(s) \equiv \exists \rho'. Trans^*(\rho_0, S_0, \rho', s) \wedge Final(\rho', s).$$

This definition is well-founded since ρ' is functionally determined by ρ_0 and s. We call theories of this form, where $Legal$ (and possibly $Final(s)$) is specified by a GameGolog program, GameGolog *theories*, and denote one by $\mathcal{D}_{GGT}$.

We want to be able to verify whether a game structure specified by GameGolog program satisfies some properties of interest. To develop an effective verification method, it is helpful to combine the specification of the game structure (*Legal*) in terms of a GameGolog program with the specification of the property that we want to verify on this game structure. We do this by introducing a program-constrained version of the $\langle\langle G \rangle\rangle \bigcirc \varphi$ operator, $\ll G \gg \bigcirc \varphi$, as follows:

$$\begin{aligned} &\quad \ll G \gg \bigcirc \varphi \doteq \\ &(\exists agt \in G. \\ &\quad \exists a, \rho'.\ agent(a) = agt \wedge \\ &\qquad Trans(\rho_{now}, now, \rho', do(a, now)) \wedge \varphi[\rho', do(a, now)]) \vee \\ &(\exists agt \notin G. \\ &\quad \exists a, \rho'(agent(a) = agt \wedge \\ &\qquad Trans(\rho_{now}, now, \rho', do(a, now))) \wedge \\ &\quad \forall a, \rho'.\ agent(a) = agt \wedge \\ &\qquad Trans(\rho_{now}, now, \rho', do(a, now)) \supset \varphi[\rho', do(a, now)]). \end{aligned}$$

Note that this operator depends on two suppressed parameters, the current situation now, and the current program ρ_{now}. For any $\mathcal{L}$-formula Ψ, we have a related formula $\widehat{\Psi}$ with two suppressed arguments as above, which we will use in the following. We call the resulting logic $\mathcal{L}_p$.

It is easy to verify that for any GameGolog theory $\mathcal{D}_{GGT}$ and any situation calculus formula φ uniform in s:

$$\begin{aligned} \mathcal{D}_{GGT} \models &\langle\langle G \rangle\rangle \bigcirc \varphi[s] \equiv \\ &\quad \exists \rho. Trans^*(\rho_0, S_0, \rho, s) \wedge \ll G \gg \bigcirc \varphi[\rho, s] \end{aligned}$$

More generally we have that:

Theorem 3 *For every GameGolog theory $\mathcal{D}_{GGT}$ and associated program ρ_0, and $\mathcal{L}$-formula Ψ, the corresponding $\mathcal{L}_p$ formula $\widehat{\Psi}$ is such that*

$$\mathcal{D}_{GGT} \models \forall \rho, s. Trans^*(\rho_0, S_0, \rho, s) \supset (\Psi[s] \equiv \widehat{\Psi}[\rho, s]).$$

Note that given ρ_0, S_0 and s functionally determine ρ.

Example 4 Returning to our tic-tac-toe example, the existence of a strategy to win the game for player m, can be now formalized as:

$$\mu Z.\, Wins(m) \vee \ll m \gg \bigcirc Z$$

i.e., $\ll m \gg \Diamond Wins(m)$. Again, if we start for example, from the board resulting from executing $move(X, 2, 2)$ followed by $move(O, 1, 2)$, then X does have a strategy that ensures a win, and indeed, we have that $\mathcal{D}_{TTTP} \models \ll X \gg \Diamond Wins(X)[\rho, do(move(O, 1, 2), do(move(X, 2, 2), S_0))]$, where $\mathcal{D}_{TTTP}$ is the GameGolog theory for tic-tac-toe defined above (now including the program ρ_{TTT}), and ρ is what remains of the original program ρ_{TTT} in situation $do(move(O, 1, 2), do(move(X, 2, 2), S_0))$ (in this case incidentally, $\rho = \rho_{TTT}$). □

Verifying Properties of GameGolog Theories

We propose a technique to verify properties over game structures specified using GameGolog programs. The technique is based again on *(i) regression* and *(ii) fixpoint approximates*, but now with the addition of *(iii)* a variant of *characteristic graphs* (Claßen and Lakemeyer 2008), which are used to compactly represent all the possible configurations that a GameGolog program may visit during its execution. Given a GameGolog program ρ_0, its *characteristic graph*, is a graph where the nodes V are tuples of the form $\langle \rho, \chi \rangle$, meaning that ρ is a possible remaining program during ρ_0's execution and that χ characterizes the conditions under which ρ may terminate (i.e., be $Final$). The initial node is $v_0 = \langle \rho_0, \chi_0 \rangle$. Edges in $\mathcal{G}$ stand for single transitions between program configurations and are labeled with tuples of the form $\langle \pi\vec{x} : \alpha, \omega\rangle$, where α is an action term with a specified action type (i.e., not an action variable) and variables $\vec{x}$ may appear free in α and ω. Roughly speaking, an edge represents the fact that the program in the source node may perform a transition to a configuration with the program in the destination node when one chooses instantiations for $\vec{x}$ and performs action α in a situation where ω holds.

Let's see how we can adapt the characteristic graph-based verification methods proposed in (Claßen and Lakemeyer 2008) to check properties over game structures specified by GameGolog programs. First, note that in every node $v = (\rho, \chi)$ of a characteristic graph $\mathcal{G}$ of a GameGolog program, all outgoing edges will be labeled by actions of the same agent, the agent that controls the node (this follows by induction on the structure of the program ρ). We denote this agent by $agent(v)$. Then, we can specify a verification procedure that labels nodes differently depending on whether or not they are controlled by an agent in the coalition.

We assume without loss of generality that the free variables occurring in formulas to be checked are distinct from those occurring in the program ρ_0, quantified by the π construct. We will develop a procedure $\llbracket\Psi\rrbracket$ that labels nodes in a characteristic graph $\mathcal{G}$ (the characteristic graph of the GameGolog program specifying the game structure of interest) for any $\mathcal{L}_p$-formula Ψ. If this procedure terminates (it may not), it produces a labeling of the nodes in the graph, i.e. a set $\{\langle v, \varphi\rangle | v \in \mathcal{G}\}$ where each φ is a first order formula, and this labeling can be used to check whether the property of interest Ψ holds. We denote such a labeling by $\mathcal{Z}$. We begin by introducing the following definitions.

$\llbracket\varphi\rrbracket \doteq \{\langle v, \varphi\rangle | v \in \mathcal{G}\}$ where φ is any first-order, possibly open, formula.

$\mathcal{Z}_1$ AND $\mathcal{Z}_2 \doteq \{\langle v, \phi_1 \wedge \phi_2\rangle | \langle v, \phi_1\rangle \in \mathcal{Z}_1, \langle v, \phi_2\rangle \in \mathcal{Z}_2\}$.

$\mathcal{Z}_1$ OR $\mathcal{Z}_2 \doteq \{\langle v, \phi_1 \vee \phi_2\rangle | \langle v, \phi_1\rangle \in \mathcal{Z}_1, \langle v, \phi_2\rangle \in \mathcal{Z}_2\}$.

EXISTS $x.\mathcal{Z} \doteq \{\langle v, \exists x.\phi\rangle | \langle v, \phi\rangle \in \mathcal{Z}\}$.

ALL $x.\mathcal{Z} \doteq \{\langle v, \forall x.\phi\rangle | \langle v, \phi\rangle \in \mathcal{Z}\}$.

$Pre(G, \mathcal{Z}) \doteq \{\langle v, \phi\rangle | v \in \mathcal{G},$ where
if $agent(v) \in G$ then
$\phi = \bigvee_{v \xrightarrow{\pi\vec{x}:\alpha,\omega} v' \in \mathcal{G}, (v',\phi') \in \mathcal{Z}} \exists\vec{x}.\omega(\vec{x}) \wedge \mathcal{R}(\phi'(do(\alpha, now)))$
and if $agent(v) \notin G$ then
$\phi = \bigvee_{v \xrightarrow{\pi\vec{x}:\alpha,\omega} v' \in \mathcal{G}, (v',\phi') \in \mathcal{Z}} \exists\vec{x}.\omega(\vec{x}) \wedge$
$\bigwedge_{v \xrightarrow{\pi\vec{x}:\alpha,\omega} v' \in \mathcal{G}, (v',\phi') \in \mathcal{Z}} \forall\vec{x}.\omega(\vec{x}) \supset \mathcal{R}(\phi'(do(\alpha, now)))\}$.

$\overline{Pre}(G, \mathcal{Z}) \doteq \{\langle v, \mathrm{NNF}(\neg\phi)\rangle | \langle v, \phi\rangle \in Pre(G, \mathcal{Z})\}$.

LFP$\mathcal{Z}.\Psi(\mathcal{Z})$, where $\Psi(\mathcal{Z})$ denotes an parametrized expression in which $\mathcal{Z}$ occurs as a parameter (possibly together with other parameters), stands for the result of the following procedure (in which $\mathcal{Z} \neq \mathcal{Z}_{new}$ is an abbreviation for $\mathcal{D}_{ca} \not\models \bigwedge_{\langle v,\varphi\rangle \in \mathcal{Z}, \langle v,\varphi_{new}\rangle \in \mathcal{Z}_{old}} \varphi \equiv \varphi_{new}$):

```
$\mathcal{Z} := \llbracket False\rrbracket;$
$\mathcal{Z}_{new} := \Psi(\mathcal{Z});$
while $(\mathcal{Z} \neq \mathcal{Z}_{new})\{$
    $\mathcal{Z} := \mathcal{Z}_{new};$
    $\mathcal{Z}_{new} := \Psi(\mathcal{Z})\ \}.$
```

GFP$\mathcal{Z}.\Psi(\mathcal{Z})$, where $\Psi(\mathcal{Z})$ denotes a parametrized expression in which $\mathcal{Z}$ occurs as a parameter, stands for the result of the following procedure:

```
$\mathcal{Z} := \llbracket True\rrbracket;$
$\mathcal{Z}_{new} := \Psi(\mathcal{Z});$
while $(\mathcal{Z} \neq \mathcal{Z}_{new})\{$
    $\mathcal{Z} := \mathcal{Z}_{new};$
    $\mathcal{Z}_{new} := \Psi(\mathcal{Z})\ \}.$
```

We can show that $Pre(G, \llbracket\varphi\rrbracket)$ characterizes the condition for $\langle\!\langle G\rangle\!\rangle \bigcirc \varphi$ to be satisfied by a program in the characteristic graph in a situation:

Theorem 4 *For all situation terms s and nodes $v = \langle\rho, \chi\rangle \in \mathcal{G}$, we have that $\mathcal{D}_{GGT} \models \langle\!\langle G\rangle\!\rangle \bigcirc \varphi[\rho, s] \equiv \psi[s]$, where $\langle v, \psi\rangle \in Pre(G, \llbracket\varphi\rrbracket)$.*

To verify that $\langle\!\langle G\rangle\!\rangle \bigcirc \varphi$ holds for the initial game situation, we must determine whether $\mathcal{D}_{GGT} \models \langle\!\langle G\rangle\!\rangle \bigcirc \varphi[\rho_0, S_0]$. Given the above theorem, we can do this by checking whether $\mathcal{D}_{S_0} \cup \mathcal{D}_{ca} \models \psi[S_0]$ where $\langle v, \psi\rangle \in Pre(G, \llbracket\varphi\rrbracket)$.

Given these definitions, we can define our general labeling procedure $\llbracket\Psi\rrbracket$ for any $\mathcal{L}_p$-formula Ψ as follows:[2]

$\llbracket\varphi\rrbracket$, where φ is first-order is as defined earlier
$\llbracket Z\rrbracket \doteq \mathcal{Z}$ where $\mathcal{Z}$ is any labeling
$\llbracket\Psi_1 \wedge \Psi_2\rrbracket \doteq \llbracket\Psi_1\rrbracket$ AND $\llbracket\Psi_2\rrbracket$
$\llbracket\Psi_1 \vee \Psi_2\rrbracket \doteq \llbracket\Psi_1\rrbracket$ OR $\llbracket\Psi_2\rrbracket$
$\llbracket\exists x.\Psi\rrbracket \doteq$EXISTS $x.\llbracket\Psi\rrbracket$
$\llbracket\forall x.\Psi\rrbracket \doteq$ALL $x.\llbracket\Psi\rrbracket$
$\llbracket\langle\!\langle G\rangle\!\rangle \bigcirc \Psi\rrbracket \doteq Pre(G, \llbracket\Psi\rrbracket)$
$\llbracket [[G]] \bigcirc \Psi\rrbracket \doteq \overline{Pre}(G, \llbracket\Psi\rrbracket)$
$\llbracket\mu Z.\Psi(Z)\rrbracket \doteq$ LFP$\mathcal{Z}.\llbracket\Psi(Z)\rrbracket$
$\llbracket\nu Z.\Psi(Z)\rrbracket \doteq$ GFP$\mathcal{Z}.\llbracket\Psi(Z)\rrbracket$

We can show that when $\llbracket\Psi\rrbracket$ terminates, we can use the uniform formulas that label the nodes in the resulting labeling to check whether Ψ holds:

Theorem 5 *For every $\mathcal{L}_p$-formula Ψ, if $\llbracket\Psi\rrbracket$ terminates and $\langle v, \varphi\rangle$, with $v = \langle\rho, \chi\rangle$, is in the returned set, then for all situation terms s, $\mathcal{D}_{GGT} \models \Psi[\rho, s] \equiv \varphi[s]$.*

For checking whether a property holds in the initial situation, we can strengthen the above into the following:

Theorem 6 *For every $\mathcal{L}_p$-formula Ψ, if $\llbracket\Psi\rrbracket$ terminates and $\langle v_0, \varphi\rangle$, with $v = \langle\rho_0, \chi_0\rangle$, is in the returned set, then $\mathcal{D}_{GGT} \models \Psi[\rho_0, S_0]$ iff $\mathcal{D}_{S_0} \cup \mathcal{D}_{ca} \models \varphi[S_0]$.*

Example: Contingent Planning

In (Lespérance, De Giacomo, and Ozgovde 2008), a model of contingent planning for use with agent programming languages is developed. A contingent planning problem is specified in terms of a program (in ConGolog or some other agent programming language) specifying the agent's task δ_{Agt} and a program specifying the possible behaviors of the environment δ_{Env} (possibly involving various other agents), together with a background basic action theory $\mathcal{D}$.

We can can use our framework to formalize this account of contingent planning quite simply (assuming complete information). We can specify the game structure as follows:[3]

$$\rho_{cp} = [Env\ \delta_{Env} \parallel [Agt\ \delta_{Agt}; finish(Agt)]]$$

[2] Such a procedure can be immediately extended to deal with the ability of expressing $Final$ of a program in a situation directly in $\mathcal{L}_p$, following (Claßen and Lakemeyer 2008).

[3] Note that here we don't use prioritized concurrency, as in (Lespérance, De Giacomo, and Ozgovde 2008), to make the environment run at higher priority than the agent.

where δ_{Env} is the program specifying the environment and δ_{Agt} is the program specifying the agent's task. Note that the environment chooses when the agent gets to execute. Also, we append to the agent program a new action $finish(Agt)$, which we use to record its successful completion. $finish(Agt)$ is always possible and makes the new fluent $Finished(Agt)$ true. $Finished(Agt)$ is initially false and is unaffected by other actions. Solving the contingent planning task amounts showing that:

$$\mathcal{D}_{GGT} \models \ll Agt \gg \Diamond Finished(Agt)[\rho_{cp}, S_0]$$

i.e., the agent can ensure that eventually it has completed its task no matter how the environment behaves.

Let's look at some simple concrete instantiations of Agt and Env. Suppose that we have an agent Agt whose task δ_{Agt} is specified by the following program:

$$\textbf{while } \neg Accepted \textbf{ do } (request|sleep).$$

Agt might be a child that repeatedly sleeps or requests candy from his mother until she accepts to give it to him (alternatively, Agt might be a client process that requests a resource from a server). Suppose that the environment's behavior is specified as follows:

$$\delta_{Env} \doteq \textbf{while } True \textbf{ do } (Requested?; (accept|reject)).$$

Here, Env might be the child's mother (or alternatively a server) that simply responds to requests from Agt by either accepting or rejecting the request. We assume that $Requested$ is initially false and is made true by the action $request$ and made false by both $accept$ and $reject$. We also assume that $Accepted$ is initially false and is made true by $accept$ and made false by $request$. We assume that all actions are always physically possible.

Given this specification, it is easy to show that Agt cannot ensure that she will successfully complete her task and get her request accepted:

$$\begin{aligned}\mathcal{D}_{GGT} \models\ & \neg \ll Agt \gg \Diamond Finished(Agt)[\rho_{cp}, S_0] \\ & \wedge \neg \ll Agt \gg \Diamond Accepted[\rho_{cp}, S_0].\end{aligned}$$

The reason for this is that Env is allowed to be unfair and keep rejecting all requests forever. We can express the form of strong fairness (SF) that would rule this out as follows:[4]

$$SF \doteq \Box \Diamond Requested \supset \Box \Diamond Accepted.$$

Note that for the above specification, strong fairness does not hold along all execution paths:

$$\mathcal{D}_{GGT} \models \neg \ll \gg SF[\rho_{cp}, S_0].$$

Now, suppose that we change the environment program so that after a request is rejected, a subsequent request is always accepted:

```
$\delta_{Env} \doteq$ while $True$ do (
        $Requested?$;
        if $LastReqRejected$ then $accept$
        else($accept|reject$) ).
```

Here, $LastReqRejected$ is initially false and is made true by $reject$ and made false by $accept$. Then we can show that strong fairness does hold along all paths and that Agt can ensure that he will successfully complete his task and get his request accepted:

$$\begin{aligned}\mathcal{D}_{GGT} \models\ & \ll \gg SF[\rho_{cp}, S_0] \wedge \\ & \ll Agt \gg \Diamond Finished(Agt)[\rho_{cp}, S_0] \wedge \\ & \ll Agt \gg \Diamond Accepted[\rho_{cp}, S_0].\end{aligned}$$

Now, suppose that we change the environment program again so that after a request is rejected, a subsequent request is always accepted provided that Env is not insensitive:

```
$\delta_{Env} \doteq$
  while $True$ do (
    $Requested?$;
    if $LastReqRejected \wedge \neg Insensitive$ then $accept$
    else($accept|reject$) ).
```

Suppose also that it is not known whether $Insensitive$ holds initially, and that it is unaffected by actions. Then, we can show that strong fairness holds along all paths if and only if $\neg Insensitive$ holds initially, and that Agt can ensure that he will complete his task and get his request accepted if and only if strong fairness holds along all paths:

$$\begin{aligned}\mathcal{D}_{GGT} \models\ & \neg Insensitive(S_0) \equiv \ll \gg SF[\rho_{cp}, S_0] \equiv \\ & \ll Agt \gg \Diamond Finished(Agt)[\rho_{cp}, S_0] \equiv \\ & \ll Agt \gg \Diamond Accepted[\rho_{cp}, S_0].\end{aligned}$$

Example: Process Orchestration

In (De Giacomo and Sardina 2007), an account of service/process orchestration is proposed and techniques for synthesizing an orchestrator are developed. In this account, the desired service, as well as the available services that may be orchestrated to try to produce the desired service, are modeled as transition systems. It is assumed that the desired service is "deterministic" in the sense that the current state and the action performed by the desired service functionally determine the next state of the desired service. The available services need not be deterministic in this sense, and the orchestrator cannot control what state they end up in after a transition involving a given action. An orchestrator is essentially a function that given a system state and an action selected by the desired service selects an available service to perform that action. Roughly speaking, an orchestrator *realizes* a desired service if for any run of the desired service, the orchestrator can select at every step an available service and have it perform the action selected by the desired service. This must hold regardless of the nondeterministic state transition performed by the selected available service at each step (the other available services are assumed to remain in their current state). The available services must also be in a final state whenever the desired service is in a final state (where it may terminate).

To illustrate how this works, let's look at a simple example involving a service for searching for songs and listening to them, taken from (Berardi et al. 2005). We model the desired service by the program $\delta_0 = ((searchByAuthor|searchByTitle); listen)^*$, i.e. we want to be able to repeatedly search for a song by either author or title, and then listen to it. The

[4] We use LTL notation for simplicity, but the formula can be expressed using fixpoints in a standard way, see e.g., (de Alfaro, Henzinger, and Majumdar 2001; Piterman, Pnueli, and Sa'ar 2006).

two available services are modeled by the programs $\delta_1 = (\mathit{searchByAuthor}; \mathit{listen})^*$, which allows repeatedly searching for a song by author only, and then listening to it, and $\delta_2 = (\mathit{searchByTitle}; \mathit{listen})^*$, which allows repeatedly searching for a song by title only, and then listening to it. Note that here we model services as ConGolog programs. Clearly, in this example, we can in fact orchestrate the available services to realize the desired service by selecting δ_1 to perform $\mathit{searchByAuthor}$ and then listen, when the desired service selects these actions, and by selecting δ_2 to perform $\mathit{searchByTitle}$ and then listen, when the desired service selects them. If we do this, the available services are final whenever the desired service is final.

We can use GameGolog to formalize the account of process orchestration discussed above as follows. We can specify the system/game structure using the following GameGolog program:

$$\begin{array}{l}\rho_o = [\mathit{Sched}\ [\mathit{DesS}\ \delta_0'; \mathit{finish}(\mathit{DesS})] \parallel \\ \qquad [\mathit{AvailS}_1\ \delta_1; \mathit{finish}(\mathit{AvailS}_1)] \parallel \ldots \parallel \\ \qquad [\mathit{AvailS}_n\ \delta_n; \mathit{finish}(\mathit{AvailS}_n)]].\end{array}$$

Here, δ_0' is a modified version of the program specifying the desired service δ_0 (we explain the modification below) and $\delta_1, \ldots, \delta_n$ are the programs specifying the available services. We introduce an agent DesS that controls the choices made in executing the desired service δ_0', and agents $\mathit{AvailS}_1, \ldots, \mathit{AvailS}_n$, where each AvailS_i controls the choices in executing the available service δ_i. In the model of the system ρ_o, all of these services run concurrently, with the interleaving under the control of a scheduler agent Sched. Note that the environment here is deterministic and can be modeled by the basic action theory. As earlier, we introduce a $\mathit{finish}(\mathit{agt})$ action for each agent with an associated $\mathit{Finished}(\mathit{agt})$ fluent to record when the agent terminates.

The idea behind our formalization is that the scheduler agent Sched can choose an interleaving of the available services to perform the actions "requested" by the desired service. To model the "requesting" of actions by the desired service, each action a in the program δ_0 specifying the desired service is replaced by the new action $\mathit{request}(a)$, yielding the program δ_0'. We also introduce a functional fluent $\mathit{requested}$ to keep track of the last requested action:

$$\begin{array}{l}\mathit{requested}(do(a,s)) = x \equiv \\ \quad \mathit{requested}(s) = \mathit{nil} \land a = \mathit{request}(a_r) \land x = a_r \lor \\ \quad \mathit{requested}(s) = a \land x = \mathit{nil} \lor \\ \quad \neg(\mathit{ControlAction}(a) \lor \exists \mathit{agt}(a = \mathit{finish}(\mathit{agt}))) \land \\ \qquad \mathit{requested}(s) \neq a \land x = \mathit{error} \lor \\ \quad \mathit{requested}(s) = x \land \\ \qquad \neg(\mathit{requested}(s) = \mathit{nil} \land a = \mathit{request}(a_r) \land x = a_r) \land \\ \qquad \neg(\mathit{requested}(s) = a \land x = \mathit{nil} \land \\ \qquad \neg(\neg(\mathit{ControlAction}(a) \lor \exists \mathit{agt}(a = \mathit{finish}(\mathit{agt}))) \land \\ \qquad\quad \mathit{requested}(s) \neq a \land x = \mathit{error}).\end{array}$$

A requested action is cleared only when it is performed for real, by one of the available services. Moreover, if a desired service performs a non-control non-finish action that has not been requested, $\mathit{requested}$ records the error by permanently taking on the value error. The desired service can only request an action after its last request has been cleared/served:

$$\mathit{Poss}(\mathit{request}(a), s) \equiv \mathit{requested} = \mathit{nil}.$$

Initially, no requests have yet been made, i.e. $\mathit{requested}(S_0) = \mathit{nil}$.

Given this model, solving the orchestration task amounts showing that: $\mathcal{D}_{GGT} \models \Psi[\mathit{rho}_{\mathit{orch}}, S_0]$ where

$$\begin{array}{rcl}\Psi & \doteq & \ll \mathit{Sched} \gg \Box(\mathit{requested} \neq \mathit{error} \land \\ & & \quad \Diamond(\mathit{requested} \neq \mathit{nil} \lor \mathit{AllFinished}) \land \\ & & \quad (\mathit{requested} \neq \mathit{nil} \supset \Diamond \mathit{requested} = \mathit{nil}))\end{array}$$

expressible using fixpoints as

$$\begin{array}{rcl}\Psi & \doteq & \nu Z.\, \Psi_1 \land \Psi_2 \land \Psi_3 \land \ll\mathit{Sched}\gg \bigcirc Z \\ \Psi_1 & \doteq & \mathit{requested} \neq \mathit{error} \\ \Psi_2 & \doteq & \mu Z.\, \mathit{requested} \neq \mathit{nil} \lor \mathit{AllFinished} \\ & & \qquad \lor \ll\mathit{Sched}\gg \bigcirc Z \\ \Psi_3 & \doteq & \mathit{requested} \neq \mathit{nil} \supset \\ & & \qquad \mu Z.\, \mathit{requested} = \mathit{nil} \lor \ll\mathit{Sched}\gg \bigcirc Z\end{array}$$

where $\mathit{AllFinished} \doteq \mathit{Finished}(\mathit{DesS}) \land \mathit{Finished}(\mathit{AvailS}_1) \land \ldots \land \mathit{Finished}(\mathit{AvailS}_n)$. That is, we must show that the scheduler agent can ensure that in any run of the system, no error occurs (the performance of an unrequested non-control non-finish action), the desired service keeps requesting actions until it and all available services finish, and any requested action by the desired service is eventually done by one of the available services.

Discussion

In this paper, we have devised techniques for the verification of properties over game structures and GameGolog programs. Interestingly, in presence of complete information on the initial situation, the techniques presented here can be extended to actually synthesize strategies that allow the agents in the coalition to ensure that the property hold. To do so, we can rely on the notion of *witness* used in model checking (Clarke, Grumberg, and Peled 1999; Piterman, Pnueli, and Sa'ar 2006). Such witnesses can be obtained by storing additionally the labeling corresponding to the various intermediate approximates used in the computation of fixpoints. Examples of such constructions in the situation calculus and characteristic graphs can be found in (Sardina and De Giacomo 2009). Interestingly in the case of complete information, the logical tasks required by Theorem 6, from *logical implication* become *evaluation* of first-order formulas over a (possibly infinite) interpretation. Furthermore, if the fluents are propositional, then we can use results such as (Fritz, Baier, and McIlraith 2008) to reduce our situation calculus-based game structures to finite game structures, and use ATL model checkers such as MCMAS (Lomuscio, Qu, and Raimondi 2009), to actually get a sound and complete effective tool for both verification and synthesis. Notice that in the case of propositional fluents the computation of the approximates always converges.

In the presence of incomplete information about the initial situation, getting synthesis techniques out of verification techniques is much more involved. Indeed, strategies not only need to exists (as verification in our logic guarantees), but need to be *epistemically feasible*, i.e., the agent performing a strategy needs to have enough information to actually make all the choices needed for its execution; see e.g., (Sardina et al. 2004). As well, it would obviously be of interest to generalize our framework to the full incomplete information case, where agents have different knowledge and can acquire new information as they act (Sardina et al. 2004; Ghaderi, Levesque, and Lespérance 2007).

Note that our framework supports incomplete specifications of the application domain (our basic action theories need not have a single model), but that all agents are assumed to have complete knowledge of the theory, actions are observable by all agents, and there are no sensing actions that allow an agent to gain additional private knowledge. We hope to relax these restrictions in future work.

In related work on game-theoretic logics, we should mention ATEL (van der Hoek and Wooldridge 2003), a variant of ATL that deals with incomplete knowledge and ensures epistemic feasibility of strategies. However, this logic is propositional and there is no expressive language for specifying the game structure. (Walther, van der Hoek, and Wooldridge 2007) develops a variant of ATL where strategies can be referred to explicitly; but it denotes strategies by primitive terms and they cannot be specified explicitly in the object language. Our approach is also quite different from that in (Finzi and Lukasiewicz 2004; 2006), which develop "game theoretic" agent programming languages based on Golog; their languages are based on classical game theory and aim to support restricted cases of game theoretic reasoning by an agent at execution time. We focus more on verification and offline synthesis and do not use probabilities and utilities. There has has also been work on logical frameworks to represent games. The Game Description Language (GDL) (Genesereth, Love, and Pell 2005) is a declarative specification language to represent discrete complete information games that has been used as a standard in "general game playing" competitions, where agents must compete in games that are not known in advance. Our basic formalization of game structures in the situation calculus is similar to GDL's. But GDL does not deal with the representation and verification of properties over game structures, nor does it consider specifying game structures procedurally. Schulte and Delgrande (Schulte and Delgrande 2004) have proposed a situation calculus-based formalization of classical von Neumann-Morgenstern game theory, including optimal strategies and Nash equilibria. Our approach is quite different, and focuses on multiagent interaction problems that are naturally specified using a procedural language and on their verification.

We conclude by emphasizing that the study here is essentially theoretical. While effectiveness guarantees will only be available for very specific cases (e.g. finite states, or structures that allow for quantifier elimination such as Presburger arithmetic), it remains very important to complement our work (as well as that of (Claßen and Lakemeyer 2008; Sardina and De Giacomo 2009)) with experimental studies, to understand whether these techniques, based on labeling of characteristic graphs, are effective in practical cases.

References

Alur, R.; Henzinger, T. A.; and Kupferman, O. 2002. Alternating-time temporal logic. *J. ACM* 49(5):672–713.

Berardi, D.; Calvanese, D.; Giacomo, G. D.; Lenzerini, M.; and Mecella, M. 2005. Automatic service composition based on behavioral descriptions. *Int. J. Cooperative Inf. Syst.* 14(4):333–376.

Bradfield, J., and Stirling, C. 2007. Modal mu-calculi. In *Handbook of Modal Logic*, volume 3. Elsevier. 721–756.

Clarke, E. M.; Grumberg, O.; and Peled, D. A. 1999. *Model checking*. Cambridge, MA, USA: The MIT Press.

Claßen, J., and Lakemeyer, G. 2008. A logic for non-terminating Golog programs. In *Proc. of KR'08*, 589–599.

Cousot, P. 1990. Methods and logics for proving programs. In *Handbook of Theor. Comp. Science, Volume B*. Elsevier. 841–994.

de Alfaro, L.; Henzinger, T. A.; and Majumdar, R. 2001. From verification to control: Dynamic programs for omega-regular objectives. In *Proc. of LICS'01*, 279–290.

De Giacomo, G., and Sardina, S. 2007. Automatic synthesis of new behaviors from a library of available behaviors. In *Proc. of IJCAI'07*, 1866–1871.

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

Finzi, A., and Lukasiewicz, T. 2004. Game-theoretic agent programming in golog. In *Proc. of ECAI'04*, 23–27.

Finzi, A., and Lukasiewicz, T. 2006. Adaptive multi-agent programming in gtgolog. In *Proc. of ECAI'06*, 753–754.

Fritz, C.; Baier, J. A.; and McIlraith, S. A. 2008. ConGolog, Sin Trans: Compiling ConGolog into basic action theories for planning and beyond. In *Proc. of KR'08*, 600–610.

Genesereth, M. R.; Love, N.; and Pell, B. 2005. General game playing: Overview of the AAAI competition. *AI Magazine* 26(2):62–72.

Ghaderi, H.; Levesque, H. J.; and Lespérance, Y. 2007. A logical theory of coordination and joint ability. In *Proc. of AAAI'07*, 421–426.

Lespérance, Y.; De Giacomo, G.; and Ozgovde, A. N. 2008. A model of contingent planning for agent programming languages. In *Proc. of AAMAS'08*, 477–484.

Lomuscio, A.; Qu, H.; and Raimondi, F. 2009. MCMAS: A model checker for the verification of multi-agent systems. In *Proc. of CAV'09*, 682–688.

Park, D. 1976. Finiteness is mu-ineffable. *Theor. Comput. Sci.* 3(2):173–181.

Pirri, F., and Reiter, R. 1999. Some contributions to the metatheory of the situation calculus. *J. ACM* 46(3):261–325.

Piterman, N.; Pnueli, A.; and Sa'ar, Y. 2006. Synthesis of reactive(1) designs. In *Proc. of VMCAI'06*, 364–380.

Reiter, R. 1982. Towards a logical reconstruction of relational database theory. In *On Conceptual Modeling*, 191–233.

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

Ruan, J.; van der Hoek, W.; and Wooldridge, M. 2009. Verification of games in the game description language. *J. Logic and Comput.*

Sardina, S., and De Giacomo, G. 2009. Composition of congolog programs. In *Proc. of IJCAI'09*, 904–910.

Sardina, S.; De Giacomo, G.; Lespérance, Y.; and Levesque, H. J. 2004. On the semantics of deliberation in IndiGolog – From theory to implementation. *Ann. Math. Artif. Intell.* 41(2–4):259–299.

Schulte, O., and Delgrande, J. P. 2004. Representing von neumann-morgenstern games in the situation calculus. *Ann. Math. Artif. Intell.* 42(1-3):73–101.

Tarski, A. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. of Mathematics* 5(2):285309.

van der Hoek, W., and Wooldridge, M. 2003. Cooperation, knowledge, and time: Alternating-time temporal epistemic logic and its applications. *Studia Logica* 75(1):125–157.

Walther, D.; van der Hoek, W.; and Wooldridge, M. 2007. Alternating-time temporal logic with explicit strategies. In *Proc. of TARK'07*, 269–278.