Logic for Mechanism Design — A Manifesto

Marc Pauly and Michael Wooldridge
Department of Computer Science
University of Liverpool
Liverpool L69 7ZF, UK
pauly,mjw@csc.liv.ac.uk

ABSTRACT

We advance two main claims. The first is that logics — and in particular, modal strategy logics — are a powerful formal tool for representing and reasoning about the properties of game theoretic mechanisms. The second is that techniques developed for the formal specification and verification of computer systems (and in particular, the use of logic for specification, and automated verification via model checking) can be usefully applied to mechanism design problems. We begin by reviewing the mechanism design problem, and discussing the issue of how mechanisms might be specified and verified. We then discuss a logic (Alternating-time Temporal Logic — ATL), which is well suited to representing and reasoning about strategic encounters. We show how ATL can be used to formally specify two social choice mechanisms, and we demonstrate how current model checkers for ATL can be used to verify properties of these mechanisms. We conclude by discussing issues for future research.

1. INTRODUCTION

The problem of *mechanism design*, as understood and studied in the game theory community, has recently begun to attract attention in the computer science community. This interest has been spurred at least in part by the development of electronic auction houses and automated negotiation techniques, in which the participants are not humans, but software agents. Although algorithmic approaches to mechanism design, and the computational issues surrounding these approaches, have begun to attract some attention in the computer science community [24], very little research has to date addressed the issue of systematically specifying, implementing, and verifying mechanisms from the perspective of computer science.

The purpose of this paper is to advance two key claims.

- The first claim is that logics — specifically, modal logic [9], and more specifically, modal strategy logics such as coalition logic [26, 27, 28] and Alternating-time temporal logic [3] — have an important role to play in the specification, implementation, and verification of mechanisms. Modal strategy logics in this sense play a role analogous to that of (for example) temporal logics in the development of reactive systems (see, e.g., [23]).
- Our second claim is that techniques developed for the automated verification of computer systems can be usefully applied to mechanism design problems in the game-theoretic sense. Arguably the most successful technology developed in the formal methods community is that of *model checking* [11], which was originally intended as a technique for verifying that finite state concurrent systems satisfy specifications expressed in the language of temporal logic [12]. Just as model checking has played an enormously successful role in the verification of finite state concurrent systems, so we believe that it can have an analogous role to play in the analysis and verification of mechanisms.

We proceed as follows. We begin by reviewing the mechanism design problem, comparing the issues that arise in mechanism design to those that arise in the development of provably correct computer systems, and discuss how mechanisms might be specified and verified. We then review a logic (Alternating-time Temporal Logic — ATL), which is well suited to representing and reasoning about strategic, multi-agent encounters. By way of illustrating our approach, we show how ATL can be used to formally specify two social choice mechanisms, and we demonstrate how current model checkers for ATL can be used to verify properties of these mechanisms. We conclude by discussing issues for future research.

2. MECHANISMS, PROGRAMS, AND CORRECTNESS

Computer science and game theory share a concern with *mechanisms*, broadly conceived. The mechanisms of computer science are *computational mechanisms*, or programs, which may be modeled mathematically in a variety of ways (e.g., as input-output relations, Turing machines, or state transition diagrams to name just three popular alternatives). *Game-theoretic mechanisms*, or game forms, on the other hand can be viewed as *multi-agent programs*, where a number of players are involved in determining the behaviour of the mechanism. The main models for mechanisms used within noncooperative game theory are strategic and extensive game forms. There is a great degree of similarity between extensive game forms and computational models. This similarity is perhaps most apparent when comparing a state transition diagram of a nondeterministic program on the one hand, and an extensive game form of perfect information on the other [7]. As has been discussed more formally elsewhere [26], conventional (e.g., deterministic Turing machine) programs can thus be viewed formally as 1-player game forms. At this level of abstraction, therefore, the notion of a game form *subsumes* the notion of a program. Thus, the notion of a mechanism we arrive at is sufficiently general to encompass computer programs, as well as various social procedures such as voting procedures, auctions, fair division algorithms — and multiagent systems.

Computer science has developed a great deal of theory about what it means for a computer program to be *correct* [10]. In general, verifying the correctness of a program means checking that the program *satisfies its specification*. Checking correctness might involve proving, e.g., that a program has indeed calculated the greatest common divisor of two numbers A and B; that two subprocesses do not read and write into a memory location at the same time; or that no subprocess deadlocks. Starting with the work of Hoare [18], various *deductive* methods for reasoning about program correctness have been developed, some of which have turned into full-fledged program logics such as Dynamic Logic [16]. More recently, *semantic* approaches to verification — and in particular, model checking — have been successfully used to verify that systems satisfy properties expressed in various temporal and modal logics [11].

While a correct program may be seen as an implementation of its specification, game theory has its own theory of mechanism design and implementation. The *mechanism design* problem in the context of game theory is in fact quite a broad notion, but it is most often expressed in the context of auction design (see, e.g., [8, pp.523–536] for an overview, and [22] for an in-depth study). In the general setting, the problem is understood in terms of a "principal", who wishes to design a protocol (in fact, a mechanism), for a collection of agents, so that if the agents behave rationally when enacting the protocol (in the sense of, for example, attempting to maximise their expected utility), then the resulting overall behaviour will be optimal for the principal. In the context of auction design, for example, the principal may be identified with an agent selling some good, and may wish to simply maximise expected revenue.

Of course, mechanism design is not restricted to auctions. For example, voting procedures — social choice functions — can be understood as mechanisms in exactly the same way. Without going into any detail, consider a social choice function c which, given the preferences p of the individual agents over a set of alternatives, selects a particular alternative $c(p)$ as the social choice. A *Nash-implementation* of this social choice function c will be a strategic game form such that for every preference profile p there will be a Nash-equilibrium whose outcome is $c(p)$. Things are complicated somewhat by the fact that games may have multiple Nash equilibria and that social choice functions often will select not only one but multiple alternatives. Also, analogous implementation notions exist which are based on refinements of the Nash-equilibrium.

As we noted above, the formal, logic-based specification and verification of computing mechanisms (i.e., programs) has received considerable attention in the computer science community. However, there has been only very little work in game theory on formal, logical approaches to mechanism verification. It is thus the aim of this manifesto to (a) argue for the need of work in this area, (b) develop some key points for a research agenda, and (c) illustrate the general approach by means of two modest, but meaningful case studies.

Before proceeding, it may be useful to point out the advantages of formal logic approaches to mechanism specification and verification. While many computer scientists may already be convinced by the arguments for the formal verification of computer programs, the game theorist may not know these arguments and may also wonder whether these arguments can simply be transferred from computer programs to game-theoretic mechanisms. We can summarise the key benefits of such an approach as follows.

Explicitness: Formal verification requires specifying the mechanism under consideration in great detail, removing all ambiguities and vagueness, making explicit any hidden assumptions. For an auction mechanism, this may refer to tie-breaking rules, whether bids have to be made in increments of pounds rather than pennies, and so on. While this level of detail will often be a nuisance to the mechanism designer, it has turned out that such details can greatly influence the properties of a program (e.g., leading to termination failure) or a mechanism (e.g., leading to signaling phenomena through bids, greatly reducing auction revenue).

Automation The greater degree of explicitness allows us to automate mechanism verification, obtaining tools which perform a semi-automatic analysis of the mechanism under consideration, indicating why, for example, a particular desirable property fails. As a result, while losing time by having to specify mechanisms to a greater level of detail, we make up for it by automating (part of) the mechanism's analysis. Thus, developing a logical system for reasoning about mechanisms is not only a tool which illuminates conceptual difficulties, it also provides the prerequisite for automated tools for mechanism analysis. We are thinking here primarily of the use of *model checking* as a tool for automated verification [11].

Confidence As a result of the added degree of explicitness and a computer-verified mechanism, we have increased our confidence in the correctness of the mechanism, partly because we will have already eliminated some design errors in the process. Note that for all practical purposes, this will not lead to a guarantee that the mechanism is optimal. First, there may be properties which turn out to be important but which we did not check for. Second, the automated verification tools may themselves have errors, and hence errors in the mechanism may go undetected.

Having made the argument for why logical approaches to mechanism specification and verification are likely to be useful, our next step is to illustrate our approach by means of two case studies. In these case studies, we use a logic called Alternating-time Temporal Logic (ATL) [3]; we first review the syntax and semantics of this logic.

3. ATL

In 1997, Alur, Henzinger, and Kupferman introduced a temporal logic intended to make it possible to express cooperative properties of multiagent systems [3]. This logic, Alternating-time Temporal Logic (ATL), can be understood as a generalisation of the well-known branching time temporal logic CTL [12], in which path quantifiers are replaced by *cooperation modalities*. A cooperation modality $\langle\langle\Gamma\rangle\rangle\varphi$, where Γ is a group of agents, expresses the fact that the group Γ can cooperate to ensure that φ; more precisely, that there exists a strategy profile for Γ such that by following this strategy profile, Γ can ensure φ. Thus, for example, the system requirement "agents 1 and 2 can cooperate to ensure that the system never enters a fail state" may be captured by the ATL formula $\langle\langle 1, 2\rangle\rangle \Box\neg \mathit{fail}$. The "$\Box$" temporal operator means "now and forever more": additional temporal connectives in ATL are "$\Diamond$" ("either now or at some point in the future"), "$\mathcal{U}$" ("until"), and "$\bigcirc$" ("in the next state").

One of the main reasons for the current level of interest in ATL is that the ATL model checking problem is in general no more complex than that of CTL: it can be solved in time $O(\ell \times m)$, where ℓ is the size of the formula, and m is the size of the model in which the formula is to be checked. This tractability result has led to the development of an ATL model checking system called MOCHA [4, 1].

To give a precise definition of ATL, we must first introduce the semantic structures over which formulae of ATL are interpreted. An *alternating transition system* (ATS) is a 5-tuple $\langle \Pi, \Sigma, Q, \pi, \delta \rangle$, where:

- Π is a finite, non-empty set of *atomic propositions*;
- $\Sigma = \{a_1, \ldots, a_n\}$ is a finite, non-empty set of *agents*;
- Q is a finite, non-empty set of *states*;
- $\pi : Q \to 2^{\Pi}$ gives the set of primitive propositions satisfied in each state;
- $\delta : Q \times \Sigma \to 2^{2^Q}$ is the system transition function, which maps states and agents to the choices available to these agents. Thus $\delta(q, a)$ is the set of choices available to agent a when the system is in state q. We require that this function satisfy the requirement that for every state $q \in Q$ and every set $Q_1, \ldots, Q_n$ of choices $Q_i \in \delta(q, a_i)$, the intersection $Q_1 \cap \cdots \cap Q_n$ is a singleton.

We denote the set of sequences over Q by Q^*, and the set of non-empty sequences over Q by Q^+.

An ATL formula, formed with respect to an alternating transition system $S = \langle \Pi, \Sigma, Q, \pi, \delta \rangle$, is then one of the following:

(S1) p, where $p \in \Pi$ is a primitive proposition;

(S2) $\neg\varphi$ or $\varphi \vee \psi$, where φ and ψ are formulae of ATL;

(S3) $\langle\langle\Gamma\rangle\rangle \bigcirc \varphi$, $\langle\langle\Gamma\rangle\rangle \Box \varphi$, or $\langle\langle\Gamma\rangle\rangle \varphi \mathcal{U} \psi$, where $\Gamma \subseteq \Sigma$ is a set of agents, and φ and ψ are formulae of ATL.

To give a precise semantics to ATL, we need some further definitions. (Note that the semantics given here is based on [3] and differs slightly from the semantics in [2].) For two states $q, q' \in Q$ and an agent $a \in \Sigma$, we say that state q' is an *a-successor* of q if there exists a set $Q' \in \delta(q, a)$ such that $q' \in Q'$. Intuitively, if q' is an a-successor of q, then q' is a possible outcome of one of the choices available to a when the system is in state q. We denote by $succ(q, a)$ the set of a successors to state q. We say that q' is simply a *successor* of q if for all agents $a \in \Sigma$, we have $q' \in succ(q, a)$; intuitively, if q' is a successor to q, then when the system is in state q, the agents Σ can cooperate to ensure that q' is the next state the system enters.

A *computation* of an ATS $\langle \Pi, \Sigma, Q, \pi, \delta \rangle$ is an infinite sequence of states $\lambda = q_0, q_1, \ldots$ such that for all $u > 0$, the state q_u is a successor of q_{u-1}. A computation $\lambda \in Q^+$ starting in state q is referred to as a *q-computation*; if $u \in \mathbb{N}$, then we denote by $\lambda[u]$ the u'th state in λ; similarly, we denote by $\lambda[0, u]$ and $\lambda[u, \infty]$ the finite prefix $q_0, \ldots, q_u$ and the infinite suffix $q_u, q_{u+1}, \ldots$ of λ respectively.

Intuitively, a *strategy* is an abstract model of an agent's decision-making process; a strategy may be thought of as a kind of plan for an agent. By *following* a strategy, an agent can bring about certain states of affairs. Formally, a strategy f_a for an agent $a \in \Sigma$ is a total function $f_a : Q^+ \to 2^Q$, which must satisfy the constraint that $f_a(\lambda \cdot q) \in \delta(q, a)$ for all $\lambda \in Q^*$ and $q \in Q$. Given a set $\Gamma \subseteq \Sigma$ of agents, and an indexed set of strategies $F_\Gamma = \{f_a \mid a \in \Gamma\}$, one for each agent $a \in \Gamma$, we define $out(q, F_\Gamma)$ to be the set of possible outcomes that may occur if every agent $a \in \Gamma$ follows the corresponding strategy f_a, starting when the system is in state $q \in Q$. That is, the set $out(q, F_\Gamma)$ will contain all possible q-computations that the agents Γ can "enforce" by cooperating and following the strategies in F_Γ. Note that the "grand coalition" of all agents in the system can cooperate to uniquely determine the future state of the system, and so $out(q, F_\Sigma)$ is a singleton. Similarly, the set $out(q, F_\emptyset)$ is the set of all possible q-computations of the system.

We can now give the rules defining the satisfaction relation "$\models$" for ATL, which holds between pairs of the form S, q (where S is an ATS and q is a state in S), and formulae of ATL:

- $S, q \models p$ iff $p \in \pi(q)$ (where $p \in \Pi$);
- $S, q \models \neg\varphi$ iff $S, q \not\models \varphi$;
- $S, q \models \varphi \vee \psi$ iff $S, q \models \varphi$ or $S, q \models \psi$;
- $S, q \models \langle\langle\Gamma\rangle\rangle \bigcirc \varphi$ iff there exists a set of strategies F_Γ, such that for all $\lambda \in out(q, F_\Gamma)$, we have $S, \lambda[1] \models \varphi$;
- $S, q \models \langle\langle\Gamma\rangle\rangle \Box \varphi$ iff there exists a set of strategies F_Γ, such that for all $\lambda \in out(q, F_\Gamma)$, we have $S, \lambda[u] \models \varphi$ for all $u \in \mathbb{N}$;
- $S, q \models \langle\langle\Gamma\rangle\rangle \varphi \mathcal{U} \psi$ iff there exists a set of strategies F_Γ, such that for all $\lambda \in out(q, F_\Gamma)$, there exists some $u \in \mathbb{N}$ such that $S, \lambda[u] \models \psi$, and for all $0 \leq v < u$, we have $S, \lambda[v] \models \varphi$.

4. TWO CASE STUDIES

Our aim in this section is to add some technical substance to the ideas and claims we have sketched out thus far. We will demonstrate how the approach works with respect to scenarios from the domain of social choice. Thus, we want mechanisms that will choose an outcome from a set of possible outcomes, in accordance with certain principles. Of course, there are impossibility results in the social choice domain, (Arrow's impossibility theorem being probably the best known [5]), which tell us that there are rather important limits to the general applicability of any mechanism that we might find. However, these negative results do not preclude the generation of mechanisms that are of value for particular scenarios.

So, we will consider two social choice scenarios, of increasing complexity. For both, we will demonstrate how the requirements for these scenarios can be naturally captured in Alternating-time Temporal Logic. We will then show how model checking tools for ATL may be used to verify these mechanisms.

4.1 A Social Choice Scenario

The first scenario we consider is adapted from [26]. Consider the following, informal, requirements for a social choice mechanism.

> *Two agents, A and B, must choose between two outcomes, p and q. We want a mechanism that will allow them to choose, which will satisfy the following requirements. First, whatever happens, we definitely want an outcome to result — that is, we want either p or q to be selected. Second, we really do want the agents to be able to collectively choose an outcome. However, we do not want them to be able to bring about both outcomes simultaneously. Similarly, we do not want either agent to dominate: we want them both to have equal power.*

It should be clear that we can naturally capture these requirements using ATL, as follows.

$$\langle\langle\rangle\rangle \bigcirc (p \vee q) \quad (1)$$
$$\langle\langle A, B\rangle\rangle \bigcirc p \wedge \langle\langle A, B\rangle\rangle \bigcirc q \quad (2)$$
$$\neg\langle\langle A, B\rangle\rangle \bigcirc (p \wedge q) \quad (3)$$
$$\neg\langle\langle A\rangle\rangle \bigcirc p \wedge \neg\langle\langle B\rangle\rangle \bigcirc p \quad (4)$$
$$\neg\langle\langle A\rangle\rangle \bigcirc q \wedge \neg\langle\langle B\rangle\rangle \bigcirc q \quad (5)$$

The first requirement states that an outcome *must* result: this will happen inevitably, whatever the agents do. Requirement (2) states that the two agents can choose between the two outcomes: they have a collective strategy such that, if they follow this strategy, outcome x will occur, where x is either p or q. Requirement (3), however, says that the agents cannot choose *both* outcomes. Requirements (4) and (5) state that neither agent can bring about an outcome alone. It should be immediately obvious how these axioms capture the requirements as stated above.

Now consider the following mechanism, intended to permit the agents to select between the outcomes in accordance with these requirements.

> *The two agents vote on the outcomes, i.e., they each choose either p or q. If there is a consensus, then the consensus outcome is selected; if there is no consensus, (i.e., if the two agents vote differently), then an outcome p or q is selected non-deterministically.*

Notice that, given this simple mechanism, the agents really can collectively choose the outcome, by cooperating. If they do not cooperate, however, then an outcome is chosen for them.

Having formally set out the desirable properties that we wish a mechanism to satisfy, and having described a mechanism that we believe satisfies these properties, our next step is to formally verify that the mechanism does indeed satisfy them. We do this via model checking: we express the mechanism as a model suitable for the ATL model checking system MOCHA, and then, using MOCHA, we check whether the requirements are realised in this model.

A MOCHA model of the mechanism is given in Figure 1. While space restrictions preclude a detailed introduction to the modelling language of MOCHA, it is nevertheless worth briefly describing the key features of this representation. We model the scenario via three agents, which in MOCHA terminology are called `module`s:

- `AgentA` and `AgentB` correspond to the A and B in our scenario. Each agent controls (i.e., has exclusive write access to) a variable that is used to record their vote. Thus `voteA` records the vote of `AgentA`, where a value of `false` in this variable means voting for outcome `P`, while `true` implies voting for `Q`. The "program" of each agent is made up of two remaining guarded commands, which simply present the agent with a choice of voting either way.

- The `Environment` module is used to model the mechanism itself. This module simply looks at the two votes, and if they are the same, sets the variable `outcome` to be the consensus outcome; if the two votes are different, then the guarded commands defining `Environment`'s behaviour say that an outcome will be selected non-deterministically.

Notice that in translating this simple mechanism in a form suitable for MOCHA, it has not been possible to remain entirely neutral with respect to all issues. For example, the way we have coded the mechanism means that it is in principle possible for one agent to see another agent's vote (i.e., votes are common knowledge), even though, in the implementation given here, agents do not make any use of this information. The informal description of the mechanism — and indeed, the original requirements — said nothing about whether votes (and hence preferences) should remain hidden or should be common knowledge, and in fact, we could have coded the scenario in such a way that an agent's vote was visible only to the `Environment` module. But the point is that we have been forced to make a commitment one way or the other by the need to code the scenario. (It is of course likely that in more sophisticated (and realistic) scenarios, we would desire votes to remain private. To express such requirements, we would require the use of *knowledge modalities*, of the type described in [13]. We will not explore this issue further, except to report that preliminary work on extending ATL with knowledge modalities is described in [19, 21, 20].)

Having captured the mechanism in the modelling language of MOCHA, we can now actually use MOCHA to check the properties (1)–(5), above. Property (1) turns out to hold trivially, by the law of the excluded middle, from the way in which we represent the outcome. Property (2) can be represented in the following two MOCHA properties to be checked.

```
atl "f00"
  <<>> G  <<AgentA, AgentB>> X outcome ;
atl "f01"
  <<>> G  <<AgentA, AgentB>> X ~outcome;
```

The prefix to these formulae (`<<>> G`) simply says that we check this property across all reachable states of the system; the remainder of the property is simply the property to be checked, rewritten in the textual input form required by MOCHA. In exactly the same way, we can check the following properties, and we trust the reader can see that these are direct expressions of the remaining original requirements given above, as follows (note that, again, property (3) follows trivially: `outcome` cannot be both `true` and `false`):

```
atl "f02"
  <<>> G  ~<<AgentA>> X outcome ;
atl "f03"
  <<>> G  ~<<AgentA>> X ~outcome ;
atl "f04"
  <<>> G  ~<<AgentB>> X outcome ;
atl "f05"
  <<>> G  ~<<AgentB>> X ~outcome ;
```

All these properties are correctly checked, as we would hope.

Having checked these properties, we can in fact go on to check some further interesting properties. For example, the following say that the mechanism itself — captured in the `Environment` variable — cannot choose the outcome.

```
atl "f06"
  <<>> G  ~<<Environment>> X outcome ;
atl "f07"
  <<>> G  ~<<Environment>> X ~outcome ;
```

Again, these properties correctly hold, as we would expect.

Note that the example described in this section only makes use of a very limited fragment of ATL known as Coalition Logic [28]. The relationship between ATL and Coalition Logic is investigated in [14].

4.2 Eternal Voting

For the second case study (adapted from [26, pp.99-101]), we consider a more complex scenario, as follows.

> *A political body $\Sigma = \{1, 2, 3, 4\}$ has to decide on passing a new law. There are two versions of the law,*

```
-- voteA == false ...  agent A votes for outcome P
-- voteA == true ...  agent A votes for outcome Q
module AgentA
   interface voteA : bool
   atom controls voteA
   init update
        [] true -> voteA' := false
        [] true -> voteA' := true
   endatom
endmodule

-- voteB == false ...  agent B votes for outcome P
-- voteB == true ...  agent B votes for outcome Q
module AgentB
   interface voteB : bool
   atom controls voteB
   init update
        [] true -> voteB' := false
        [] true -> voteB' := true
   endatom
endmodule

-- outcome == false ...  P is selected
-- outcome == true ...  Q is selected
module Environment
   interface outcome :  bool
   external voteA, voteB : bool
   atom controls outcome awaits voteA, voteB
   init update
   -- if votes are the same, go with selected outcome
        [] (voteA' = voteB') -> outcome' := (voteA' & voteB')
   -- otherwise select outcome non-deterministically
        [] ~(voteA' = voteB') -> outcome' := true
        [] ~(voteA' = voteB') -> outcome' := false
   endatom
endmodule -- Environment

System := (AgentA || AgentB || Environment)
```

Figure 1: A simple social choice mechanism, defined in the ReactiveModules language of the MOCHA model checker.

`law1` and `law2`, and the process begins by a single agent, agent 2, proposing which of these versions should be adopted. Once agent 2 has selected a version, the entire body votes on whether to accept the proposal; if there is a majority in favour of acceptance, then the proposed version is accepted; if there is a majority against, then there is deadlock, and the process begins again, with agent 2 selecting a version of the law to propose; if there is no majority one way or the other, then the vote of the chairman, agent 1, is decisive, in either accepting the proposed law or returning it to agent 2.

Here, we can use ATL to examine the relative powers of coalitions, and determine whether the mechanism has any undesirable properties; one such undesirable property, for example, would be if it was possible for a coalition to force a deadlock indefinitely, with no agreement ever being reached.

We begin, as before, by modelling the mechanism in a form suitable for MOCHA — the code is given in Figure 2:

- `Agent1` is the "generic" agent in this scenario, which simply has to decide whether to accept or reject the proposed law. The vote of `Agent1` is held in a variable `agree1`. (Agents 3 and 4 are essentially identical, and for this reason we do not give their code.)

- `Agent2` is composed of two separate "update threads". The first of these is responsible for proposing a law when the mechanism is in "subcommittee" phase (i.e., at the start, and whenever a proposal has been rejected by the whole political body). The variable `subCommittee` keeps track of when the mechanism is in subcommittee phase. The proposal made by agent 2 is carried in variable `scchosen`. The second update thread is responsible for deciding whether to accept a proposal, as with the other agents in the system. (Of course, it would in some sense be nonsensical for agent 2 to vote against accepting a proposal that it had put forward, but the mechanism does not preclude it.)

- The `Outcome` module is responsible for determining the outcome, based on the votes of the agents in the system. It is also composed of two update threads. The first is responsible for keeping track of whether the mechanism is in sub-committee phase (this will be initially, and whenever the overall outcome is deadlock). The second update thread decides what the outcome is: the three rules defining this thread correspond to (a) whether there is an equal number of agents for and against accepting the proposal, in which case the mechanism looks to the vote of agent 1, and if this was positive (i.e., agent 1 voted to accept), then the proposal is accepted; otherwise it is rejected; (b) a majority agree on accepting the proposal, in which case the overall outcome is that in the variable `scchosen`; and (c) there is a majority against accepting the proposal, in which case the outcome is `deadlock`. Note that the overall outcome (i.e., `law1`, `law2`, or `deadlock`) is recorded in variable `outcome`. Note that:

  – `MajorityInFavourFmla`,

```
type law : {law1, law2, deadlock}
module Agent1
   external scchosen : law
   interface agree1 : bool
   atom controls agree1 reads scchosen
   update
        [] true -> agree1’ := true
        [] true -> agree1’ := false
   endatom
endmodule -- Agent1
module Agent2
   external subCommittee : bool
   interface agree2 : bool;
        scchosen : law
   atom controls scchosen reads subCommittee
   update
        [] subCommittee -> scchosen’ := law1
        [] subCommittee -> scchosen’ := law2
   endatom
   atom controls agree2 reads scchosen
   update
        [] true -> agree2’ := true
        [] true -> agree2’ := false
   endatom
endmodule -- Agent2
...
module Outcome
   external
        scchosen : law;
        agree1, agree2, agree3, agree4, subCommittee : bool
   interface
        outcome : law;
        subCommittee : bool
   atom controls subCommittee reads outcome awaits outcome
   init
        [] true -> subCommittee’ := true
   update
        [] outcome’ = deadlock -> subCommittee’ := true
        [] ∼(outcome’ = deadlock) -> subCommittee’ := false
   endatom
   atom controls outcome
        reads scchosen, agree1, agree2, agree3, agree4
        awaits scchosen, agree1, agree2, agree3, agree4
   init
        [] true -> outcome’ := deadlock
   update
        [] NoConsensusFmla -> outcome’ :=
           if (agree1’) then scchosen’ else deadlock fi
        [] MajorityInFavourFmla -> outcome’ := scchosen’
        [] MajorityAgainstFmla -> outcome’ := deadlock
   endatom
endmodule -- Outcome
System := (Agent1 || Agent2 || Agent3 || Agent4 || Outcome)
```

Figure 2: An eternal voting scenario, represented in the MOCHA modelling language.

– `MajorityAgainstFmla`, and
– `NoConsensusFmla`

are abbreviations we are using here for MOCHA expressions which capture whether or not there is a majority for or against acceptance, or whether there is a deadlock, respectively.

Now we have our mechanism, we can check some properties. The first three properties demonstrate that the coalition $\{1,2\}$ is in fact all-powerful: this coalition can bring about any outcome, including deadlock. Moreover, they can do this indefinitely.

$$\langle\langle\rangle\rangle \Box \langle\langle 1,2\rangle\rangle \bigcirc (outcome = law_1) \quad (6)$$
$$\langle\langle\rangle\rangle \Box \langle\langle 1,2\rangle\rangle \bigcirc (outcome = law_2) \quad (7)$$
$$\langle\langle\rangle\rangle \Box \langle\langle 1,2\rangle\rangle \bigcirc (outcome = deadlock) \quad (8)$$

These can be straightforwardly encoded in the required MOCHA textual form, as follows — all of these properties were successfully checked against our mechanism.

```
atl "f00"
  <<>> G <<Agent1,Agent2>> X (outcome = law1);
atl "f01"
  <<>> G <<Agent1,Agent2>> X (outcome = law2);
atl "f02"
  << >> G <<Agent1,Agent2>> X (outcome = deadlock);
```

In contrast to the coalition $\{1,2\}$, the coalition $\{3,4\}$ is very weak. In fact, they cannot achieve anything: all the following properties fail when we check them against the mechanism.

```
atl "f03"
  << >> G <<Agent3,Agent4>> X (outcome = law1);
atl "f04"
  << >> G <<Agent3,Agent4>> X (outcome = law2);
atl "f05"
  << >> G <<Agent3,Agent4>> X (outcome = deadlock);
```

This demonstrates that, in fact, our mechanism has several undesirable properties. Not only does it transpire that the coalition $\{1,2\}$

is vastly more powerful than $\{3, 4\}$, it also turns out that the mechanism does not even guarantee an outcome: a deadlock can be forced indefinitely.

5. TOWARDS A RESEARCH AGENDA

While we believe that the ATL-based approach discussed provides an interesting step in the direction of a general formal mechanism verification framework, a lot remains to be done. Below, we mention some key issues which need to be addressed for this research program to be successful:

- We need to incorporate more game-theoretic notions in the logics we use. While the logics discussed are capable of capturing some game theoretic notions, they are still too close to their computer science origins. For example, players' preferences, strategies, equilibrium notions, are all notions which so far are inadequately represented both in the underlying semantic models and in the logical languages used. It is also still an open question whether we will eventually end up with one general-purpose logic which functions as a standard, much the way first-order logic or modal logic do in computer science. Equally possible seems a rich logical landscape of formalisms, each well-equipped to handle a particular range of problems. In analogy with the situation in temporal logic, one could also imagine a combination of these two, where there is a general logical framework (the μ-calculus) with many special purpose fragments (LTL, CTL, ...) for specific applications. There are some promising results on capturing game theoretic notions in modal and dynamic logics. For example, Harrenstein and colleagues show how notions such as Nash equilibrium and subgame perfect Nash equilibrium can be captured in a dynamic logic [17]; similarly, Baltag shows how a range of game theoretic constructs can be captured in dynamic epistemic update logics [6]; and of course, regular epistemic logics can be used to capture many interesting information-theoretic properties of game-like scenarios, such as perfect recall [13, 15].

- We need more interaction with research and researchers in game theory and social choice theory. In order to convince researchers in these areas of the usefulness of formal mechanism verification, they need to be involved from the beginning in the development of these logics.

- Convincing these communities will be greatly facilitated once we have more substantial case studies. An example might be the formal verification of a specific auction procedure. Auction procedures which have been used e.g. in national spectrum auctions provide a good example of mechanisms which are (a) sufficiently complex to demonstrate the usefulness of formal verification, and (b) not too complex to make formal verification infeasible. The extreme complexity of the mechanism under consideration has been a problem which stands in the way of more widespread software verification, but we believe that even rather complicated social mechanisms such as auction procedures are still very simple when compared to common computer programs in use today.

- Eventually, we need computational tools implementing the logical formalisms we come up with. It is only through these tools that semi-automatic verification of mechanisms will be possible. From computer science, we have a great deal of experience with the principles underlying these tools, usually they will rely on theorem proving or model checking.

Naturally, we do not expect this research program to be successfully completed in the near future. We do however believe that this research program deserves our efforts because it can potentially be of great benefit to society as a whole. Furthermore, we do believe that the initial steps which have been taken in this direction and which have been surveyed here provide sufficient grounds for optimism: we are at least beginning to develop the tools necessary to obtain a better analysis of *social software*, in the sense proposed by Parikh [25].

6. REFERENCES

[1] R. Alur, L. de Alfaro, T. A. Henzinger, S. C. Krishnan, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Taşiran. MOCHA user manual. University of Berkeley Report, 2000.

[2] R. Alur, T. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49:672–713, 2002.

[3] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proceedings of the 38th IEEE Symposium on Foundations of Computer Science*, pages 100–109, Florida, October 1997.

[4] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Taşiran. Mocha: Modularity in model checking. In *CAV 1998: Tenth International Conference on Computer-aided Verification, (LNCS Volume 1427)*, pages 521–525. Springer-Verlag: Berlin, Germany, 1998.

[5] K. Arrow. *Social Choice and Individual Values*. Yale University Press, 1951.

[6] A. Baltag. A logic for suspicious players. *Bulletin of Economic Research*, 54(1):1–46, 2002.

[7] J. van Benthem. Extensive games as process models. *Journal of Logic, Language, and Information*, 11(3):289–313, 2002.

[8] K. Binmore. *Fun and Games: A Text on Game Theory*. D. C. Heath and Company: Lexington, MA, 1992.

[9] P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic*. Cambridge University Press: Cambridge, England, 2001.

[10] R. S. Boyer and J. S. Moore, editors. *The Correctness Problem in Computer Science*. The Academic Press: London, England, 1981.

[11] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press: Cambridge, MA, 2000.

[12] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science Volume B: Formal Models and Semantics*, pages 996–1072. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1990.

[13] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. The MIT Press: Cambridge, MA, 1995.

[14] V. Goranko. Coalition games and alternating temporal logics. In J. van Benthem, editor, *Proceedings of the 8th conference on Theoretical Aspects of Rationality and Knowledge (TARK VIII)*, pages 259–272. Morgan Kaufmann, 2001.

[15] J. Y. Halpern. A computer scientist looks at game theory. *Games and Economic Behavior*, 2001.

[16] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. The MIT Press: Cambridge, MA, 2000.

[17] P. Harrenstein, W. van der Hoek, J.-J Meyer, and C. Witteveen. On modal logic interpretations for games. In *Proceedings of the Fifteenth European Conference on Artificial Intelligence (ECAI-2002)*, pages 28–32, Lyon, France, 2002.

[18] C. A. R. Hoare. An axiomatic basis for computer

programming. *Communications of the ACM*, 12(10):576–583, 1969.

[19] W. van der Hoek and M. Wooldridge. Tractable multiagent planning for epistemic goals. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002)*, pages 1167–1174, Bologna, Italy, 2002.

[20] W. van der Hoek and M. Wooldridge. Model checking cooperation, knowledge, and time — a case study. *Research in Economics*, 57(3), September 2003.

[21] W. van der Hoek and M. Wooldridge. Time, knowledge, and cooperation: Alternating-time temporal epistemic logic and its applications. *Studia Logica*, 2003.

[22] V. Krishna. *Auction Theory*. The Academic Press: London, England, 2002.

[23] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag: Berlin, Germany, 1992.

[24] Noam Nisan and Amir Ronen. Algorithmic mechanism design. In *Proceedings of the Thirty-first Annual ACM Symposium on the Theory of Computing (STOC-99)*, pages 129–140, May 1999.

[25] R. Parikh. Social software. *Synthese*, 132(3):187–211, 2002.

[26] M. Pauly. *Logic for Social Software*. PhD thesis, University of Amsterdam, 2001. ILLC Dissertation Series 2001-10.

[27] M. Pauly. A logical framework for coalitional effectivity in dynamic procedures. *Bulletin of Economic Research*, 53(4):305–324, 2002.

[28] M. Pauly. A modal logic for coalitional power in games. *Journal of Logic and Computation*, 12(1):149–166, 2002.