An Algorithmic Approach to Specifying and Verifying Subgame Perfect Equilibria

Frank Guerin
Department of Computing Science, King's College,
University of Aberdeen, Aberdeen AB24 3UE, Scotland.
f.guerin@abdn.ac.uk

ABSTRACT

Game theory is a popular tool for designing interaction protocols for agent systems. It is currently not clear how to apply this to open agent systems. By "open" we mean that foreign agents will be free enter and leave different agent systems at will. This means that agents will need to be able to work with previously unseen protocols. There does not yet exist any agreement on a standard way in which such protocols can be specified and published. Furthermore, it is not clear how an agent could be given the ability to use an arbitrary published protocol; the agent would need to be able to work out a strategy for participation. To address this we propose a machine readable language in which a game theory mechanism can be written in the form of a program. Enabling agents to automatically determine the game theoretic properties of an arbitrary game is difficult. Rather than requiring agents to find the equilibrium of the game, we propose that a recommended equilibrium strategy profile will be published along with the mechanism; agents can then check the recommendation to decide if it is indeed an equilibrium. We present an algorithm for this decision problem. It is hoped that this work could eventually contribute to interoperability in open agent systems.

1. INTRODUCTION

Game theory is a popular tool for designing interaction protocols for agent systems. It allows a sort of social engineering for agents; we can design the rules of a system so that the agents will have an incentive to reach the outcome we desire. All of this rests on the assumption that the rules of the system and their game theoretic properties are common knowledge amongst the agents. This is fine if we are designing a closed system, where we (the agent designer) know the rules, and we program the agents with knowledge of the game they will be participating in. It is currently not clear how to extend this work to open agent systems. By "open" we mean that foreign agents will be free enter and leave different agent systems at will. This means that agents will need to be able to work with previously unseen protocols. There does not yet exist any agreement on a standard way in which such protocols can be specified and published. Furthermore, it is not clear how an agent could be given the ability to use an arbitrary published protocol; the agent would need to be able to understand the rules and to work out a strategy for participation. Thus our area of work is in providing the theoretical infrastructure needed to enable game theory mechanisms to be employed in open multi-agent systems. This amounts to *specifying* and *verifying* the mechanisms in a way appropriate for agents. In this paper we will restrict our attention to pure strategy subgame perfect Nash equilibria in games of complete information.

Firstly, for *specification*, we will need a machine readable language in which a game theory mechanism can be written in the form of an agent interaction protocol, and published for agents to inspect. This would allow the workings of the protocol to be made public so that (i) the behaviour of agents enacting the protocol can be tested to determine if they are complying with the published rules; (ii) agents can inspect the published specification to determine its properties and hence their best strategy. For the machine readable language, we propose an algorithmic representation, as it is likely to be the most succinct way to represent most types of games. A protocol written in this language constitutes a specification of behaviour for agents, and they can be tested for compliance as in (i); however, this kind of compliance testing is relatively straightforward and has been discussed elsewhere [4, 7, 15, 16]. Our focus will be on point (ii).

To determine the best strategy for participation, an agent should "solve" the game to find the equilibrium strategies. Computing Nash equilibria is an open problem [12] and can be difficult [5, 17]. The solution proposed here is to include, with the specification of the protocol, the designer's recommended strategy; the recommendation should of course be an equilibrium of the game. Because trust is an issue in open systems, agents will need to verify for themselves that the published recommendation is indeed an equilibrium. This brings us to the idea of *verification* for mechanisms.

Given a published game and strategy profile, an agent will need a procedure to determine if it is in his interest to follow the recommended strategy. This is a much simpler problem than finding an equilibrium from scratch. Essentially, it is the difference between (i) checking every possible combination of values for a set of parameters, and (ii) checking every possible value of one parameter while all others are held constant, and repeating this process for each parameter. The first is exponential in the number of parameters, while the second is polynomial. The number of parameters here corresponds to the number of players in the game, and the possible values of each parameter correspond to the strategies agents can play.

The main contribution of the paper is an algorithm which can take, as input, an algorithmic representation of (i) a game and (ii) a strategy for each agent, and decide if the strategies are a subgame perfect Nash equilibrium of the game. It is hoped that this work could eventually contribute to interoperability in open agent systems, allowing agents to participate in foreign institutions and to understand the rules there.

Section 2 describes the syntax and program semantics of SMPL (Simple Mechanism Programming Language), a machine readable language for publishing both games and strategies. In section 3.1 we build on this program semantics, and define the game represented by an SMPL program. Section 4 shows how subgame perfect Nash equilibrium can be defined and presents the algorithm

for deciding if a published strategy profile is an equilibrium of a published game. Section 5 looks at related work, and Section 6 concludes with a discussion of the limitations of the work and directions for future work.

2. SMPL: SIMPLE MECHANISM PROGRAMMING LANGUAGE

This section presents the syntax and semantics for SMPL (Simple Mechanism Programming Language). It is copied from the Simple Programming Language (SPL) of Manna and Pnueli [9], with some modifications. We do not directly give a semantics to SMPL in terms of games; instead we first describe a standard program semantics for SMPL, in terms of the sequences of states it could produce, then in the next section (Section 3) we use the program semantics to describe the game represented by the SMPL program. SPL was chosen as the basis for our approach for two reasons: firstly it is a theoretical language, and hence extremely simple (as opposed to using Java, for example); secondly, it was originally designed for specifying reactive systems, hence it is well adapted to modular specification, where it is important to keep track of the communications passed between each module, and the variables local to each particular module. Space constraints force the presentation here to be terse; a reader not familiar with this style of semantics may wish to consult the original. To help the reader to get a feel for the language, Figure 1 gives a very simple example of an SMPL program describing the classic prisoners' dilemma involving two agents and the principal, as in the following normal form representation:

	1 (Cooperate)	**2 (Defect)**
1 (Cooperate)	$-1, -1$	$-6, 0$
2 (Defect)	$0, -6$	$-3, -3$

The messages exchanged are simple integers. Agent 1 chooses an integer, 1 or 2, and sends it to the principal. The principal reads input channel $\alpha_{P,1}$, storing the result in the variable *plyr1*. He then informs agent 2 that it is his turn, reading agent 2's action into *plyr2*. Finally the utilities are calculated and sent to the special channel $\alpha_{U,P}$.

2.1 SMPL Syntax

A program has the following syntax:

$$P :: \left[\begin{array}{l} Ag_1 :: \big[\text{declaration}; [\ell_1\text{:}\ S_1; \hat{\ell}_1\text{:}]\big] \parallel \ \ldots \\ \parallel Ag_P :: \big[\text{declaration}; [\ell_P\text{:}\ S_P; \hat{\ell}_P\text{:}]\big] \end{array}\right]$$

It consists of a number of *modules* representing each of the agent processes. Each Ag_i is an identifier for an agent in the game and each S_i is a statement which may itself be composed of other statements. Label ℓ_i is the location of part of the *program control variable* just before execution of the statement S_i, and $\hat{\ell}_i$ is its location just after. It is required that the module of exactly one agent will begin with a **choose** statement. The final agent (Ag_P above) has a special status and is known as the *principal*; no **choose** statements may appear in his module.

A declaration is a sequence of *declaration statements* with the following syntax:

$<$ **own in** | **own out** | **local** $>$ variable, ..., variable: type **where** φ_i

Statements in the program may only refer to variables declared in the declaration. Initial values for variables may be specified by the optional assertion **where** φ_i. Keyword **local** is for variables used by this program, not accessible to any other agent. Keywords **own**

$$Ag_1 :: \left[\begin{array}{lll} \textbf{local} & a & : \textbf{integer where } a = 0 \\ \textbf{own out} & \alpha_{P,1} & : \textbf{channel } [1..] \textbf{ of integer} \\ \big[\ \ell_0\text{: } \textbf{choose } a\ 1..2; \ell_1\text{: } \alpha_{1,P} \Leftarrow a\ : \ell_2\big] & & \end{array}\right]$$

$$\Big\| Ag_2 :: \left[\begin{array}{lll} \textbf{local} & b & : \textbf{integer where } b = 0 \\ \textbf{own in} & \alpha_{2,P} & : \textbf{channel } [1..] \textbf{ of integer} \\ \textbf{own out} & \alpha_{P,2} & : \textbf{channel } [1..] \textbf{ of integer} \\ \left[\begin{array}{l} m_0 : \textbf{await} |\alpha_{2,P}| > 0; \\ m_1 : \textbf{choose } b\ 1..2; m_2 : \alpha_{P,2} \Leftarrow b\ : m_3 \end{array}\right] & & \end{array}\right]$$

$$\Big\| Ag_P :: \left[\begin{array}{ll} \textbf{local} & \textit{plyr1}, \textit{plyr2}\text{: } \textbf{integer where } \textit{plyr1} = \textit{plyr2} = 0 \\ \textbf{own in} & \alpha_{P,1}, \alpha_{P,2} : \textbf{channel } [1..] \textbf{ of integer} \\ \textbf{own out} & \alpha_{U,P}, \alpha_{2,P}\text{: } \textbf{channel } [1..] \textbf{ of integer} \\ \left[\begin{array}{ll} p_0: & \textbf{await} |\alpha_{P,1}| > 0; p_1 : \alpha_{P,1} \Rightarrow \textit{plyr1} \\ p_2: & \alpha_{2,P} \Leftarrow 0; \\ p_3: & \textbf{await} |\alpha_{P,2}| > 0; p_4 : \alpha_{P,2} \Rightarrow \textit{plyr2} \\ p_5: & \textbf{if } \textit{plyr1} = 1 \wedge \textit{plyr2} = 1 \\ & \textbf{then } p_6 : \alpha_{U,P} \Leftarrow \langle -1, -1 \rangle \\ p_7: & \textbf{if } \textit{plyr1} = 1 \wedge \textit{plyr2} = 2 \\ & \textbf{then } p_8 : \alpha_{U,P} \Leftarrow \langle -6, 0 \rangle \\ p_9: & \textbf{if } \textit{plyr1} = 2 \wedge \textit{plyr2} = 1 \\ & \textbf{then } p_{10} : \alpha_{U,P} \Leftarrow \langle 0, -6 \rangle \\ p_{11}: & \textbf{if } \textit{plyr1} = 2 \wedge \textit{plyr2} = 2 \\ & \textbf{then } p_{12} : \alpha_{U,P} \Leftarrow \langle -3, -3 \rangle : p_{13} \end{array}\right] & \end{array}\right]$$

Figure 1: A simple game of two agents playing prisoners' dilemma.

in and **own out** are for asynchronous communication channels for input and output respectively. A channel is a variable whose value is a list of integers. We identify channel variables as follows: $\alpha_{i,j}$ will be an input channel for agent i and an output channel for agent j; i.e. only agent i can read from this channel, and only agent j can write to it. Each non-principal agent i must have an output channel $\alpha_{P,i}$ to send messages to the principal; agent i may also have an input channel $\alpha_{i,P}$ on which to receive messages from the principal. Finally, the principal must have a special channel $\alpha_{U,P}$ which he can write to once at the end of the game, to determine the utility received by all agents. There are no other channels. We allow no communication channels between other individual agents; everything must go through the principal. This is because we need to have a global history of the game to uniquely identify a game state, and that will come from all of the messages received by the principal.

Basic Statement	***Description***
$u := e$	*assignment*: assign value e to variable u
choose $u\ c_1..c_2$	*choose* a value in the interval for variable u
await c	*wait* for Boolean expression c
$\alpha \Leftarrow e$	*send* expression e on channel α
$\alpha \Rightarrow u$	*receive* on channel α and store in variable u
if c **then** S_1 **else** S_2	*conditional* statement
if c **then** S_1	*one branch conditional* statement
$S_1; \ldots; S_k$	*concatenation*: sequential execution
while c **do** S	*repetition* of s

Statements may be basic or compound. A compound statement is enclosed in parentheses $[\ldots]$ when it is a sub statement of a larger statement except when the compound statement has a line to itself. Sub statements within concatenation statements are separated by semicolons which we omit if there is a line break.

2.2 SMPL Program Semantics

The semantics are defined via a transition system. The transition system has variables corresponding to the program's variables, and it has transitions which describe how those variables change as program statements are executed. The program will identify a transition system, and the transition system will define the possible sequences of states it could produce. Thus the semantics of a program is given in terms of possible sequences of states (of variables) it could produce.

A program identifies a unique transition system $\langle V, \Theta, \mathcal{T} \rangle$. The variables come from a universal set of typed variables $\mathcal{V}$, called the vocabulary. From this we can construct expressions (such as $x + 3y + 4$), atomic formulae (such as $(x + 3y) > 7$) and *Assertions* (such as $x > y \wedge y < 4$). A *state* s is an interpretation of $\mathcal{V}$, assigning each variable $u \in \mathcal{V}$ a value $s[u]$ over its domain. $V \subseteq \mathcal{V}$ is the set of system variables: one of these is the control variable π which represents the location of the next statement to be executed, the remainder represent program variables. π is an $(n+1)$-tuple, where n is the number of agents in the program ($+1$ for the principal); π has one part of its tuple to point to the location within each agent's module. The initial condition Θ is a conjunction of all initial values for variables (appearing in **where** clauses), an empty value for all channels ($\alpha = \lambda$) and the control variable equal to the set of entry locations for each agent. If a state s of the system satisfies the assertion Θ, then it is a state from which the system can start running. $\mathcal{T}$ is a set of transitions including one transition corresponding to each statement in the program, as follows. Note that primed values refer to the value in the successor state, and the $\bullet$ symbol is used to add an element to one end of a list; for example $\alpha' = \alpha \bullet e$ means that the value of α in the successor state will be equal to what it was previously, but with e appended to the end.

SMPL Statement	***Transition Relation***
$u := e$	$m(\ell, \hat{\ell}) \wedge u' = e \wedge p(Y - \{u\})$
choose $u\ c_1..c_2$	$m(\ell, \hat{\ell}) \wedge p(Y - \{u\}) \wedge \bigvee_{c=c_1}^{c_2} u' = c$
await c	$m(\ell, \hat{\ell}) \wedge c \wedge p(Y)$
$\alpha \Leftarrow e$	$m(\ell, \hat{\ell}) \wedge \alpha' = \alpha \bullet e \wedge p(Y - \{\alpha\})$
$\alpha \Rightarrow u$	$m(\ell, \hat{\ell}) \wedge \lvert\alpha\rvert > 0 \wedge$ $\alpha = u' \bullet \alpha' \wedge p(Y - \{u, \alpha\})$
if c **then** ℓ_1: S_1	$[m(\ell, \ell_1) \wedge c \wedge p(Y)] \vee$ $[m(\ell, \hat{\ell}) \wedge \neg c \wedge p(Y)]$
if c **then** ℓ_1: S_1 **else** ℓ_2: S_2	$[m(\ell, \ell_1) \wedge c \wedge p(Y)] \vee$ $[m(\ell, \ell_2) \wedge \neg c \wedge p(Y)]$
while c **do** $[\ell_1$: $S\ \ell$:]	$[m(\ell, \ell_1) \wedge c \wedge p(Y)] \vee$ $[m(\ell, \hat{\ell}) \wedge \neg c \wedge p(Y)]$

This assumes that ℓ is the statement's label and $\hat{\ell}$ its post-label. The abbreviation $m(\ell, \hat{\ell})$ means a move of control from location ℓ to location $\hat{\ell}$; i.e. the part of π which now points to ℓ will subsequently point to $\hat{\ell}$, and all other parts of π remain the same. The abbreviation $p(U)$ means that all variables in the set U are not changed by this transition. Y is the set of non control variables, so $V = \{\pi\} \cup Y$. Each transition maps each state onto a set of possible successor states. If a transition τ maps a state s to a non-empty set of possible successor states then τ is enabled on s, if it maps s to the null set then the transition is disabled on state s. The transitions in the system tell us how one state can move to the next. A transition is *taken* at state s if the next state is related to s by the transition.

A sequence of states (possibly infinite) $s^0, s^1, s^2, s^3, \ldots$ is called a *computation of the program* P (which identifies our transition system) if s^0 satisfies the initial condition Θ and if each state s^{j+1} is accessible from the previous state s^j via one of the transitions $\mathcal{T}$ in the system. If it is a finite computation then there will be a final state s^n which has no successor state. A computation is a sequence of states that could be produced by an execution of the program. For example, playing the equilibrium path for the program of Figure 1 would produce the following sequence of states:

$\{\langle \ell_0, m_0, p_0 \rangle, \lambda, \lambda, \lambda, \lambda, 0, 0, 0, 0\} \rightarrow$
$\{\langle \ell_1, m_0, p_0 \rangle, \lambda, \lambda, \lambda, \lambda, 2, 0, 0, 0\} \rightarrow$
$\{\langle \ell_2, m_0, p_0 \rangle, 2, \lambda, \lambda, \lambda, 2, 0, 0, 0\} \rightarrow$
$\{\langle \ell_2, m_0, p_1 \rangle, 2, \lambda, \lambda, \lambda, 2, 0, 0, 0\} \rightarrow$
$\{\langle \ell_2, m_0, p_2 \rangle, 2, \lambda, \lambda, \lambda, 2, 0, 2, 0\} \rightarrow$
$\{\langle \ell_2, m_0, p_3 \rangle, 2, 0, \lambda, \lambda, 2, 0, 2, 0\} \rightarrow$
$\{\langle \ell_2, m_1, p_3 \rangle, 2, 0, \lambda, \lambda, 2, 0, 2, 0\} \rightarrow$
$\{\langle \ell_2, m_2, p_3 \rangle, 2, 0, \lambda, \lambda, 2, 2, 2, 0\} \rightarrow$
$\{\langle \ell_2, m_3, p_3 \rangle, 2, 0, 2, \lambda, 2, 2, 2, 0\} \rightarrow$
$\{\langle \ell_2, m_3, p_4 \rangle, 2, 0, 2, \lambda, 2, 2, 2, 0\} \rightarrow$
$\{\langle \ell_2, m_3, p_5 \rangle, 2, 0, 2, \lambda, 2, 2, 2, 2\} \rightarrow \ldots \rightarrow$
$\{\langle \ell_2, m_3, p_{13} \rangle, 2, 0, 2, \langle -3, -3 \rangle, 2, 2, 2, 2\}$

Where each state gives the values of variables in this order:

$\{\pi, \alpha_{P,1}, \alpha_{2,P}, \alpha_{P,2}, \alpha_{U,P}, a, b, plyr1, plyr2\}$.

Control is initially at the start of each agent's module and channels are empty (λ).

Given a fixed decision for each agent's choice points, an SMPL program should produce a single computation; otherwise it is not a valid SMPL program. This means that at any state, all the agents, except one, should be at an **await** statement, or should have terminated. This restriction ensures that we have a unique history of the system corresponding to a single state of the game. Furthermore, a valid SMPL program must have no infinite computations; this ensures that all games represented by SMPL programs are finite. The program should also have a unique start state.

3. REPRESENTING GAMES AND STRATEGIES

This section makes use of the SMPL semantics to define the Game represented by an SMPL program. Essentially this involves stepping through the states of the SMPL program (as defined by the SMPL program semantics) until a **choose** statement is encountered, at which point a game node is created, and then the process of stepping through the program states continues.

3.1 The Game Represented by a Program

Since each node in a game tree has a unique history of actions taken to reach it, the tree can be represented by the set of histories. We will use the terms node and history interchangeably. Each action corresponds to a message sent to the principal in our case. If we can extract the possible message histories from the SMPL program, then we can use them to represent the game. We can also use the messages sent and received by non-principal agents to determine the history apparent to them, and hence the game nodes which they cannot distinguish (information sets).

A history tuple H in a system of n agents (+principal) is a tuple $\langle h_0, h_1, \ldots, h_n \rangle$ where h_0 is the global history: an ordered list of the messages received by the principal, and each other h_i is the communication history apparent to agent i; i.e. an ordered list of the messages sent and received by i. We will refer to individual elements h_i as $H[i]$.

Let $takeTransition$ describe an interpreter function; it takes in an SMPL program state and an SMPL program and it returns the state reached after taking a single transition. This can be used to perform any transition in the system except for the transition cor-

algorithm: $runGame$ runs the SMPL program
inputs: $prog$: program
outputs:s: terminal state, h_0: terminal node,
Inf: information set data
$\langle s, m, A, H\rangle := initialState(prog)$
$Inf := \{\langle m, H[m], H[0], A\rangle\}$
repeat
select $a \in A$ # nondeterministic selection
$\langle s, m, A, H\rangle := takeAction(s, m, a, H, prog)$
$Inf := Inf \cup \{\langle m, H[m], H[0], A\rangle\}$
until $terminal(s, prog) = True$
return $\langle s, H[0], Inf\rangle$

Figure 3: Algorithm: runGame

responding to a **choose** statement, because the transition system has no way to know what action an agent has taken. The algorithm $takeAction$ (Figure 2) handles the agent's choice, updating the system variable appropriately; it then steps through the transitions using $takeTransition$, until it meets a **choose** transition, at which point it stops and returns the current system state, the history, and the identity of the agent who needs to choose. The algorithm additionally updates histories every time a communication statement is about to be executed. If it is something read by the principal then it is added to P's history $H[0]$, and also the sender's. If it is something read by another agent i then it is only added to i's history $H[i]$.

Note that because we are writing an algorithm to interpret another program, we have two sets of variables. To avoid confusion, variables describing the state of the program $prog$ will be prefixed by "s." for example $s.u$ means the variable u within the program $prog$. Variables without prefixes are part of our algorithm, and not $prog$.

The $terminal$ function returns $True$ if the state passed as the first parameter is a terminal state of the program passed as the second parameter; it returns $False$ otherwise. A terminal state is one where no location in the control variable is pointing to a **choose** statement, and no transition is enabled; i.e. the program is not waiting for any agent to make a choice.

Figure 3 shows a tiny algorithm that runs the game to find the global histories and apparent histories produced by all possible computations. The only reason for not putting this together with the previous algorithm is that we want to use $takeAction$ alone later. This algorithm includes nondeterministic choice, so it can give several possible answers.

The $initialState$ function will take a game program as a parameter and will return a 4-tuple: s, the initial state of the game program; m, the agent who has the move; A, the set of actions to choose from; H, the initial history tuple $H = \langle[], [], \ldots, []\rangle$. The $utility(s)$ function returns the utility of a terminal state s; this is a simple matter of looking at the value of the channel $\alpha_{U,P}$, within s.

DEFINITION 3.1. *An SMPL program $prog$ represents the following finite extensive form game $G = \langle T, U, I, M\rangle$:*

- The game tree T is the set $\{h_0 \mid \langle m, h_m, h_0, A\rangle \in Inf$ where $\langle \_, \_, Inf\rangle = runGame(prog)\}$; i.e. we take all the possible outputs from running the algorithm $runGame$ on $prog$, this means taking all possible choices for the nondeterministic selection of an action; then from the 3-tuples returned, we look at the final Inf and we collect all the global histories h_0 that are in there. These are all the possible nodes of the game. A node h_s is a successor node of a node h_p iff h_p is a prefix of h_s and its length is one less. The set of all successors of a node is given by the function S. Nodes having an empty set of successors are termed *terminal*.

- The utility function U gives the utility of a terminal node; it is obtained by first creating a set of tuples which relate a terminal node to its corresponding terminal program state. $\Phi = \{\langle h_0, s\rangle \mid \langle s, h_0, Inf\rangle = runGame(prog)\}$.
$U(h_0) \triangleq utility(s)$ where $\langle h_0, s\rangle \in \Phi$.

- The information set function I maps each nonterminal node to the set of nodes in the same information set as it; it is obtained by unifying all possible Inf sets.
$$\mathcal{I} = \{\langle m, h_m, h_0, A\rangle \mid \langle m, h_m, h_0, A\rangle \in Inf \text{ and } \langle \_, \_, Inf\rangle = runGame(prog)\}.$$
$$I(h_0) \triangleq \{n \mid \langle m, h_m, n, A\rangle \in \mathcal{I} \text{ and } \langle m, h_m, h_0, A\rangle \in \mathcal{I}\}.$$

- The mover function M maps each game node to the agent who has the move at that node; it is given by $M(h_0) \triangleq m$ such that $\langle m, \_, h_0, \_\rangle \in \mathcal{I}$.

Note that an SMPL program can only represent a game of perfect recall. This is because the history apparent to an agent i is used specify agent i's information sets. It is assumed that the apparent history determines what game nodes agent i can distinguish between. Hence it is impossible to specify a game which would place two nodes, with the same apparent history, in different information sets. Neither is it possible to have the principal artificially send extra messages to agent i to make the history different; the idea of publishing the game is that the entire SMPL program is available for inspection by any participant, hence agent i would be aware that the extra messages it received did not affect the game state. Apart from this restriction, SMPL can be used to represent any finite imperfect information game (this follows from the Turing-completeness of SMPL).

3.2 The Strategy Represented by a Program

For a given game, a strategy σ_i is a function mapping each information set owned by agent i to an action. A restricted version of the SMPL syntax can be used to write a *strategy program*:

$$strat_i :: \left[\begin{array}{l} \textbf{in } history : \textbf{array } [1..] \textbf{ of integer}; \\ \textbf{out } action : \textbf{integer}; \\ declaration;\ S \end{array}\right]$$

The program must have exactly one input (for the history) and one output (for the action taken). As an example, the following is the equilibrium strategy for agent 1 in the game of Figure 1.

$$strat_1 :: \left[\begin{array}{l} \textbf{in } history : \textbf{array } [1..] \textbf{ of integer}; \\ \textbf{out } action : \textbf{integer}; \\ action := 2 \end{array}\right]$$

It is rather simple because there is only one contingency in which the agent is called on to act. Now let run denote a function that can interpret the program passed as the first parameter, and run it on the input passed as the second parameter, returning the program's output.

DEFINITION 3.2. *Let the SMPL program $prog$ represent a game, and let $\mathcal{I}$ be the set of tuples $\langle m, h_m, h_0, A\rangle$ describing its mover, apparent history, global history and action set as in Definition 3.1. Let $strat_i$ be some strategy program such that for all inputs h where*

algorithm: $takeAction$ returns the program state after an agent makes a choice
inputs: s: state, m: mover, a: action, H: history tuple, $prog$: program
outputs: s: resultant state, n: mover, A: actions available, H: history tuple

ℓ_0:= $s.\pi[m]$, where ℓ_0 points to a statement
of the form ℓ_0: **choose** u $c_1..c_2$ ℓ_1:
and u is a state variable of s

let $s.u := a$ # Set value of state variable u to action a.
let $s.\pi[m] := \ell_1$: # mth component of $s.\pi$ to point to ℓ_1:
repeat # At this point exactly one transition is enabled
  if The enabled transition relation contains # (we never have more than one enabled,
    an atomic formula of the form # one is always enabled after a **choose**)
    $\alpha_{P,i} = u' \bullet \alpha'_{P,i}$ # Note: all these variables belong to s.
    where i is some agent and P is the principal.
  then $H[0] := H[0] \bullet s.u'$ # Update global and apparent histories
    $H[i] := H[i] \bullet s.u'$ **fi** # with action received by Principal
  if The enabled transition relation contains
    an atomic formula of the form
    $\alpha_{i,P} = u' \bullet \alpha'_{i,P}$ # Update apparent history
  then $H[i] := H[i] \bullet s.u'$ **fi** # with action received by i
  $s := takeTransition(s, prog)$
until s has no successor state
if $terminal(s, prog)$ **then** $A := \{\}$
else n:= the positive integer such that # Now one of the locations in $s.\pi$
    $s.\pi[n]$ points to a statement # must point to a **choose** statement.
    of the form **choose** v $d_1..d_2$
  $A := \{a \mid a$ is an integer in the range $d_1..d_2\}$ **fi**
return $\langle s, n, A, H\rangle$

Figure 2: Algorithm: takeAction

$\langle i, h, n, A\rangle \in \mathcal{I}$: $strat_i$ terminates[1] *and $run(strat_i, h) \in A$. Then we say that $strat_i$, describes a strategy σ_i for agent i in the game represented by $prog$, where for each node n in the game tree such that $\langle i, h, n, A\rangle \in \mathcal{I}$:*
$\sigma_i(I(n)) = run(strat_i, h)$.

A profile of strategies is represented by $Strat = \langle strat_1, strat_2, \ldots\ strat_c\rangle$, where c is the number of agents; it is a tuple of strategy programs, one for each agent. We will refer to the individual elements as $Strat[i]$.

4. VERIFYING SUBGAME PERFECT NASH EQUILIBRIA

The following is the standard definition of a SPNE [2]. It is first necessary to define an NE. Strategy profile σ is a NE if, for all players i,

$u_i(\sigma_i, \sigma_{-i}) \geq u_i(s_i, \sigma_{-i})$ for all $s_i \in S_i$

$u_i(\ldots)$	the utility obtained by agent i if the strategies ... are played in the game.
σ_i	the strategy in σ which is to be played by agent i.
σ_{-i}	the strategies in σ which are to be played by all other players (not player i).
S_i	the set of possible strategies for agent i.

Strategy profile σ is a SPNE if the restriction of σ to G is a NE of G for every subgame G. A subgame is any branch of the game tree which starts at a singleton information set and does not cut any information set.

We now look at an alternative definition in our framework, and then prove its equivalence. We extend the domain of the utility function U to all nodes of the game tree. For a nonterminal node n, $U(n) = U(n \bullet \sigma(I(n)))$; i.e. n's utility is the same as the utility of the successor node reached by taking the action recommended by the strategy profile. An agent i's utility at a node n is $U(n)[i]$; if all agents follow σ, then this is the utility this agent can expect to get once node n is reached. A simple inductive proof on the length of histories shows that this is true. Take the longest histories as the base case. Now the inductive step: if U correctly describes the expected utility for all nodes of length l, then it must also be correct for nodes of length l-1. This is clear because all of the successors of an l-1 node, being of length l, have a correct U, and the U at an l-1 node is simply copied from its successor node by σ, which is what is expected to be played.

We define the function $maxU(n)$ as follows: if n is terminal then $maxU(n) = U(n)$; if n is nonterminal then:

$$\text{for } i = M(n) : maxU(n)[i] = \max_{m \in S(n)} \big(maxU(m)[i]\big)$$

$$\text{for } i \neq M(n) : maxU(n)[i] = maxU(n \bullet \sigma(I(n)))[i]$$

Again, the same type of simple inductive proof can show that $maxU(n)[i]$ is the maximum utility that agent i can expect to get, once node n is reached, if all other agents follow the strategies σ and agent i takes optimal actions at each node. Information sets do not matter here because the agent knows the strategies of other agents, so he knows exactly which node he will be at in any information set. It is possible that the value of $maxU(\ldots)[i]$ at two nodes, in an information set belonging to agent i, may each be relying on different actions being taken in that set; this is not a problem because $maxU(\ldots)[i]$ for the closest common ancestor will take on the value of only the greatest of these two, and so it is a value that really is achievable for agent i without requiring the agent to act differently at two nodes in an information set.

[1]This of course makes the language of strategy programs undecidable, but there should be no problem in determining that cases of practical interest are in the language.

Requirement 1 *(for strategy profile σ and a game G) For all agents i, and for all game nodes n which are the root of some proper subgame of G (i.e. n is in a singleton information set, and the subgame which starts there does not cut any of G's information sets), $maxU(n)[i] \leq U(n)[i]$.*

CLAIM 4.1. *A strategy profile σ is a subgame perfect Nash equilibrium (SPNE) for a game G iff Requirement 1 holds.*

Proof The "if" part (by contradiction): Assume Requirement 1 holds, but σ is not a SPNE for G. Since σ is not a SPNE, there must exist a proper subgame G_s of G and a strategy s_i, for some agent i, such that for the restriction of s_i and σ to G_s, $u_i(s_i, \sigma_{-i}) > u_i(\sigma)$. Let r be the root node of G_s. The value of $maxU(r)[i]$ cannot be less than the value of $u_i(s_i, \sigma_{-i})$ in the restriction to G_s, because $maxU$ derives from optimal actions, and these cannot be worse than s_i's actions. The value of $U(r)[i]$ is equal to the value of $u_i(\sigma)$ in the restriction to G_s, hence $maxU(r)[i] > U(r)[i]$, violating Requirement 1 (contradiction). The "only if" part (by contradiction): Assume σ is a SPNE for G, but Requirement 1 does not hold. If Requirement 1 does not hold then there is some agent i and node d, where d is the root of G_s, a proper subgame of G, such that $maxU(d)[i] > U(d)[i]$. We simply build a restricted strategy s_i (restricted to G_s) which takes actions which will result in utility $maxU(n)[i]$ from any node n in G_s (given that opponents are following σ_{-i}). Now, taking restrictions to G_s: $u_i(s_i, \sigma_{-i}) > u_i(\sigma)$, violating the requirements for SPNE (contradiction). ■

4.1 An Algorithm for Checking a SPNE

We need to build the game tree by running through its program for all possible choices. Then we can do a simple backwards induction to calculate U and $maxU$ at all nodes, and hence determine if Requirement 1 is violated anywhere. The only potentially tricky bit is how to recognise proper subgames. This can be done during the backwards induction phase; we define a common ancestor function $ca(n)$ as follows: if n is terminal then $ca(n) = n$; if n is non-terminal then $ca(n)$ is the node which is a prefix of every node in $I(n)$ and for all $s \in S(n)$, n must be a prefix of $ca(s)$ too. Note that a node is a prefix of itself. It is easy to see that, for any node n, if $n = ca(n)$ then n is the root of a proper subgame. To see this note that if n is the prefix of every node in $I(n)$ then it is a singleton information set. Also, for any node p which is on a path descending from n in the tree, if p had an information set which included nodes not on a path descending from n in the tree, then $ca(p)$ would precede n in the tree, and the ca value for any ancestor of p would have to be $ca(p)$ or an ancestor of $ca(p)$.

The checking algorithm (Figure 4) has as inputs the SMPL program $prog$ which defines the game, and the strategy tuple $Strat$ which defines the strategy profile. In order to do the checking, it will build a tree, which is a set of nodes. Each node is a 5-tuple: $\langle m, h_m, h_0, A, U\rangle$, m is the agent who has the move at this node; h_m is the history which is apparent to the agent who has the move (this is important to determine information sets); h_0 is the global history; A is the set of actions available to agent m; U records the utility obtained if the path proceeding from this node is followed, by taking the actions recommended by the strategy profile.

Firstly we build the tree T, using the recursive algorithm $exp-andBranch$ (Figure 5). Next we create a new set of nodes T^p with extended information; these nodes have the form $\langle h_0, U, maxU, ca\rangle$. Using these we propagate the U, $maxU$ and ca values up the tree, annotating nodes of T^p from the bottom up, because nodes higher up the tree derive their utility from their descendants. Once we build a new node of T^p we can discard its successors as they will no longer be needed (each node has a unique direct predecessor).

```
# algorithm: expandBranch build the tree
# inputs: s: state, m: mover, A: actions,
          H: history tuple, prog: program
# outputs: Nodes
Nodes := {}
for each a ∈ A do
   ⟨s, m, A, H⟩ := takeAction(s, m, a, H, prog)
   if terminal(s, prog) = True
   then U := utility(s)
   else  U := null
         SuccNodes := expandBranch(s, m, A, H, prog)
         Nodes := Nodes ∪ SuccNodes
   fi
   node := ⟨m, H[m], H[0], A, U⟩
   Nodes := Nodes ∪ {node}
od
return Nodes
```

Figure 5: Algorithm: expandBranch

If n is its own common ancestor, then n is the root of a proper subgame; for such nodes we check if $maxU$ exceeds U.

The $getAll(S, t)$ function returns all elements of set S which match the template t; it returns $null$ if there are none. Similarly, $getAny(S, t)$ function returns one element, it is used when only one element of set S will match t. The $commonAncestor(ca_1, ca_2)$ function returns the longest common prefix of ca_1 and ca_2, which may turn out to be one of the input arguments.

CLAIM 4.2. *For an input SMPL program $prog$, which represents a game G, and a tuple of strategies $Strat$, which represent a strategy profile σ, Algorithm $checkSPNE$ correctly decides if σ is a SPNE for G.*

Proof Sketch (by induction on the length of nodes, for the main **repeat** loop) We must show that for each node n in the game: n is assigned the correct values of $U(n)$, $maxU(n)$ and $ca(n)$. it is simple to show that this is true for the longest nodes in the game, because they are all terminal nodes. Then we show that if this is true for nodes of length l, it is also true for nodes of length l+1. Some nodes of length l+1 may again be terminal; for any node n among the remainder, it is clear that all elements of $S(n)$ are in T^p because they are length l+1 and hence they were added there in a previous iteration of the **repeat** loop. Using those nodes it is straightforward to see how n is assigned the correct values of $U(n)$, $maxU(n)$ and $ca(n)$. The first **for** loop gets the common ancestor with each element of $I(n)$, note that these are retrieved from the complete tree T, and not T^p, so all nodes are present. The second **for** loop gets the common ancestor with the common ancestor of each element $p \in S(n)$. This time the nodes are retrieved from T^p, so they are annotated with $U(p)$, $maxU(p)$ and $ca(p)$, and these utility values are used to calculate $U(n)$ and $maxU(n)$. A special case is $maxU(n)[m]$ where $m = M(n)$, this must be the best of all the $maxU(p)[m]$. ■

The complexity of checking depends very much on the game; let us consider the case of a multi-stage game with observed actions, where there are p players, with a actions to choose from, and m stages. The number of terminal nodes is a^{pm}; to find each of these, and their utilities, will require the SMPL program to be run each time (although not always from the beginning). Thus it is clearly only feasible for games with small numbers of players and stages. None of these terminal nodes can be neglected, because if any has a utility higher than all other plays of the game, then any strategy profile which does not achieve it could not be a SPNE. For this

algorithm: $checkSPNE$ decides if a strategy profile is a SPNE of a game
inputs: $prog$: the program that is the mechanism, $Strat$: the strategy profile
outputs: $True$ or $False$

$\langle s, m, A, H\rangle := initialState(prog)$ # get the initial state of the game's program
$T := expandBranch(s, m, A, H, prog)$ # T is the entire tree
$T^S := sortLongestFirst(T)$ # T^S will be a list of nodes
$T^p := \{\}$ # for nodes with extended information
repeat # repeat for each node in the tree
  $\langle m, h_m, h_0, A, U\rangle := head(T^S); T^S := tail(T^S)$ # pull the head off
  $ca := h_0$; $bestUtil := 0$ # start with common ancestor = this node
  if $U \neq null$ # if it is terminal
  then $maxU := U$
  else # get its information set and common ancestor
    $Iset := getAll(T, \langle \_, h_m, \_, \_, \_\rangle)$ # get all nodes with same apparent history
    for each $node \in Iset$ **do** # every node in information set
      $\langle \_, \_, hist, \_, \_\rangle := node$ # we want the history of each $node \in Iset$
      $ca := commonAncestor(ca, hist)$ # to find the common ancestor
    od
    for each $a \in A$ **do** # every action that can be taken
      $succNode := getAny(T^p, \langle (h_0 \bullet a), \_, \_, \_\rangle)$ # get extended information for successor nodes
      $T^p := T^p \setminus \{succNode\}$ # we no longer need the extended information
      $\langle \_, maxU_s, U_s, ca_s\rangle := succNode$ # get utilities and common ancestor of successor nodes
      $ca := commonAncestor(ca, ca_s)$ # common to our current node and the successor node
      if $a = run(Strat[m], h_m)$ # was this successor node recommended?
      then $U := U_s; maxU := maxU_s$ # then propagate utilities
      fi
      if $maxU_s[m] > bestUtil$
      then $bestUtil := maxU_s[m]$
      fi
    od
    $maxU[m] := bestUtil$
    if $(h_0 = ca) \wedge (maxU[m] > U[m])$ # $h_0 = ca$ if it's a proper subgame
    then return $False$
    fi
  fi
  $T^p := T^p \cup \{\langle h_0, U, maxU, ca\rangle\}$ # this will be a successor node for the next iteration
until $T^S = [\,]$
return $True$

Figure 4: Algorithm: checkSPNE

reason we are looking into weaker equilibrium notions, where some portions of the tree can be neglected.

5. COMPARISON WITH RELATED WORK

This paper has a similar motivation to previous work [6], but a different approach. Firstly, the previous work did not tackle the issue of enabling agents to check the properties of published specifications for themselves; instead it proposed an offline verification which would be carried out by the agent protocol designer. However, agents in open e-commerce systems might not necessarily trust the protocol designer, and furthermore, we envisage scenarios where agents themselves could generate protocols on the fly, tailored for a specific auction scenario, for example. This means agents need to be able to check the properties of a published specification for themselves. The second major difference is that the previous work proposed that the mechanism be specified in the form of a mapping from strategy profiles to outcome scenarios; this is infeasible for all but the most simple of mechanisms; for example, if there are n agents with m strategies each, then there will be m^n strategy profiles, and each agent's strategy could itself require an unwieldy representation. Worst of all, even extremely simple games can have an astronomical number of possible strategies. This is why we have moved to an algorithmic representation for mechanisms.

The work of Marc Pauly [13] has had a major influence on our approach. Pauly also has explicit choice statements as part of the syntax of his Mechanism Programming Language (MPL). He proves game-theoretic properties for 2-player games using correctness assertions, via an extension of Hoare's calculus. The main advantage of Pauly's approach is that it offers the possibility of verifying large games without needing to construct the entire tree; the advantage of our algorithmic checking approach is that it offers the possibility of agents checking mechanisms automatically. Two further differences relate to preferences and information sets. Pauly incorporates explicit preferences in the verification framework; this is absent in our framework. On the other hand, our framework allows games with arbitrary information sets; Pauly's is restricted to games of "almost-complete" information, i.e. where the only non-singleton information sets are those resulting from simultaneous moves. Preferences and information sets are closely related when one considers our intended future application to auctions; incomplete information games can be modeled as imperfect information games, and hence we need information sets to obscure the knowledge of other agents' preferences.

In another paper Pauly and Wooldridge [14] do take an automated model checking approach to the mechanism verification problem, and set forth their vision for how this approach can contribute to the *mechanism design problem* of game theorists. The paper shows, as an example, how a voting mechanism can be formalised and checked to see if a coalition of agents can force a deadlock indefinitely. It would be interesting to investigate how the checking of equilibria could be performed through ATL formulae. The main difference between that paper, and ours, is with respect to the motivation. Pauly and Wooldridge aim to contribute to the area of mechanism design by providing computing tools which can make mechanism specifications unambiguous, reveal hidden assumptions and automate the process of proving that the mechanisms possess desired properties. In contrast, our proposal aims to contribute to multi-agent systems; we aim not to design new mechanisms, but to make existing ones useful to agents in open systems, by giving agents a way of checking the properties of a previously unseen game specification.

Apart from the above work, there appears to be very little work in the area of verifying properties of games. There is however related work in the area of specifying and solving games, and also in the area of specifying protocols for multi-agent systems. This work will be reviewed briefly now.

5.1 Specifying Games

Although their motivations are different to ours, a number of related works illustrate various different ways in which games can be specified. Gambit [10] can automatically compute the equilibria of games, and it allows games to be represented in extensive form or normal form. The GAMUT [11] software has the capability to generate a wide variety of games, where Java classes are written to encode each type of game; The software also has the ability to output games in a form readable by Gambit. Gala [8] uses an innovative declarative language for representing and solving imperfect information games; this allows the rules of the game to be specified, and removes the need to explicitly represent each possible game state. This language is based on Prolog, and, similar to our own, it includes **choose** statements at choice points, and is deterministic if all choices are specified. Of the above approaches, the Gala language appears to be the most promising for games with extremely large numbers of possible states.

5.2 Specifying Agent Communication Protocols

In the area of agent communication there are a number of approaches which are sufficiently general to capture arbitrary games, for example Electronic Institutions [1, 3] or Commitment Protocols [18]. These works are concerned with facilitating agent communication at a high level, via the specification of protocols using notions such as norms and commitments. These approaches to agent communication, and other similar proposals, are sufficiently generic to allow the specification of arbitrary games; one can specify a game by specifying what agents are allowed to do at each protocol state. While it would be possible to specify our games in these terms, it would add a significant overhead; we have instead opted for a language which specifies only the essential information and hence represents games as simply as possible.

6. DISCUSSION AND FUTURE DIRECTIONS

This paper opens up an interesting new area: publishing game theory mechanisms in a machine readable format, along with a claim about some properties of the mechanism (for example a purported equilibrium), and then automatically checking that the claimed properties do indeed hold. In the future we are interested in exploring this area and finding the limits of the approach. For simple mechanisms the approach certainly does seem feasible. The published specification of a game and strategy profile is extremely concise; on the downside, the entire game tree does need to be built during the verification, but this is similar to the exploration done by model checkers, an approach which is currently feasible even for very large numbers of states. The algorithm presented in this paper would only be feasible with small finite domains for the values which agents can choose, but as we extend these domains, we intend to borrow techniques which have been successfully employed by the model checking community.

Our current plan is for a generalisation of the current work to check for Perfect Bayesian Equilibrium. This would need to take agents' preferences into account. We can foresee a published strategy recommendation which maps preferences to strategies; i.e. in the form of an algorithm which can provide the following type of recommendation to an agent: "If x is your preference over outcomes, then y is your best strategy". In particular, we are inter-

ested in applying the framework to auctions, and so facilitating the publication of auction specifications in electronic institutions where trading agents are free to roam between different auction houses. Note that verifying the equilibrium of an auction is not possible with the current framework, because firstly preferences cannot be specified, and secondly, in an auction we should be checking for a Bayesian equilibrium, rather than a subgame perfect equilibrium.

Ultimately we would like to explore the space of possible mechanisms and their properties, to find the limits of the approach; i.e. to find the boundary where the complexity of the mechanism makes the approach advocated here infeasible. This would give valuable information about the types of mechanisms which should be used in scenarios with resource bounded agents.

Acknowledgements

Special thanks to Emmanuel Tadjouddine and to the anonymous referees and for their suggestions.

7. REFERENCES

[1] M. Esteva, J. Rodriguez, C. Sierra, and P. Garcia. On the formal specification of electronic institutions. In *LNAI 1991*, pages 126–147. Springer, 2001.

[2] D. Fudenberg and J. Tirole. *Game Theory*. MIT Press, 1991.

[3] A. Garcia-Camino, J. Rodriguez-Aguilar, C. Sierra, and W. Vasconcelos. A rule-based approach to norm-oriented programming of electronic institutions. *SIGecomm Exchanges (Newsletter of the ACM Special Interest Group on E-Commerce)*, 5.5, 2006.

[4] L. Giordano, A. Martelli, and C. Schwind. Specifying and verifying systems of communicating agents in a temporal action logic. In *AI*IA 2003: Advances in Artificial Intelligence; LNCS, vol. 2829*, pages 262 – 274. Springer-Verlag, 2003.

[5] G. Gottlob, G. Greco, and F. Scarcello. Pure nash equilibria: Hard and easy games. *Journal of Artificial Intelligence Research*, 24:357–406, 2005.

[6] F. Guerin and J. V. Pitt. Guaranteeing properties for e-commerce systems. In J. Padget, D. Parkes, O. Shehory, and N. Sadeh, editors, *LNAI volume 2531: Agent-Mediated Electronic Commerce IV. Designing Mechanisms and Systems*, pages 253–272. Springer-Verlag, Heidelberg, 2002.

[7] F. Guerin and J. V. Pitt. Verification and compliance testing. In M.-P. Huget, editor, *LNAI volume 2650: Communication in Multiagent Systems: Agent Communication Languages and Conversation Policies*, pages 253–272. Springer-Verlag, 2003.

[8] D. Koller and A. Pfeffer. Representations and solutions for game-theoretic problems. *Artificial Intelligence*, 94(1):167–215, 1997.

[9] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems (Safety), vol. 2*. Springer-Verlag, New York, Inc., 1995.

[10] R. D. McKelvey, A. M. McLennan, and T. L. Turocy. Gambit: Software tools for game theory, version 0.2006.01.20 http://econweb.tamu.edu/gambit, 2006.

[11] E. Nudelman, J. Wortman, Y. Shoham, and K. Leyton-Brown. Run the gamut: A comprehensive approach to evaluating game-theoretic algorithms. In *International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 880–887, 2004.

[12] C. Papadimitriou. Algorithms, games and the internet. In *Proceedings of the Annual Symposium on Theory of Computing (STOC)*, pages 749–753, 2001.

[13] M. Pauly. Programming and verifying subgame perfect mechanisms. *Journal of Logic and Computation*, 15(3):295–316, 2005.

[14] M. Pauly and M. Wooldridge. Logic for mechanism design - a manifesto. In *Proceedings of the 2003 Workshop on Game Theory and Decision Theory in Agent-based Systems (GTDT-2003), Melbourne, Australia*, 2003.

[15] R. M. van Eijk, F. S. de Boer, W. van der Hoek, and J.-J. Meyer. A verification framework for agent communication. *Journal of Autonomous Agents and Multi-Agent Systems*, 6(2):185–219, 2003.

[16] M. Venkatraman and M. P. Singh. Verifying compliance with commitment protocols: Enabling open web-based multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 2(3):217–236, 1999.

[17] B. von Stengel. Computing equilibria for two-person games. In *Handbook of Game Theory with Economic Applications, Vol. 3, eds. R. J. Aumann and S. Hart*. Elsevier, Amsterdam, 2002.

[18] P. Yolum and M. Singh. Reasoning about commitments in the event calculus: An approach for specifying and executing protocols. *Annals of Mathematics and AI, To appear.*, 2003.