Automated Mechanism Design Using Process Algebra

Emmanuel M. Tadjouddine[1]

Abstract. This paper shows how process algebra can be used to automatically generate verifiable mechanisms for multi-agent systems wherein agents need to trust the system. We make the link between games and process models and then present an iterative algorithm allowing us to generate mechanisms as computer programs implementing given systems' requirements, which are expressed as constraints and desirable properties such as incentive compatibility. This methodology can be used to deploy for example agent mediated e-commerce systems.

1 Introduction

As game theory is used to model and analyze multi-agent systems composed of selfish but rational agents, there is a need to tailor game rules according to given parameters of the underlying system. This is known as *Automated Mechanism Design* (AMD) [3, 23, 24]. For example, in agent mediated e-commerce, software agents may be engaged in financial transactions in which auction rules may be chosen to suit the features of the participants. It is not obvious to provide a mechanism that induces certain behaviour of the agents in such a scenario. For example, how do we prevent agents from colluding bearing in mind it is even difficult to detect collusion between the agents in the system. We therefore restrict ourselves to models of rationality and behaviour given by equilibrium concepts that are well studied in game theory mechanism design.

There are at least two ways of designing mechanisms automatically. First, we can rely on well-known hand-coded mechanisms to find new ones that satisfy the given system objectives and constraints by solving an optimization problem [3, 15, 23, 24]. Second, we can generate automatically verifiable mechanisms as computer programs in a logical framework.

This preliminary work adopts the latter approach and makes the following contributions:

- it relies on van Benthem's work presented in [22] and makes the connections between mechanism design and process algebra [9] by describing game mechanisms as process models in the modal μ-calculus [8]. This connection permits us to choose a process algebra language, e.g., Promela [7] and to automate the generation of verifiable game mechanisms given some desirable properties of the system.
- it presents an iterative algorithm that generates Promela programs representing mechanisms whose properties can be verified by the SPIN model checker [7] and illustrates the approach using a cake cutting protocol and a single item auction mechanism.

We stress on the importance of producing verifiable mechanisms by AMD since in a setting where agents are self-motivated, claimed mechanism properties must be checked by the participants in order to avoid cheating and to ensure trust in the system.

The remainder of this paper is organized as follows. Section 2 presents a brief introduction to mechanism design. Section 3 describes game mechanisms in a logical game calculus, provides a method in order to decide in two mechanisms are the same, and justifies the use of Promela as a process algebra language for describing games. Section 4 presents an iterative algorithm to generate Promela programs representing game mechanisms from a set of requirements i.e., the mechanism's objectives and constraints. Section 5 discusses the related work and Section 6 concludes.

2 Mechanism Design

Mechanism design, see for example [12, 11] aims to find a decision procedure that determines the outcome for a game according to some desired objective. An objective may be *incentive compatibility* (no agent can benefit from lying provided all other agents are truthful) or *strategy-proofness* (no agent can benefit from lying regardless of what its opponents do). In this section, we present two classes of mechanism we intend to study. The first concerns games with *complete information* (players' utility functions are common knowledge) and for the case of dynamic games we assume *perfect information* (each player knows the history of the game thus far), the second is related to auction design using the class of direct revelation mechanisms, see for example [11] for details.

Definition 1 *A mechanism design problem for n player games of complete and perfect information is defined by*

- *a finite set O of outcomes,*
- *an utility function $\mathbf{u}$ associating each allowed outcome $o \in O$ to a n-vector of real numbers $\mathbf{u}(o)$,*
- *a desirable property of the mechanism, e.g., total utility maximization or incentive compatibility.*

The aim is to find a function that selects outcomes for which the desired property is satisfied.

An example of such a mechanism is the cake cutting protocol wherein we need to share a cake between, say two agents. The protocol consists in asking one agent to cut the cake and let the other pick up its share first. This forces the first acting agent to cut the cake in equal pieces.

Definition 2 *A direct revelation mechanism design problem for n agents is described by*

- *a finite set O of outcomes,*
- *n types or valuation functions $v_i(o \in O), i = 1 \dots n$, private to the agents,*

[1] Department of Computing Science, King's College, University of Aberdeen, Aberdeen AB24 3UE, Scotland, Email: etadjoud@csd.abdn.ac.uk

- *n quasi-linear utility functions $u_i(o) = v_i(o) - p_i$ wherein p_i depends on the course of play for agent i.*

The objective is to find (i) a function f, which, given a vector $\mathbf{v}$ of declared valuations, returns an outcome $f(\mathbf{v}) \in O$ and (ii) a payment function $\mathbf{p}(\mathbf{v}) = (p_1(\mathbf{v}), \ldots, p_n(\mathbf{v}))$ such that reporting its true valuation is a dominant strategy.

An example of such a mechanism is the well-known Vickrey auction and more generally the class of VCG mechanisms, see [11]. For the purpose of generating game mechanisms for a given logical property, we need a closer look at the connections between games, logic, and computer programs.

3 Games: Actions, Outcomes, Utilities and Equilibria

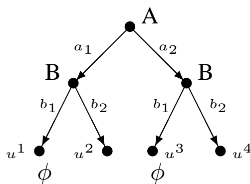
We first confined ourselves to games of complete and perfect information. Such games can be represented in *extensive form* as follows.

Definition 3 *The extensive form representation of a game with perfect and complete information is a tuple $(T, P, (A_i)_{i \in P}$, $turn_{i \in P}$, $move_{i \in P}$, end, $(u_i)_{i \in P})$ wherein*

- *T is a tree composed of an initial node, intermediate or decision nodes, and final nodes with no successors and edges labelled by players' actions,*
- *P is the set of players in the game,*
- *$(turn_i)_{i \in P}$ is a function which marks player i's turn,*
- *$(move_i)_{i \in P}$ represents the set of actions or outgoing transitions for each player i at each decision node,*
- *end is a mark for final nodes,*
- *$(u_i)_{i \in P}$ are the utilities of the players at the final nodes of the tree T.*

Consider a simple game with two players A and B having the set of actions $\{a_1, a_2\}$ and $\{b_1, b_2\}$ respectively as described by Figure 1. Clearly, player B has a strategy forcing the outcome associated with

Figure 1. A two player game



the utility vectors u^1 or u^3. This can be viewed as a partial transition relation for players' turns by using conditionals of the form:

if A plays this, then B should play that.

Thus, the two players interact using strategies involving basic moves and tests for conditions. More generally, a strategy is a plan of actions; it is viewed as a computer program describing alternative sequence of actions that are taken at decision nodes for each player. A sequence of actions can be chosen so as to attain a certain outcome of the game. Each outcome is associated with a value (utility) and each player strives for maximal utility by means of competition or cooperation. In the case of non-cooperative games, the foundational result due to Nash [10] proves the existence of a mixed strategy *Nash equilibrium*, which describes a strategy profile by which no player has an incentive to deviate from it provided all its opponents stick to it. In general, the players have preferences, beliefs, or expectations about the game play and different equilibrium concepts can be found in the literature, see for example [2]. In short, a game is viewed as being composed of basic moves, relational operations such as choice, composition, and conditionals, and utilities involving real number arithmetic.

3.1 A Logical Game Calculus

The above description of a game can be carried out using modal $\mu-$calculus, a modal logic with fixed points introduced in [8]. Starting with the basic moves, the *turn* function and the *end* mark, we can add modal operators such as composition, choice, iteration, or test. Observe that the turn function describes the way players take turns in the game and can be concurrent allowing us for example to describe simultaneous moves in the game. For a given action a and a property ϕ, the modal operator $[a]\phi$ means the execution of a will necessarily make ϕ hold and $< a > \phi$ means the execution of a will possibly make ϕ hold. Obviously, we can have a set of actions in lieu of a single one in the above modal formulae.

Consider for example the game of Figure 1 wherein ϕ is a logical formula involving the players' utilities at the designated end points. Whatever the action of player A, player B has a clear strategy to make property ϕ holds. In modal logic, this is expressed by the formula, wherein $\cup$ represents the choice operator.

$$[a_1 \cup a_2] < b_1 \cup b_2 > \phi$$

We can also define a *winning strategy* for a player i using a recursive predicate $\mathtt{win}_i$. At the terminal nodes $\mathtt{win}_A$ indicates player A wins the game. At any other node, it indicates A has an action which will eventually make him the winner. This gives a recursive definition of the winning strategy. With recursion comes the question of termination. However, the modal μ-calculus provides us with a fixed point iteration as follows. For a given propositional variable p and a formula Φ, the expression $\mu p.\Phi$ represents the least fixed point of the function that maps the set of states where p is true to the set of states where Φ is true. The existence of this least fixed point is ensured by the Knaster-Tarski fixed point theorem [21] provided p occurs positively in Φ.

The winning strategy can then be expressed as follows:

$$\mu\mathtt{win}_A.(\text{end} \wedge \mathtt{win}_A) \vee (\text{turn}_A \wedge < a_1 \cup a_2 > \mathtt{win}_A) \vee (\text{turn}_B \wedge [b_1 \cup b_2]\mathtt{win}_A)$$

Notice that in any finite game, there is a player that has a winning strategy. If for example the strategy profile (a_1, b_2) is a Nash equilibrium, then we have the following formula:

$$\text{end} \wedge (\text{turn}_B \wedge [b_2](u_A^2 \geq u_A^4)) \wedge (\text{turn}_A \wedge [a_1](u_B^2 \geq u_B^1)),$$

meaning that player A $[B]$ gets a higher utility by playing a_1 $[b_2]$ provided B $[A]$ sticks to b_2 $[a_1]$. This logic permits us to reason on the interactions between players, their strategies as well as the game outcomes. Following [1], we can add atomic propositions at all end nodes similar to ϕ encoding preferences or expectations about the course of the game. This is important in mechanism design wherein we are given a desirable property about the course of the game or an objective function to optimize (e.g., social welfare maximisation) and strive to design the game rules so as to achieve those outcomes. At this level of description, a natural question is when are two mechanisms the same?

3.2 Game Mechanism Equivalence

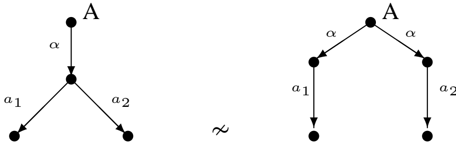
In automated mechanism design, we may be interested in finding out if a newly created game mechanism is the same as a popular one. This is possible if we can compare two given game mechanisms. Because a game is made of processes (composed of basic moves, communication commands such as send or receive a message, relational operations such as choice, composition, iterations or conditionals), we need to compare processes. There is already a large body of literature on equivalences of two processes. Generally speaking, two processes are viewed to be equivalent if an external observer interacting with them cannot distinguish them. This is called *bisimulation* [22].

Let us consider the well-known examples in Figure 2 representing two simple one-player games. If we just care about inputs/outputs the

Figure 2. Two examples of one-player games not observationally equivalent

a) First example

b) Second example



two games are the same. Because in mechanism design, the designer strives to give incentives to the players to behave in a desirable way, we need to have a closer look at the internals of the processes leading to the outcomes. We can consider the execution trace. The two games in Figure 2 a) do not have the same execution trace since in the right hand game, the player makes a move τ to reach a position where it can only performs action a_2. They cannot be equivalent. In Figure 2 b), the two games have the same execution trace $\{\alpha.a_1, \alpha.a_2\}$. They are the same under trace equivalence. However, the two games display different behaviour. In the left hand game of Figure 2 b), after the move α, the player is in a position to perform action a_1 or a_2 whereas in the right hand side, the move α takes the player to a position where it can only performs a_1 or a_2 exclusively. The notion of trace equivalence can be refined so as to account for these internal differences giving rise to the notion of bisimulation. Clearly these two examples display one-player games that are not bisimular. Following [22], we have adopted the stronger notion of bisimulation to define game mechanism equivalence. This can be justified by the fact that we are interested in studying behaviour in a given system. To this end, we need to pay a closer look to the internals of the system in lieu of viewing it as a black box whereby what is important is to ensure the same inputs lead to the same outputs in order to compare two mechanisms.

Considering a game as a bunch of concurrent and communicating processes in a competition or cooperation mode, the question is to find the game rules given a desirable property of the system. For that purpose, a game is generated as a computer program by an iterative procedure in a process modelling language or process algebra so that, at each iteration, the required property can be model checked.

3.3 Process Algebra

Process Algebras (PAs) are mathematical models for concurrent and communicating processes. There are various PAs among them the CCS [9] or Promela [7]. These languages are commonly composed of simple constructs, compositional and operational semantics, behavioural reasoning and observational equivalence. Although, a more generic framework for automatically generating verifiable mechanisms requires dealing with the beliefs of agents (typically in Bayesian games) and therefore the use of a *stochastic* PA [6], the algebra chosen in this paper is Promela [7]. We justify this choice by the fact that Promela is expressive enough for the presentation of our approach and the existence of a well established model checker, SPIN [7] to verify certain properties of the obtained mechanism.

Promela programs are composed of independent and parallel processes communicating through named channels. A process can send or receive a message e trough a channel c by performing actions $[c!e]$ or $[c?e]$ respectively. A process's body is a sequence of operations that can be declarations of typed constants or variables or statements (assignments, conditionals, loops, etc.) made of expressions that manipulate basic algebra terms, see [7] for more details.

The Promela process algebra has a well-defined semantics indicating how processes are executed from the inputs to the outputs in the form of transition system model. This semantics using *structural operational semantics* rules is detailed in [25]. Structural operational semantics rules present a formal way of defining semantics for PAs and other kind of operational semantics. Our aim is to generate well-defined semantics Promela programs representing game mechanisms for a given property of the system.

4 Automatic Generation of Games

This section presents an algorithm to generate Promela programs representing game mechanisms from a set of requirements i.e., the mechanism's objectives and constraints.

4.1 Mechanism Requirements

Mechanism requirements are given as a set of *declarative statements*. A declarative statement describes an identifier (variable or constant) along with its definition, its type, its IO (input or output) status, and its calculation condition.

For simplicity reasons, we use a *choose*$_S$ command, similar to that of [14], which can be modelled in Promela as a non-deterministic choice over the values in the set S. Table 1 gives an example of such requirements. One can see for example that the variable `fp` is an input that is used in the test conditions and a_1, a_2 represent actions taken by the two players. An action is simply a random choice over the set of integers from 0 to 10. Obviously, these requirements mimic constructs of a programming language. However, there is no need to specify the order in which the definitions must be executed; the control logic is left out of the requirements.

Given the requirements of Table 1, we aim to generate the mechanism so that the strategy profile $(a_1{=}5, a_2{=}5)$ is a Nash equilibrium by determining the parameters c_1, c_2 used in the calculation of the utilities u_1, u_2 of both players.

Table 1. Requirements for a design of a game with complete and perfect information

Identifier	Definition	Condition	Type	IO
fp			bit	IN
a_1	choose$_{\{0..10\}}$		byte	
a_2	choose$_{\{0..10\}}$		byte	
u_1	$10 + c_1 a_1$	$fp = 0$	int	OUT
u_2	$10 + c_2 a_2$	$fp = 1$	int	OUT
u_1	$10 - u_2$	$\neg(fp = 0)$	int	OUT
u_2	$10 - u_1$	$\neg(fp = 1)$	int	OUT

Table 2 represents a set of requirements for the design of an auction protocol wherein two agents value an item for sale at v_1 and v_2 and bid the numbers b_1 and b_2 respectively. The numbers b_1, b_2 are chosen in the integer interval $(0, 10)$. The two agents must pay a price p_1 and p_2 given in some parameterised form to get the item with associated utilities u_1 and u_2. The aim is to find the model parameters c_{11}, c_{12}, c_{21}, and c_{22} so that the strategy profile $(b_1{=}v_1, b_2{=}v_2)$ is a dominant strategy equilibrium for both agents.

Table 2. Requirements for a single item auction

Identifier	Definition	Condition	Type	IO
v_1			byte	IN
v_2			byte	IN
b_1	choose$_{\{0..10\}}$		byte	
b_2	choose$_{\{0..10\}}$		byte	
p_1	$c_{11}b_1 + c_{12}b_2$		int	
p_2	$c_{21}b_1 + c_{22}b_2$		int	
u_1	$v_1 - p_1$	$b_1 \geq b_2$	int	OUT
u_2	$v_2 - p_2$	$\neg(b_1 \geq b_2)$	int	OUT
u_1	0	$\neg(b_1 \geq b_2)$	int	OUT
u_2	0	$b_1 \geq b_2$	int	OUT

The first question is how to generate a Promela program given the mechanism requirements? This can be carried out using the following procedure:

1. each declarative statement of the requirements is translated to a Promela instruction.
2. we then construct the data dependency graph between the Promela instructions by analyzing the chain of definitions of the variables and their uses.
3. we find a valid control-flow of the resulting Promela code. It is reasonable to assume that each variable is assigned only once to avoid the case of a variable being overwritten. In this case, the Promela instructions must be reordered so that data dependencies are respected. Namely, we must ensure that no variable is used before being defined. For example, given the requirements in Table 2, the definition of the price p_1 must be executed before that of the utility u_1.

This procedure will enable us to generate a Promela code that has a well-defined semantics and that represents a game mechanism. Figure 3 and Figure 4 show two codes generated from the requirements of Table 1 and Table 2 in which the non Promela `input` command is used for clarity.

The second question is how to choose the parameters specified in the generated code so that the desired properties are satisfied? Notice that the parameter values $c_1{=}c_2{=}-1$ give a solution to the cake cutting protocol design. The values $c_{11}{=}c_{22}{=}0$ and $c_{12}{=}c_{21}{=}1$ give the Vickrey auction mechanism. The values $c_{11}{=}c_{22}{=}c_{12}{=}c_{21}{=}1/2$ give the modified Vickrey auction, which is better in terms of expected revenue for the auctioneer [2]. These parameters can be determined by an iterative procedure.

Figure 3. Mechanism generation for two agents: Cake cutting

```
input(fp);
if
  :: (fp==1 ) ->
     a1 = choose in 0..10;
     u1 = 10+c1*a1;
     u2 = 10-u1;
  :: (fp==2) ->
     a2 = choose in 0..10;
     u2 = 10+c2*a2;
     u1 = 10-u2;
  :: else -> skip;
fi;
```

Figure 4. Mechanism generation for two agents: Single item auction

```
input(v1); input(v2);
b1 = choose in 0..10;
b2 = choose in 0..10;
if
  :: (b1 >= b2) ->
     p1 = c11*b1+c12*b2;
     u1 = v1 - p1;
     u2 = 0;
  :: else ->
     p2 = c21*b1+c22*b2;
     u2 = v2 - p2;
     u1 = 0;
fi;
```

4.2 An Iterative Algorithm

To generate a mechanism from a set of requirements, we generate a computer code that has a well-defined semantics by the following iterative algorithm:

1. Generate a program that respects the definition-use chains in the Promela language as described in Section 4.1,
2. Choose random values for the model parameters,
3. Verify the required property for the obtained mechanism using the language associated model checker,
4. If the property is not satisfied, use a local search method to find new improved model parameters,
5. Repeat Step 2-4 until the property is satisfied or a maximum number of iterations is reached.

By construction, this algorithm always terminates and when it converges, the resulting mechanism satisfies the objective of the system. When the process is stopped because of attaining a maximum number of iterations, the obtained mechanism may be near enough to satisfying the system's objective. The choice of the Promela language is guided by the ability to use the SPIN model checker. This allows us to generate mechanisms that are guaranteed to satisfy the system requirements at convergence. However, using a model checker to verify desirable properties for multi-agent systems can lead to state space explosion for large models. To avoid this, we can use abstraction techniques, see for example [13, 19, 20]. A property-preserving abstraction map can be found so as to transform a detailed concrete domain into a less complex one; rewrite and check the property in the abstract model and deduce its validity in the concrete domain. Finding such an abstraction map is not straightforward, see for example [19].

Although this approach allows us to generate mechanisms guaranteed to have specified desirable properties, there is the scalability problem (how to deal with deal with large-scale models) due to the facts that:

- certain mechanisms may require large number of parameters,
- model checkers may suffer from state-space explosion,
- property preserving abstractions are hard to invent.

Nonetheless, this is a promising approach that deserves further investigation.

5 Related Work

We can only mention some items of the relevant literature. AMD, see for example [3], is a young topic motivated by the need to provide tailored computationally efficient mechanisms given the objectives and constraints of a system. An incremental approach to designing strategy-proof mechanisms is investigated in [4] and a framework for multistage mechanism design is discussed in [17]. In [23, 24], a general AMD framework is discussed. It relies on existing hand-coded mechanisms (e.g., Vickrey auction) to improve the desired objective of the mechanism by an iterative process. In [15], the focus is on improving the pricing rules for an auction mechanism by using evolutionary algorithms. To some extent, these automated mechanisms make some assumptions on the rules of the game and try to fit the objective and constraints by solving an optimization problem. In here, we aim to generate the entire function encoding the game rules and utilities so that we can verify the requirements of the system are satisfied.

Automated mechanisms using process algebra have been developed elsewhere, e.g., music composition [16] or software code generation given its requirements [5]. Our work builds on that of [22] in which extensive games are analyzed as process models using modal logic and bisimulation [22] and extends it to generating mechanisms using an approach similar to that of [5]. Moreover, a research project looking for a logical framework to specify and verify social choice mechanisms is described in [26]. In here, we use a process algebra language to automatically generate verifiable game mechanisms. Techniques for verifying game-theoretic properties of mechanisms are explored in [14, 18] but we adopted the SPIN model checking approach [7] for this study. The use of SPIN in verifying game equilibria is also explored in [19, 20].

6 Concluding Remarks

In this preliminary work, we have used van Benthem's description of games as process models to make the connection between mechanism design and process algebra. This allows us to choose a process algebra language and to automate the production of game mechanisms viewed as computer programs given some desirable properties. In particular, we present a novel algorithm based on this logical framework in order to generate verifiable mechanisms given the objective and constraints of the system. Our logical approach uses an iterative algorithm, which is computationally expensive and therefore requires some restrictions on the space search for the choice of parameters in order to make it feasible. However it can provide us with the entire game mechanism without assumptions on the game rules. On the other hand the numerical optimisation approach [15, 23, 24] can improve the social welfare by solving a fairly large optimisation problem but it assumes the rules of the game.

Future work includes studying the convergence of our algorithm, possibly identifying a class of mechanisms for which the convergence is guaranteed, extending this approach to Bayesian games, and finally applying it to real-life scenarios involving automated mechanism design such as e-commerce systems.

ACKNOWLEDGEMENTS

The author is grateful to Dr Frank Guerin for helpful discussions on game theory and logics. He thanks the reviewers for helpful comments in an early version of this paper. He also acknowledges funding from the UK EPSRC under grant EP/D02949X/1.

REFERENCES

[1] P. Battigalli and G. Bonanno, 'Synchronic information, knowledge and common knowledge in extensive games', *Research in Economics*, **53**, 77–99, (1999).

[2] Ken Binmore, *Fun and Games, A text on Game Theory*, D.C. Heath and Company, 1992.

[3] Vincent Conitzer and Tuomas Sandholm, 'Automated mechanism design for a self-interested designer', in *EC'03*, pp. 232–233. ACM Press, (2003).

[4] Vincent Conitzer and Tuomas Sandholm, 'Incremental mechanism design', in *IJCAI*, pp. 1251–1256, (2007).

[5] Hamido Fujita, Mohamed Mejri, and Béchir Ktari, 'A process algebra to formalize the lyee methodology', *Knowl.-Based Syst*, **17**(5-6), 263–281, (2004).

[6] Peter G. Harrison and B. Strulo, 'Spades - a process algebra for discrete event simulation', *J. Log. Comput.*, **10**(1), 3–42, (2000).

[7] Gerard J. Holzmann, *The SPIN Model checker: Primer and Reference Manual*, Addison, Boston, USA, 2004.

[8] D. Kozen, 'Results on the propositional mu-calculus', *Theoretical Computer Science*, **27**, 333–354, (1983).

[9] R. Milner, *Communication and Concurrency*, Prentice-Hall, 1989.

[10] John Nash, 'Noncooperative games', *Annals of Mathematics*, **54**, 289–295, (1951).

[11] Noam Nisan and Amir Ronen, 'Computationally feasible vcg mechanisms', *JAIR*, **29**, 19–47, (2007).

[12] D. C. Parkes, 'Auction design with costly preference elicitation', *Annals of Mathematics and AI*, **44**, 269–302, (2004).

[13] Corina S. Pasareanu, Matthew B. Dwyer, and Willem Visser, 'Finding feasible abstract counter-examples', *Soft. Tools for Tech. Transfer*, **5**(1), 34–48, (2003).

[14] Marc Pauly, 'Programming and verifying subgame-perfect mechanisms', *J. Log. Comput.*, **15**(3), 295–316, (2005).

[15] Steve Phelps, Peter McBurney, Simon Parsons, and Elizabeth Sklar, 'Applying genetic programming to economic mechanism design: evolving a pricing rule for a continuous double auction', in *AAMAS*, pp. 1096–1097, (2003).

[16] Brian J. Ross, 'A process algebra for stochastic music composition', in *International Computer Music Conference, ICMC 1995*, pp. 448–451, (1995).

[17] Tuomas Sandholm, Vincent Conitzer, and Craig Boutilier, 'Automated design of multistage mechanisms', in *IJCAI*, pp. 1500–1506, (2007).

[18] Emmanuel M. Tadjouddine and Frank Guerin, 'Verifying dominant strategy equilibria in auctions.', in *CEEMAS'07*, volume 4696, pp. 288–297, Leipzig, Germany, (Sep. 2007). LNAI, Springer.

[19] Emmanuel M. Tadjouddine, Frank Guerin, and Wamberto Vasconcelos, 'Abstracting and model-checking strategy-proofness for auction mechanisms', in *DALT*, (2008). accepted.

[20] Emmanuel M. Tadjouddine, Frank Guerin, and Wamberto Vasconcelos, 'Abstractions for model checking game-theoretic properties in auctions', in *AAMAS*, (2008). accepted.

[21] Alfred Tarski, 'A lattice-theoretic fixed point theorem and its applications', *Pacific J Math*, **5**, 285–309, (1955).

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

[23] Yevgeniy Vorobeychik, Christopher Kiekintveld, and Michael P. Wellman, 'Empirical mechanism design: methods, with application to a supply-chain scenario', in *EC'06*, pp. 306–315, New York, NY, USA, (2006). ACM Press.

[24] Yevgeniy Vorobeychik, Daniel M. Reeves, and Michael P. Wellman, 'Automated mechanism design in infinite games of incomplete information: Framework and applications', (2007).

[25] Carsten Weise, 'An incremental formal semantics for Promela', in *Proceedings of the 3rd International SPIN Workshop*, (1997).

[26] Michael Wooldridge, Thomas Ågotnes, Paul E. Dunne, and Wiebe van der Hoek, 'Logic for automated mechanism design - a progress report', in *AAAI*, pp. 9–17. AAAI Press, (2007).