Abstract games for infinite state processes

Perdita Stevens *

Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

Abstract. In this paper we propose finding winning strategies of *abstract games* as an approach to verification problems which permits both a variable level of abstraction and on-the-fly exploration. We describe a generic algorithm which, when instantiated with certain functions specific to the concrete game, computes a winning strategy. We apply this technique to bisimulation and model-checking of value-passing processes, and to timed automata.

1 Introduction

In computer science we frequently answer questions which refer to objects which are infinite, and therefore at first sight intractable, by making a representation of the infinite object which is detailed enough to answer all questions from some class about that object, but which excludes extraneous detail. One proves that the representation is sound, in the sense that the answers obtained by using it are right. Such techniques are used, for example, to answer questions about the equivalence of processes expressed in a value-passing process algebra, or about the equivalence of timed automata, where the source of infiniteness is the fact that clocks may show any real-numbered time. In the latter case [1] showed that we may work with a finite *region automaton*. In the former, an early approach was [10], which dealt with bisimulation of data-independent processes. Later *symbolic transition graphs* (STGs) [9] and then *symbolic transition graphs with assignment* (STGAs) [11] were introduced to handle a larger class of value-passing processes. This representation has been used to answer bisimulation questions and also model-checking questions [13].

A natural disadvantage of an approach which attempts to construct a finite representation suitable for answering *any* question from a large class about a given object is that it fails to take advantage of the easiness of particular questions. A question may be easier than another in two ways: it may be answerable using a coarser representation, and/or without needing information about the whole of the representation. Both ways may allow us to answer some questions about infinite objects even when there is no such finite representation suitable for answering all questions about the object. On-the-fly algorithms address the second; the first is usually addressed implicitly if at all.

* Perdita.Stevens@dcs.ed.ac.uk, supported by EPSRC GR/K68547

The problems mentioned above (and many others, including many not obviously game-like ones, such as trace equivalence of processes) can be characterised by two-player games: the answer to the question corresponds to the player who has a winning strategy. The winning strategy itself can be seen as a proof object for the question, and a tool can use the strategy to provide feedback to the user [14]. Given such a characteristic game, we show how to define an *abstract game* which is a sound abstraction of it (which in this framework means that winning strategies for the two games correspond). We give a generic algorithm which, for a large class of such games, finds a winning strategy: that is, a correct answer to the original question together with a proof object. This fully automatic, on-the-fly algorithm works with a variable level of abstraction (cf widening, narrowing in the algebraic interpretation framework of [7], or in the particular case of model-checking, the external choice of widening in [2]). Since the questions are in general undecidable, there are of course still intractable cases in which our algorithm fails to terminate; but we can answer many "easy" questions outside known decidable classes.

We have a practical motivation, which is that we want the Edinburgh Concurrency Workbench to be able to answer a wide range of equivalence and model-checking questions about processes expressed in a range of process algebras, including value-passing and real-time process algebras. We need to minimise the effort required to permit a new class of questions: using this work, we only have to show how the question's game fits into the framework, and can then use the generic algorithm to answer the question. Of course the generic algorithm cannot take advantage of the structure of the particular game to improve its efficiency, so in particular cases we may want to implement more efficient versions. Our future plans include investigating classes of games for which efficiency improvements are possible, but the present paper is concerned not with efficiency, but with generality.

In Section 2 we informally present an example of playing an abstract game. Section 3 gives definitions and states the correspondence between strategies for a concrete game and for an abstract version of it; in fact because of space limitations this section presents only a special case, sufficient to support the work reported here, of a more general definition which can be found in the long version of this paper. Section 4 describes an algorithm for finding winning strategies for a large class of abstract games. Section 5 shows how the work applies to three classes of problems: bisimulation and mu-calculus model-checking of value-passing processes, and bisimulation of timed automata. Finally we present our conclusions and plans for future work.

2 Informal example

The *bisimulation game* starts with two processes. A process could be anything with LTS semantics, but here we will suppose for definiteness that these are processes of standard value-passing CCS ([12], or see [15]). The game has positions of two sorts. Positions from which Abelard (Player I, Opponent, etc. – the one who wants the answer to be No) must move are pairs (E, F) of processes.

The initial position has this form, so Abelard starts. Abelard must choose a transition from one of the processes, and Eloise must then choose a matching transition (i.e. one with the same action) from the other process. Thus a position from which Eloise must move is (E, F, a, b) where E and F are processes, b is 1 if Eloise must match from E, otherwise 2, and a is the action of the transition Abelard just chose, which Eloise must match from the other process. A player wins if the other player has no available move – that is, Eloise wins if play reaches (E, F) where neither E nor F have any transitions, and Abelard wins if Eloise is unable to match his transition, for example if the position is $(E, F, a, 1)$ and $E \stackrel{a}{\not\longrightarrow}$. Eloise wins every infinite play. It is easy to see that Eloise has a winning strategy for the game starting at (E, F) iff E and F are (strongly, assuming we play with strong transitions) bisimilar.

When the transition systems represent value-passing CCS processes, they will contain structure of which we may be able to take advantage when we look for a winning strategy. There will be sets of plays which differ only in the values of the data variables in the CCS derivatives. To avoid exploring more of these plays than we must, we may play with a *set game*. The intuition is that we modify the standard game by allowing the players to postpone decisions about exactly which of such a set of plays is being followed: play so far may represent many possible plays of the concrete game. When a player chooses a move, s/he is permitted to restrict the set of plays which should be considered from here on, provided that this set remains non-empty. That is, s/he is permitted to impose a satisfiable constraint on the data which is currently active. Winning conditions are as for the standard game (in a sense to be made precise in Section 3).

For example, consider games intended to establish whether the CCS processes $B = in(x).\overline{out}(x).B$ and $C = in(x).\overline{out}(5).C$ are bisimilar. Obviously Abelard has a winning strategy for the basic bisimulation game. For example, from the initial position (B, C) he could pick the transition $B \stackrel{in(7)}{\longrightarrow} \overline{out}(7).B$ giving the new position $(\overline{out}(7).B, C, in(7), 2)$ meaning that Eloise must pick a $in(7)$ transition from agent 2, i.e. C. She will pick $C \stackrel{in(7)}{\longrightarrow} \overline{out}(5).C$, yielding position $(\overline{out}(7).B, \overline{out}(5).C)$. Whichever transition Abelard chooses now she will be unable to match, so Abelard will win. Of course, had Abelard made the mistake of picking $B \stackrel{in(5)}{\longrightarrow} \overline{out}(5).B$, Eloise would have been able to match this move.

The corresponding *set game* begins at the singleton $\{(B, C)\}$ and Abelard might move to position $\{(\overline{out}(v).B, C, in(v), 2) : v \neq 5\}$. Eloise matches by moving to position $\{(\overline{out}(v).B, \overline{out}(5).C) : v \neq 5\}$. (She could also choose to impose a further constraint on v.) Again, Eloise will be unable to match Abelard's next move, whatever it is. Had Abelard made the mistake of not imposing the constraint $v \neq 5$ at his first move – for example, had he moved to the unrestricted position $\{(\overline{out}(v).B, C, in(v), 2)\}$ – Eloise could have matched by moving to $\{(\overline{out}(v).B, \overline{out}(5).C) : v = 5\}$. She cannot impose $v = 5$ given that Abelard has already imposed $v \neq 5$, because the resulting constraint would not be satisfiable: the set of concrete positions would be empty.

3 Set games: an example of abstract games

In the long version of this paper we make a general definition of a *sound abstraction* of a game in terms of a partial order on positions and a set of axioms, prove strategy transfer theorems based on these axioms, and then demonstrate that the *set games* fit the framework. Here, due to pressure of space, we will simply present the set games and state the theorems about them alone.

For the purposes of this paper, a game is always played between two players Abelard (abbrev. $\forall$) and Eloise ($\exists$). We refer to players A and B to mean Abelard and Eloise in either order. A game G is $(Pos, I, moves, \lambda, W_\forall, W_\exists)$ where:

- Pos is a set of positions. We use $u, v, \ldots$ for positions.
- $I \subseteq Pos$ is a set of starting positions.
- $moves \subseteq Pos \times Pos$ defines which moves are legal. A play is in the obvious way a finite or infinite sequence of positions starting at some $p_0 \in I$ where $p_{j+1} \in moves(p_j)$ for each j. We write p_{ij} for $p_i \ldots p_j$.
- $\lambda : Pos \to \{\text{Abelard}, \text{Eloise}\}$ defines who moves from each position.
- $W_\forall, W_\exists \subseteq Pos^\omega$ are disjoint sets of infinite plays, and W_A includes every infinite play p such that there exists some i such that for all $k > i$, $\lambda(p_k) = B$.

Player A *wins* a play p if either $p = p_{0n}$ and $\lambda(p_n) = B$ and $moves(p_n) = \emptyset$ (you win if your opponent can't go), or else p is infinite and in W_A. (Notice that some infinite play may have no winner: such a play is said to be a draw.)

Remark 1. When games are considered in logic it is normally assumed that all non-extensible plays are infinite, that the players take turns and that every play is eventually won. We relaxed these restrictions in order to fit naturally with the usual formulations of both bisimulation and model-checking games, but the relaxations are not theoretically significant, given the condition on $W_\forall$, $W_\exists$.

A (nondeterministic) strategy S for player A is a partial function from finite plays pu with $\lambda(u) = A$ to sets of positions (singletons, for deterministic strategies), such that $S(pu) \subseteq moves(u)$ (that is, a strategy may only prescribe legal moves). A play q follows S if whenever p_{0n} is a proper finite prefix of q with $\lambda(p_n) = A$ then $p_{n+1} \in S(p_{0n})$. Thus an infinite play follows S whenever every finite prefix of it does. It will be convenient to identify a strategy with the set of plays following the strategy and to write $p \in S$ for p follows S. S is a *complete* strategy for Player A if whenever $p_{0n} \in S$ and $\lambda(p_n) = A$ then $S(p_{0n}) \neq \emptyset$. It is a *winning* strategy for A if it is complete and every $p \in S$ is either finite and extensible or is won by A. It is *non-losing* if it is complete and no $p \in S$ is won by B. It is *history-free* (or *memoryless*) if $S(pu) = S(qu)$ for any plays pu and qu with a common last position. A game is *determined* if one player has a winning strategy.

Given a game $G^{\mathcal{C}}$ (the *concrete game*) in which $W_\forall \cup W_\exists = (Pos^{\mathcal{C}})^\omega$ (no draws), define an abstract game $G^{\mathcal{A}}$ (the *set game* corresponding to $G^{\mathcal{C}}$) by:

- $Pos^{\mathcal{A}} = \{U \in \mathcal{P}(Pos^{\mathcal{C}}) \setminus \{\emptyset\} : u, v \in U \Rightarrow \lambda^{\mathcal{C}}(u) = \lambda^{\mathcal{C}}(v)\}$. Thus set-game positions are non-empty sets of concrete positions: we use $U, V \ldots, P, Q, \ldots$, for set-game positions and plays. If the concrete play $p = (p_j)_{j \in J}$ and the set-game play $P = (P_j)_{j \in J}$ have the same length (finite or infinite), we write $p \leq P$ for $\forall j \in J \,.\, p_j \in P_j$, and we say P subsumes p.

- $I^{\mathcal{A}} = \mathcal{P}(I^{\mathcal{C}}) \setminus \{\emptyset\}$
- $V \in \mathit{moves}^{\mathcal{A}}(U)$ iff for each $u \in U$, $\mathit{moves}^{\mathcal{C}}(u) \neq \emptyset$ and $V \subseteq \bigcup_{u \in U} \mathit{moves}^{\mathcal{C}}(u)$
- $\lambda^{\mathcal{A}}(U) = \lambda^{\mathcal{C}}(u)$ where $u \in U$ (well-defined by definition of $\mathit{Pos}^{\mathcal{A}}$)
- $P = (P_i)_{i \in \omega} \in W_A^{\mathcal{A}}$ iff either for all but finitely many i, $\lambda^{\mathcal{A}}(P_i) = B$, or both
  1. $\exists p = (p_i)_{i \in \omega} \in W_A^{\mathcal{C}}$ s.t. $p \leq P$ (P subsumes some A-won concrete play)
  2. $\forall p = (p_i)_{i \in \omega} \in W_B^{\mathcal{C}} \quad p \not\leq P$ (P subsumes no B-won concrete play)

  (An infinite play which subsumes no concrete play is drawn.)

We may omit the superscripts $\mathcal{A}$ and $\mathcal{C}$ when they are obvious from context.

Let S be a strategy for $G^{\mathcal{C}}$. Construct a strategy $\alpha(S)$ for $G^{\mathcal{A}}$ by

$$(\alpha(S))(P) = \{\{u\} : \exists p \leq P \ s.t. \ u \in S(p)\}$$

Conversely, let S be a strategy for $G^{\mathcal{A}}$. Construct a strategy $\gamma(S)$ for $G^{\mathcal{C}}$ by

$$(\gamma(S))(p) = \{u : \exists PU \in S \ s.t. \ p \leq P \text{ and } u \in U \cap \mathit{moves}^{\mathcal{C}}(p_n)\}$$

where $p = p_0 \ldots p_n$.

We get from α and γ (which form a Galois connection) a correspondence between strategies for a concrete game and for a sound abstraction of it:

Theorem 1. *1. If S is a winning strategy for $G^{\mathcal{C}}$ then $\alpha(S)$ is a winning strategy for $G^{\mathcal{A}}$.*
2. If T is a downwards closed non-losing strategy for $G^{\mathcal{A}}$, then $\gamma(T)$ is a winning strategy for $G^{\mathcal{C}}$.
3. If there is a winning strategy for player A for $G^{\mathcal{A}}$ then there is a downward closed winning strategy for player A for $G^{\mathcal{A}}$.

(Downward closure is a technical condition on strategies which will hold for the strategies we construct.)

Using the observation that the constructions preserve history-freeness we get

Lemma 1. *1. $G^{\mathcal{A}}$ is determined iff $G^{\mathcal{C}}$ is determined.*
2. $G^{\mathcal{A}}$ has a downward closed non-losing history-free winning A strategy iff $G^{\mathcal{C}}$ has a history-free winning A strategy.

3.1 Shapes and constraints

In order to work with set games we need a way to represent the sets of concrete positions that arise. For the remainder of this paper we shall assume that the concrete game can be given a *notion of shape*, defined as follows. There is a set of shapes; shapes may be parameterised on data values, clock times, etc. Each concrete position can be uniquely represented by giving its shape and values for the parameters. If concrete positions u and v have the same shape (we write $u \approx v$) then $\lambda(u) = \lambda(v)$. We write (s, c) for the set-game position comprising the concrete positions with shape s and parameter values satisfying the *constraint* c; we will only need to consider such homogeneous set-game positions. Thus the intersection of two positions (s, c), (t, d) is empty unless $s = t$ and in that case is $(s, c \wedge d)$. We insist that if p and q are infinite (concrete) plays with the same shape (that is, for each i, $p_i \approx q_i$) then $p \in W_A$ iff $q \in W_A$. We assume that the shape-graph given by $s \to t$ if there is some legal move $(s,c) \to (t,d)$ is finite branching (so that our function for calculating the next moves from a position may terminate). We do not insist that the shape-graph be finite, though of course a terminating run of the algorithm will only visit finitely many nodes of it.

For example, the shape of an Abelard-choice position in a bisimulation game on value-passing process algebra will represent a pair of process terms with named holes for data, and the constraint will be a formula with some of those names free. In a model-checking game on value-passing processes it will be a pair of such a process term with a subformula of the formula being checked. In a bisimulation game on timed automata, the shape will be a pair of automata states and the constraint will restrict the possible values of the clocks.

We are interested in finite representations of strategies for the concrete game, but strategies for the set game are even bigger than those. However, we can define the strategy generated by a finite set of rules, and then work with the finite description. Let B be a set of pairs of sets where for each $(U,V) \in B$ we have $\forall u \in U\ \exists v \in V.v \in \mathit{moves}(u)$ and $\forall v \in V\ \exists u \in U.v \in \mathit{moves}(u)$. The *strategy generated by* B defines possible moves from U' to be the non-empty subsets of $\mathit{moves}(U \cap U') \cap V$ for any pair $(U,V) \in B$ with $U' \cap U \neq \emptyset$.

4 Algorithmics

In this section we begin to explore how winning strategies for a set game, and hence for the underlying concrete game, can be calculated. We give a generic algorithm which, when instantiated with certain functions to describe a specific concrete game, searches for winning strategies for that game. We give an informal description, together with a summary figure and a specification of the functions that must be provided to instantiate the algorithm; we omit some technical details and proofs.

From here on we restrict attention to determined games which have history-free winning strategies, and we assume that the (concrete) winning sets can be characterised by shape sequences, in the sense that given any "loop" of shapes $l = s_0 s_1 \dots s_0$ we can allocate l to a player, calling it an Abelard- or Eloise-loop, such that W_A comprises the legal plays whose shapes are infinite compositions of A-loops. Moreover we assume we can define $\leq_A$ on shape segments with common endpoints, in the sense that $s = xs_1 \dots . s_n y \leq_A t = xt_1 \dots t_m y$ iff for any segments a, b, atb is an A-loop whenever asb is, and that for any two segments at least one of $s \leq_A t$ ("t is at least as good as s for A"), $s \leq_B t$ holds. This apparently strong condition is satisfied by the examples we have in mind: bisimulation, where every loop is an Eloise loop and $\leq_\forall$, $\leq_\exists$ are both always true, and model-checking, where a loop belongs to Eloise iff the outermost fixpoint unwound on it is maximal, and where $s \leq_\exists t$ is true unless both some ν-variable is active throughout s and is the outermost such variable, and some μ-variable which subsumes it is active throughout t.

To instantiate the algorithm To use the algorithm to find a winning strategy for the set-game version of a suitable concrete game, one needs to define a notion of shape which satisfies the conditions in Section 3.1. In addition we need to ensure that the constraint language is expressive and tractable enough to compute the sets that the algorithm works with. We must provide implementations of:

1. the boolean operations, to express the combinations of sets that arise
2. $\mathsf{satisfiable}$, a function to check emptiness of a set
3. $\mathsf{getMaximalMove}$: a function which given positions U, V, returns the maximal $V' \subseteq V$ such that $V' \in moves(U)$, i.e. $\{v \in V : \exists u \in U \; v \in moves(u)\}$.
4. $\mathsf{getMaximalPredecessor}$: a function which given positions U and $V \in moves(U)$, returns $\{u \in U : V \cap moves(u) \neq \emptyset\}$.
5. $\mathsf{maximalMoves}$: that is, a function which given a position $U = (s, c)$ returns $((s_1, c_1), d_1) \ldots ((s_n, c_n), d_n))$ where:
   – for each shape s_i, (s_i, c_i) is the maximal move of that shape legal after U: that is, $(s_i, c_i) = \mathsf{getMaximalMove}(U, S_i)$ where S_i consists of all concrete positions of shape s_i.
   – each (s_i, c_i) is non-empty, i.e. we only mention the shapes that can actually occur
   – $(s, d_i) = \mathsf{getMaximalPredecessor}((s, c), (s_i, c_i))$.
6. winner: that is, a function which, given a loop P_{ij} of a set-game play, extracts a loop s_{ij} of shapes and returns the player A for whom this is A-loop.
7. $\leq_A$ for each player A, being the order on common-ended shape segments mentioned above.

(In the CWB the algorithm will be implemented in an ML functor with an argument matching a signature describing these types and values). Of course we may be satisfied with implementations that may not terminate, if we are prepared to accept this extra source of non-termination of the algorithm.

Input and output, data structures Our algorithm takes as input a single initial position $i = (s, c)$ of the set game, representing a set of positions of the concrete game with a common shape. If and when it terminates, it provides a partition of i into $(s, c_\forall)$, $(s, c_\exists)$, together with a set of rules generating a downwards-closed non-losing A strategy σ_A for the set game starting at (s, c_A) ($A \in \{\forall, \exists\}$). By Theorem 1 (and downwards-closedness) $\gamma(\sigma_A)$ is a winning strategy for the concrete game starting at (any of) the concrete positions in (s, c_A), so the algorithm specifies which parts of the original set of concrete positions can be won by each player. (Of course the set game starting at i itself won by player $\lambda(i)$, if both members of the partition are non-empty, otherwise by the owner of the non-empty member.) The algorithm always works with same-shape set positions: since by Theorem 1 any determined game with a history-free winning strategy has a strategy which only prescribes singletons, a strategy which beats all same-shape opposing strategies is a winning strategy, so this suffices.

The algorithm maintains a *playList* which is a sequence of *nodes* leading from a node recording the initial position i to the *current node*. Information in a node is updated as the algorithm proceeds.

Definition 1. *A* node n *records information most of which may be updated:*

- *a position* (s, c) *(immutable)*
- *a timestamp* creation *saying when this node was created (immutable)*
- *constraints* $(ass_\forall, ass_\exists)$ *recording which subsets have been repeated in (we say used as* assumptions *by) nodes below here.*
- *constraints* $(won_\forall, won_\exists)$ *describing for which subsets of the node we have a winning strategy (subject to assumptions strictly higher up playList).*
- *a list unexplored of maximal moves yet to be explored*
- *a constraint* prov *which specifies for what subset of an A-choice node we have explored an A move that is provisionally (subject to assumptions both here and higher up playList) good.*

We write informally $n.c$ *for the constraint in node* n*, etc. For convenience we write* $n.chooser$ *for* $\lambda(n.s, n.c)$*, and* $n.unwon$ *for* $n.c \wedge \neg(n.won_\forall \vee n.won_\exists)$.

Definition 2. *A* decision d *records (immutably, though the whole decision may have to be deleted ("forgotten")):*

- *a player* A *for whom this is a decision*
- *a position* (s, c)
- *a strategy rule* $(s, c) \mapsto (t, d)$ *if* $\lambda(s) = A$
- *a timestamp* creation *when this decision was added*
- *a sequence of shapes* (s_i).

The algorithm maintains a set Δ of decisions, adding and deleting decisions as the algorithm proceeds. The final A strategy σ_A is that generated by the rules in the A-decisions which are in Δ when the algorithm terminates. Decisions allow reuse of some previous calculation: a function applies (implemented using the game-specific implementation of $\leq_A$) takes a decision d and a *playList* and returns **ff** unless *playList* ends with a node whose position is (s', c') where (a) $s' = d.s$ and (b) $dec = c' \wedge d.c$ is satisfiable and (c) the shape-sequence (s_i) recorded in d is *compatible* with the sequence (t_i) of shapes of nodes in *playList*. Let $(s_i)_{i \leq n} = (t_i)_{i \leq n}$ be the longest common prefix of (s_i), (t_i). Then d is compatible with *playList* iff $(s_i)_{i>n} \leq_A (t_i)_{i>n}$.

Overview The algorithm alternately explores the set game graph and backtracks. A counter ("time", current time returned by now()) is incremented at each step. We explore depth first, maintaining the sequence *playList* of nodes leading from the root to the *current node* (empty if the current node *is* the root). When we explore to a new set-game position (s, c) we create a new node u with position (s, c), *creation* now() all other constraints **ff**, and *unexplored* set to maximalMoves(s, c). Next we consider whether we can allocate any parts of the position to players without doing any exploration.

- If $\lambda(s, c) = A$ the constraint describing any subset of (s, c) from which A has no legal move ($c \wedge \neg(\bigvee d_i)$ where the d_i are as returned by maximalMoves) is disjoined with won_B.
- Any applicable decisions (as returned by applies) are disjoined with the relevant won_A.

– We look for repeats: if s is also the shape of a node v higher up in *playList* and satisfiable($c \wedge v.unwon$), we let A be the player for whom the sequence of shapes between v and u is an A-loop (using winner). $c \wedge v.unwon$ is disjoined with $v.ass_A$ and with $u.won_A$: we say this part is won by A subject to the assumptions at v.

Unless we've exhausted the position ($unwon \Rightarrow prov$, which in the case of a newly created node can only happen if *unwon* is unsatisfiable) or the possible moves, we pick a maximal move $((s_i, c_i), d_i)$ from *unexplored* and try that, creating a new node for position getMaximalMove($(s, unwon \wedge \neg prov), (s_i, c_i)$).

Once we've run out of moves or provisionally allocated the whole position, we retrace our steps along the *playList*, notionally trying to build winning strategies for both players. As we backtrack, of course, we remove entries from the end of the *playList*, so the *playList* records a "straight" path from the root to the current node. When we backtrack from node m with position $(m.s, m.c)$ to a B-choice-point n with position $(n.s, n.c)$, we consider how B's choice at n may take advantage of the $m.won_B$ (if at all: of course $m.won_B$ may be unsatisfiable). We have found a provisional good move for a position in n provided B can move from it to a known-good position in m. Let $(n.s, d)$ be getMaximalPredecessor($(n.s, n.unwon), (m.s, m.won_B)$). We record a decision for B at $(n.s, d)$ with strategy rule $(n.s, d) \mapsto (m.s, m.won_B)$, timestamp now, and sequence of shapes the shapes of the current *playList*. We also disjoin d with $n.prov$.

If we still have an unallocated part of $(n.s, n.c)$ and an unexplored move, we pick a new maximal move and explore it on the unallocated part as described above. Once we exhaust either the position or the moves from it, we must consider whether the assumptions at this n, recorded in $(n.ass_\forall, n.ass_\exists)$, have been confirmed or invalidated. (This stage is only required when we exhaust a node after doing some exploration from it, i.e. we discover in backtrack that we have exhausted it. If we exhaust the node without exploring any move from it, as when in explore, there can be no assumptions at this node, so there is no need to examine them.) If we ran out of untried B-moves with satisfiable($n.unwon \wedge \neg n.prov$), we provisionally allocate this left-over to A (provisionally A has a defence against every B move). We now have a partition of the position into parts won by each player and provisionally allocated to each player: say $c \Leftrightarrow (won_\forall \vee won_\exists \vee prov_\forall \vee prov_\exists)$. Now dischargeOrInvalidate records how we examine assumptions. Player B's assumptions are *safe* if $n.ass_B \Rightarrow n.prov_B$: in this case we *discharge* them and disjoin $n.prov_B$ with $n.won_B$. Similarly for A. If a player's assumptions are unsafe we *invalidate* them, i.e. forget any decisions which may have rested on them; the simple way to do this is by timestamps, forgetting all decisions d for that player which were added after the creation of n, i.e. which have $d.creation$ later than $n.creation$. If after dischargeOrInvalidate returns, $n.c \Leftrightarrow n.won_\forall \vee n.won_\exists$, we may backtrack from n. Otherwise we must reexplore the part(s) which are still only provisionally allocated. The function iterate takes the node with unconfirmed provisional parts, reexplores from each separately, and recombines the results of the two explorations.

fun explore (s, c) *playList backtrackingList* =
create a new node n with position (s, c) appropriately initialised
allocate bits with no moves, look for applicable decisions, look for repeats
if (haven't exhausted position and there is a maximal move (s_i, c_i))
then (explore (s_i, c_i) *(n::playList) (n::backtrackingList)*)
else (backtrack n *playList backtrackingList*)

fun backtrack m *playList backtrackingList*
if (*backtrackingList* = [])
then return m [it contains the answer]
else it's $(n :: t)$.
  $A := n.chooser$
  Let $(n.s, d)$ be getMaximalPredecessor$((n.s, n.unwon), (m.s, m.won_A))$.
  Add a decision for A at $(n.s, d)$ with rule $(n.s, d) \to (m.s, m.won_A)$, timestamp now,
  shapes the shapes on playList.
  if satisfiable$(n.unwon \wedge \neg n.prov)$
    and $n.chooser$ has more unexplored moves
  then [we've exhausted neither the position nor the moves: keep going]
    pick an unexplored move (s_i, c_i)
    explore (s_i, c_i) *playList backtrackingList*
  else
    dischargeOrInvalidate n
    if satisfiable$(n.unwon)$ then iterate n *(tl playList)* t
    backtrack n *(tl playList)* t

fun iterate n *playList backtrackingList*
foreach $A \in \{\forall, \exists\}$ [reexplore any unconfirmed A-wins]
  $unconfirmed_A :=$ if $A = n.chooser$ then $n.prov \wedge \neg n.won_A$
    else $n.unwon \wedge \neg n.prov$
  if satisfiable$(unconfirmed_A)$
  then $n_A :=$ explore $(n.s, unconfirmed_A)$ *playList* []
[finally put the node back together again]
foreach $A \in \{\forall, \exists\}$
  $n.won_A := n.won_A \vee n_\exists.won_A \vee n_\forall.won_A$ (taking $m.won_A = \mathbf{ff}$ if m undefined)

fun dischargeOrInvalidate n
foreach $A \in \{\forall, \exists\}$
  $prov_A :=$ if $A = n.chooser$ then $n.prov$ else $n.unwon \wedge \neg n.prov$
  if satisfiable $(n.ass_A \wedge \neg n.prov_A)$ [i.e. if any assumption has not been supported]
  then [must invalidate these A assumptions and any decisions that rest on them]
    forget all A decisions timestamped later than $n.creation$
  else[discharge A assumptions, confirm things provisionally decided for A]
    $n.won_A := n.won_A \vee n.prov_A$
  $n.ass_A := \mathbf{ff}$

Fig. 1. Summary of the algorithm

Theorem 2. *If, when instantiated according to the specification, the algorithm run on* (s, c) *terminates and returns a node* n *with satisfiable* $n.won_\forall$*, then the Abelard decisions in* Δ *generate a downwards closed history-free non-losing Abelard strategy for the set game starting at* $(s, n.won_\forall)$*. By restriction to ground plays* (γ) *they can also be regarded as generating a winning strategy for Abelard for the concrete game starting at any concrete position* $u \in (s, n.won_\forall)$*. Similarly for Eloise and* $(s, n.won_\exists)$*.*

The proof uses an adaptation of the open game technique described in [14] (and rather more of the machinery of abstract games than we have presented here). In summary, we define *open games* which are games relativised to a set of assumptions. These allow us to consider finite-length portions of infinite plays, by stopping on hitting an assumption. We show that the algorithm maintains the invariant that σ_A, the strategy generated by the current A-decisions in Δ, is a non-losing strategy for the open game $G(\Gamma, d)$ where Γ is the current set of A-assumptions and $d \in \Delta$ is an A-decision. If the algorithm terminates, it does so with Γ empty, yielding a non-losing strategy for the original set game.

Termination The algorithm fails to terminate if explore is called infinitely often, that is, if infinitely many nodes are created. This may happens if it has to explore nodes with infinitely many shapes (as in certain cases of checking bisimulation of processes whose definitions involve recursion under a parallel operator) or if infinitely many nodes having the same shape are created. In the latter case, because of the way we deal with repeated subsets, the nodes must have infinitely many different set-game positions: we never create infinitely many nodes with the same position.

To see when we could create infinitely many different nodes with the same shape, notice that the algorithm itself only performs Boolean operations on sets (equivalently, on constraints) so the complication comes from what the game-specific functions getMaximalMove and getMaximalPredecessor do. It is straightforward to prove:

Proposition 1. *Suppose we have a notion of shape as defined in 3.1 for a concrete game* $G^{\mathcal{C}}$*. Let* $\sim$ *be an equivalence relation on* $Pos^{\mathcal{C}}$*, a refinement of* $\approx$ *("is the same shape as"), such that:*

1. $u \sim v \Rightarrow u \approx v$ *(so it is a refinement)*
2. *if the arguments to getMaximalMove or getMaximalPredecessor are unions of whole* $\sim$*-equivalence classes, then so are the results of those functions.*
3. *there are finitely many* $\sim$*-equivalence classes of concrete positions.*

(For example, if $\approx$ *itself satisfies 2 and 3 we may take that as* $\sim$*.) Then it is decidable who has a winning strategy for any such concrete game, since the algorithm terminates when started on any union of whole* $\sim$*-equivalence classes.*

5 Examples

5.1 Bisimulation games on value-passing processes

A bisimulation set game position is $((E, F), c)$ or $((E, F, a, i), c)$ where (E, F) and (E, F, a, i) are shapes, which may involve *parameters* for data items or for actual parameters of parameterised agents, and the free variables of c are drawn from among the parameters of the shape. Parameters are not symbols whose names are significant: they serve only to define a set. (Recall our example $\{(\overline{out}(v).B, \overline{out}(5).C) : v \neq 5\}$ in Section 2: here $\{s : c\}$ is alternative notation for (s, c).) Accordingly, for convenience, we will adopt the convention in our notation for sets that parameters of the first and second agents in a position are named $p_{11}, \ldots p_{1n}$ and $p_{21}, \ldots, p_{2m}$ in their order of appearance from left to right and that a parameter representing a new datum being read in is called p_a.

Following our definition of the moves in a set game, we see that from $u = \{(E, F) : c\}$ Abelard may choose a non-empty position of the form $\{(E, F', a, 1) : d\}$ or $\{(E', F, a, 2) : d\}$ provided every concrete position in it is the position resulting in the concrete game from some concrete position in u. For example, if he chooses $v = \{(E, F', a, 1) : d\}$ it must be the case that

$$\forall (E_c, F'_c, a_c, 1) \in v\ \exists F_c\ .\ (E_c, F_c) \in u \wedge F_c \xrightarrow{a_c} F'_c.$$

Similarly, Eloise's moves from $v = \{(E, F', a, 1) : d\}$ are of the form $w = \{(E', F') : e\}$ where w is non-empty and

$$\forall (E'_c, F'_c) \in w\ \exists (E_c, F'_c, a_c, 1) \in v\ .\ E_c \xrightarrow{a_c} E'_c$$

(The fact that Eloise has the opportunity to refine the constraint – that is, to choose a position which does not contain matches for every position in the previous set – reflects the fact that different matching moves may be required, depending on the actual values.)

It should be clear that the required functions are implementable for "any reasonable" value-passing process algebra and data language. We do not have space to describe all the details, but as an example let us consider how to calculate the maximal same-shape moves for Abelard from $\{(P, Q) : c\}$, where P and Q are process terms involving parameters $p_1, \ldots p_n$ and $q_1 \ldots q_m$ respectively and c is a constraint with free variables some of $p_1 \ldots p_n, q_1 \ldots q_m$; the position of course represents a set of pairs of processes.

We can assume a transition function which, given the process term P, returns a list $(a_1, P_1, e_1), \ldots (a_l, P_l, e_l)$ where each a_j is an action term (possibly involving some of the p_i, or possibly involving a single new parameter p_a, if the action involved the inputting of a new datum), each P_j is a process term (which can involve any of the p_i and possibly the p_a) and e_j is a constraint in the p_i and p_a, such that if values v_i, v_a are substituted for the parameters, the transition $P[v/p] \xrightarrow{a_j[v/p]} P_j[v/p]$ exists iff $v \models e_j$.

Pick some (a_j, P_j, e_j). The corresponding maximal new position in the set game is got by:

1. taking $\{(P_j, Q, a_j, 2), d\}$ where d is the result of taking $e_j \wedge c$ and existentially quantifying over any parameter that does not appear in $(P_j, Q, a_j, 2)$.
2. normalising the representation of the set by renaming the bound variables and the parameters and simplifying the constraint. (Simple-minded approach: replace each parameter p_i by a variable p'_i; add conjuncts specifying each p_i in terms of the p'_j; existentially quantify over all p'_i; simplify.) For example, in the example below we elide the normalisation of

$$((M_1(p_{11}+p_a), r(v).N_1(p_{21}+|v|), r(p_a), 2), p_{11} = p_{21} \wedge p_a > 0)$$

to

$$((M_1(p_{11}), r(v).N_1(p_{21}+|v|), r(p_a), 2), p_{11} = p_{21} + p_a \wedge p_a > 0).$$

If the moves resulting from several transitions have the same shape, of course they can be combined by taking the disjunction of their constraints.

Remark 2. Because, following sources such as [15], we have used semantics in which the action of reading a value from a channel is atomic, the bisimulation we get by this definition is early bisimulation. The finer equivalence relation known as late bisimulation, in contrast, regards input as a two-stage procedure: first we commit to reading from a certain channel and proceeding with a certain continuation, then we read the value. To define the game characteristic for late bisimulation: let our set of actions include the special actions $c()$ for each input channel, representing "commit to reading from channel c but don't actually read a value" and $\epsilon(v)$ representing the reading of a value v from the committed channel. Thus: $c(x).P \xrightarrow{c()} \lambda x.P$ and $\lambda x.P \xrightarrow{\epsilon(v)} P[v/x]$. Eloise has to be able to match Abelard's commitment to read from a certain channel before the value is known; then she also has to be able to match the reading of the value. The same abstract treatment applies.

Let us consider an example (slightly corrected, and rewritten into our favourite syntax) that appears in [11]. (In fact this is an example where Lin says that the predicate equation system returned by his algorithm is *not* computable by approximants; but our algorithm still terminates.) Consider the processes with integer data:

$M = r(x).M_1(x)$
$M_1(x) = \overline{c}(x).r(u).(\text{if } u > 0 \text{ then } M_1(x+u) \text{ else } M_1(x-u))$
$N = r(y).N_1(y)$
$N_1(y) = \overline{c}(y).r(v).N_1(y+|v|)$

Depending on in what order the algorithm takes the various transitions (it doesn't matter in this case) the initial exploration might go thus, where we show the shape and constraint that will be given to the explore function, after some (implementable) simplification has been done:

$((M, N), \mathit{true})$,
$((M_1(p_{11}), N, r(p_a), 2), p_{11} = p_a)$,
$((M_1(p_{11}), N(p_{21})), p_{11} = p_{21})$

$((r(u).(\text{if } u > 0 \text{ then } M_1(p_{11}+u) \text{ else } M_1(p_{11}-u)), N(p_{21}), \overline{c}(p_{11}), 2), p_{11} = p_{21}),$
$((r(u).(\text{if } u > 0 \text{ then } M_1(p_{11}+u) \text{ else } M_1(p_{11}-u)), r(v).N_1(p_{21}+|v|)), p_{11} = p_{21})$
$((M_1(p_{11}), r(v).N_1(p_{21} + |v|), r(p_a), 2), p_{11} = p_{21} + p_a \wedge p_a > 0)$
$((M_1(p_{11}), N_1(p_{21})), p_{11} = p_{21})$
At this stage the explore function will find that the whole set is a repeat so we backtrack. We explore Abelard's other options at each Abelard choice, but all cases are similar to the above and the backtracking returns $(\mathbf{ff}, \mathbf{tt})$, with the expected strategy.

5.2 Classes of decidable bisimulation questions

In this section we consider for which classes of bisimulation games the algorithm can be shown to terminate. We use Proposition 1, and find that previous work producing abstract versions of transitions systems can often be interpreted as giving an equivalence relation of the kind required by that result. A simple example is the class of non-finite state processes considered by Jonsson and Parrow in [10]. They considered processes with a finite state control – they excluded definitions in which recursion is used under a parallel operator – which may read, store and write data values, but not test them or compute with them in any other way. A typical process is (we use slightly different notation from [10]):

$$\text{MemoryCell}(x) = \overline{\text{read}}(x).\text{MemoryCell}(x) + \text{write}(y).\text{MemoryCell}(y)$$

The set of shapes which may arise in a bisimulation game is finite. An appropriate constraint language is first order logic (over just $\{=\}$). The equivalence relation which relates concrete positions iff they have the same shape and the same pairs of equal parameter values satisfies the conditions of Proposition 1. Therefore the algorithm will always terminate on such processes.

Symbolic transition graphs We omit details and refer the reader to [9]. Informally speaking, the approach is to define a symbolic semantics for a value-passing process algebra giving rise to a symbolic transition graph which is the symbolic analogue of the process's LTS. A symbolic version of bisimulation is defined and an appropriate correspondence theorem proved. If two processes both give rise to finite STGs an algorithm can compute whether (closed) processes are bisimilar, by calculating most general booleans under which pairs of open derivatives are bisimilar. The STG construction can be read as giving an equivalence relation on concrete positions of a bisimulation game, and Proposition 4.2 and 6.3 of [9] do most of the work needed to show it satisfies the conditions required by Proposition 1. We get:

Corollary 1. *The algorithm for checking bisimulation of value passing processes terminates if both processes have finite STGs.*

Region graph of timed automaton A timed (finite) automaton (timed transition table in [1]) has a (finite) set of states S, a fixed finite set C of clocks which take real number values, a finite set of letters Σ, and a transition relation $E \subseteq S \times S \times \Sigma \times 2^C \times \text{Constraint}(C)$ which specifies, given a state and input letter, a constraint on the clock evaluations under which a transition to a new state can be taken and a set of clocks which should then be set to 0. We work in terms of an operational semantics for timed automata in which there are delay transitions labelled $\epsilon(v)$ where $v \in \mathbb{R}_{>0}$ is the length of the delay, and instantaneous action transitions labelled $a \in \Sigma$. A delay transition can always occur; a letter transition is permitted if the appropriate constraint is satisfied. An extended state of the automaton is (s, v) where $s \in S$ and $v : C \to \mathbb{R}^+$ is a clock evaluation. Bisimulation is defined as usual: a bisimulation question is whether $P_1 = (\Phi_1, (s_1, t_1)) \sim P_2 = (\Phi_2, (s_2, t_2))$ for timed transition tables Φ_i and initial extended states (s_i, t_i). [3] showed that bisimulation equivalence is decidable provided that the constraints are boolean combinations of $x < c$ where x is a clock variable and $c \in \mathbb{Q}$ (or wlog $c \in \mathbb{N}$) is constant. (In fact, [3] considered a larger class of processes: we expect our result to extend to that without difficulty.) Under the same conditions we get:

Corollary 2. *The algorithm for checking bisimulation of timed automata terminates.*

Other easy questions An important advantage of our system is that it can answer easy questions about hard processes. As a trivial example, consider the question of bisimilarity of the following two processes expressed in CCS:

$P = (a(x).\bar{b}(x).P | a(y).a(z).\bar{b}(y+z).P)$

$Q = a(u).a(v).a(w).\bar{c}(u+v+w).Q$

P is a very hard process to deal with: it is infinite state, not bisimilar to any finite state process, and its STG(A) is infinite. However, it is obviously not bisimilar to Q because after at most 3 matched inputs along a we must reach a b output by P unmatchable in Q or a c output in Q unmatchable in P. This is obvious to our algorithm, as it should be.

5.3 Model-checking value-passing processes

Here we consider model-checking formulae of the modal mu-calculus on value-passing processes, where actions in the modalities may involve value-passing actions but values may not be carried across fixpoints. (Considering the generalisation to first-order mu-calculi like those of [13],[8] is work in progress.) Thus a set-game position is a shape consisting of a process term and a subformula of the formula to be checked; both may include parameters which are constrained. In [14] etc. the unwinding of fixpoints is done by "the referee": here we assign the unwinding of νs to Abelard and of μs to Eloise, in order to satisfy the condition on $W_\forall$, $W_\exists$. Again the algorithm terminates on "easy" questions.

6 Conclusion and further work

We have shown how the game-based paradigm in which winning strategies are proof objects can be extended to allow a clean consideration of abstract interpretation of games, in such a way as to allow automatic computation of winning strategies for certain games on infinite graphs. We have demonstrated the application of this to the problem of bisimulation of value-passing processes and of timed automata, and have briefly discussed application to other areas such as model-checking the modal mu-calculus.

On the practical side, we intend to complete the implementation of algorithms based on this approach into the Edinburgh Concurrency Workbench. We will also investigate efficiency improvements as mentioned above. Theoretically, we will investigate further applications of this framework. We are considering model-checking value-passing extensions of the modal mu-calculus such as are considered in [13], [8] and elsewhere. Further afield, it should be possible to apply this work to other problems involving infinite state but highly structured game graphs, such as questions about probabilistic processes. The connections with game-theoretic approaches to program and control synthesis, pointed out by a referee and by Martin Abadi, should also be investigated.

Acknowledgements I thank Colin Stirling, Julian Bradfield and the anonymous referees for helpful discussions and comments.

References

1. R. Alur and D.L.Dill, Theory of Timed Automata, Theoretical Computer Science 126(2) pp183-235, 1994
2. J. Bradfield and C. Stirling, Local model checking for infinite state spaces. Theoretical Computer Science, 96 pp 157-174 (1992).
3. K. Cerans, Decidability of bisimulation for parallel timer processes, CAV 92.
4. D. Clark, L. Errington and C. Hankin, Static Analysis of Value-Passing Process Calculi, in Theory and Formal Methods 1994.
5. R. Cleaveland, Optimality in Abstractions of Model Checking, LNCS 983 pp51-63, 1995.
6. R. Cleaveland and J. Reily. Testing-based abstractions for value-passing systems. Proceedings of CONCUR'94, LNCS 836, pp417-432.
7. P. Cousot and R. Cousot, Abstract Interpretation Frameworks, Logic and Computation 2(4) pp511-547, 1992.
8. M. Dam, Model Checking Mobile Processes, Information and Computation 129, 35-51, 1996.
9. M. Hennessy and H. Lin, Symbolic Bisimulations. Theoretical Computer Science, 138:353-389, 1995.
10. B. Jonsson and J. Parrow, Deciding bisimulation equivalences for a class of non-finite-state programs. Information and Computation, 107(2) pp 272-302, Dec 1993.
11. H. Lin, Symbolic Transition Graph with Assignment. Proceedings of CONCUR'96, LNCS 1119, pp50-65.
12. R. Milner, Communication and Concurrency. Prentice Hall 1989.
13. J. Rathke, Symbolic Techniques for Value-Passing Calculi. PhD. thesis, University of Sussex, 1997.
14. P. Stevens and C. Stirling, Practical Model-Checking using Games. Proceedings of TACAS'98, LNCS 1384, pp 85-101
15. C. Stirling, Modal and temporal logics for processes. LNCS 1043 pp149-237. 1996.