Practical Model-Checking Using Games

Perdita Stevens * and Colin Stirling **

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

Abstract. We describe how model-checking games can be the foundation for efficient local model-checking of the modal mu-calculus on transition systems. Game-based algorithms generate winning strategies for a certain game, which can then be used interactively to help the user understand why the property is or is not true of the model. This kind of feedback has advantages over traditional techniques such as error traces. We give a proof technique for verifying such algorithms, and apply it to one which we have implemented in the Edinburgh Concurrency Workbench. We discuss its usability and performance.

1 Introduction

The *modal mu-calculus* (see e.g. [9]) is an expressive logic which can be used to describe properties of systems modelled as labelled transition systems (LTSs). The problem of *model-checking* the mu-calculus on transition systems is that of deciding whether an LTS satisfies a formula. Many model-checking algorithms have been developed and implemented in tools. One such tool is the Edinburgh Concurrency Workbench ([7]), in which users design modular systems using the Calculus of Communicating Systems (CCS). The semantics of a CCS process is given by an LTS, and for model-checking purposes we will think of the LTS as the basic notion; of course feedback to the user is all in terms of CCS derivatives. We are mostly interested in local, on-the-fly algorithms; our algorithm will not require global information about the LTS, so we are free to calculate parts of the LTS as required, and in some cases we will be able to check properties of infinite-state LTSs.

Previous work on model-checking has concentrated mostly on the worst-case time complexity of the algorithm in question. This is not our focus here, and indeed, the algorithm we have implemented does not have the best known worst-case time complexity. Instead, we are concerned with two aspects of the needs of users of model-checking tools.

Firstly, and most importantly, we consider the feedback given to users. Any model-checking algorithm can tell the user whether a formula does or does not

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

hold of a process; however, for a user trying to design a system which should meet a specification given in the mu-calculus, this is not sufficiently helpful. It can be hard to debug such a system, that is, to work out why its properties are not as required and alter it so that they are. Moreover it is not always obvious how an algorithm can provide output that helps. If the system is supposed to satisfy "can do an a-action" ($\langle a \rangle \mathbf{T}$) and does not, the tool need only show the user the possible first actions of the process to demonstrate what went wrong. But if the user expects the model to satisfy a more complex property such as "there is some path on which P holds infinitely often" – in mu-calculus

$$\nu X.\, \mu Y.\, (P \wedge \langle - \rangle X) \vee \langle - \rangle Y$$

– and it does not, then it is much less clear what good diagnostics are: in particular, the tool can't give any particular path through the system, and expect it to be helpful to the user. Presumably, however, the user has some path in mind; if the user somehow gives the path, we may expect to be able to convince her/him that P does not hold infinitely often on that path. We will return to this example to show the user playing against the tool's winning strategy.

Secondly, we are interested in the performance of our algorithm on the examples that arise in practice. This can be strongly affected by common characteristics of practical systems; we shall describe some improvements we have made which have no impact on the worst-case complexity of the algorithm, yet have dramatic effects on the practical performance. This is an area which has not been much addressed in the local model-checking literature. Improving performance necessitates experimentation; it is important to be able to try out variations on an algorithm to see whether they are practical improvements, whilst remaining confident that the algorithm is correct. Therefore we need to have a flexible proof technique which will work for a wide family of algorithms; we need to be able to make changes to our algorithm and prove them correct without needing to re-prove correctness from scratch. One of our contributions here is such a framework; we shall discuss variants on the algorithm as we go.

2 Background and plan

The syntax of the modal mu-calculus we use (positive form) is:

$$\Phi ::= \mathbf{T} \mid \mathbf{F} \mid Z \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid [K]\Phi \mid \langle K \rangle \Phi \mid \nu Z.\, \Phi \mid \mu Z.\, \Phi$$

where Z ranges over a family of propositional variables, and K over subsets of the set of labels: $-K$ denotes K's complement. [9] provides a tutorial introduction to the mu-calculus and its use: here we assume familiarity with it. We write $\sigma Z.\, \Phi$ for "either $\mu Z.\, \Phi$ or $\nu Z.\, \Phi$". A *closed* formula contains no free variables.

Recently Stirling ([8]) has developed a theory of model-checking games in which the truth of a formula at a state is equivalent to the existence of a (history-free) winning strategy for a certain game. This can be seen as a reformulation of tableau-based approaches such as [10]: a (history-free) winning strategy looks

very much like a successful (canonical) tableau. The work is also related to [2] and methods in [3]. The history-free winning strategy, whose size is linear in the size of the problem, that is, in the product of the number of states of the model and the number of subformulae of the formula, should be generated by the model-checking algorithm.[1] Given a tool which takes advantage of this, the user can use the strategy as an *interactive* diagnostic tool to correct wayward intuitions and debug the model. As we shall show, expecting the opposite answer to the one the tool gave corresponds naturally to expecting to be able to beat the tool when it follows its winning strategy. Losing repeatedly to the tool and observing how this happens, the user can locate the problem with the model.

On the theoretical side, we also find thinking in terms of games helpful: it seems to expose the structure and duality of the problem well. An alternative approach is via Boolean graphs [1] or equation systems [5]: indeed Mader has demonstrated the equivalence of Boolean equation systems and the model-checking problem.

We first briefly describe the basic model-checking game, and we introduce *open games* which are a formalisation of the idea of exploring parts of the game-tree separately; a particular case of an open game is the basic model-checking game. Using open games to prove correctness, we describe our implementation of a game-based algorithm in the Edinburgh Concurrency Workbench (CWB). This includes features to allow the user to use the winning strategy interactively, as described. We discuss the usability and performance of the new algorithm, which is encouraging. Finally we mention future work.

Most proofs and some details are omitted for reasons of space.

2.1 The games

We wish to establish whether a process P satisfies a closed Φ of the modal mu-calculus. We assume all bound variables in Φ are distinct, renaming them if necessary to ensure this: the assumption is used in Rule 6 of Figure 1.

The *model-checking game* $\mathcal{G}(P, \Phi)$, is played by $\forall$belard (or Player I, or Opponent) and $\exists$loise (or Player II, or Player). $\forall$belard attempts to show that P fails to have the property Φ whereas $\exists$loise tries to show that P does have Φ. We write Player A and Player B for "a player" and "the other player" when it doesn't matter which is which.

A *play* of $\mathcal{G}(P_0, \Phi_0)$ is a finite or infinite length sequence of the form $(P_0, \Phi_0) \ldots (P_n, \Phi_n) \ldots$ where each Φ_i is a subformula of Φ_0 and each P_i is a derivative of P_0. (We call such a pair a *configuration* of the game.)

Suppose a play (so far) is $(P_0, \Phi_0) \ldots (P_j, \Phi_j)$. The moves are given in Figure 1: note that the form of the available moves, and which player chooses, are determined by the form of Φ_j. Each time the current game configuration is

[1] Somewhat surprisingly, it is not easy to construct a history-free winning strategy from the information calculated by an algorithm (even the obvious variant on this one) which does not explicitly record a strategy. The relationship between proving that a strategy exists and finding it deserves further investigation.

1. if $\Phi_j = \Psi_1 \wedge \Psi_2$ then $\forall$belard chooses Φ_{j+1} to be either Ψ_1 or Ψ_2, and P_{j+1} is P_j.
2. if $\Phi_j = \Psi_1 \vee \Psi_2$ then $\exists$loise chooses Φ_{j+1} to be either Ψ_1 or Ψ_2, and P_{j+1} is P_j.
3. if $\Phi_j = [K]\Psi$ then $\forall$belard chooses a transition $P_j \xrightarrow{a} P_{j+1}$ with $a \in K$ and Φ_{j+1} is Ψ.
4. if $\Phi_j = \langle K\rangle\Psi$ then $\exists$loise chooses a transition $P_j \xrightarrow{a} P_{j+1}$ with $a \in K$ and Φ_{j+1} is Ψ.
5. if $\Phi_j = \sigma Z.\Psi$ then Φ_{j+1} is Z and P_{j+1} is P_j.
6. if $\Phi_j = Z$ and Z is bound by $\sigma Z.\Psi$ then Φ_{j+1} is Ψ and P_{j+1} is P_j.

Figure 1. Rules for the next move in a game play

$(P, \sigma Z.\Psi)$, at the next step this fixed point is abbreviated to Z, and each time the configuration is (Q, Z) the fixed point subformula it identifies is, in effect, unfolded once as the formula becomes Ψ.[2]

The conditions for winning a play are given in Figure 2. $\forall$belard wins if a blatantly false configuration is reached, or if $\exists$loise is stuck, and dually for $\exists$loise. The remaining condition identifies who wins an infinite length play. We call variables bound by ν $\exists$loise-variables and variables bound by μ $\forall$belard-variables, and the notion of subsuming is:

Definition *Suppose $\sigma X.\Psi$ and $\sigma Y.\Psi'$ are subformulae of a formula Φ. X subsumes Y if $\sigma Y.\Psi'$ is a subformula of $\sigma X.\Psi$.*

We omit the easy proof that X in winning condition 3 is indeed unique, so that the condition is well-defined.

A *strategy* π for Player A is a set of rules telling Player A how to move: that is, it is a partial function from plays[3] to configurations, which given a play $p \in \text{dom } \pi$ ending in a configuration (Q, Ψ) from which Player A must move, returns a non-empty set of legal next configurations. If every such set is a singleton, we say that π is *deterministic* (and in this case we will usually think of $\pi(p)$ as a configuration, rather than as a singleton set of configurations). We call π *history-free* if $\pi(p)$ is determined solely by the final configuration (Q, Ψ) of p, irrespective of the rest of the play. A play q *follows* π if for every proper prefix p of q ending in an A-choice, $p \in \text{dom } \pi$ and the next configuration of q after p is in $\pi(p)$. π is *complete* if whenever p is a play following π and ending in a configuration from which Player A must choose, $\pi(p)$ is defined. Otherwise it is partial. π is a winning strategy if it is complete and B does not win any play which follows π. A history-free complete strategy may be regarded as a partial function from configurations to configurations.

The basic theorem we exploit is from [8]:

[2] As there are no choices here it doesn't matter who "chooses" – we say that the Referee moves.

[3] Plays are just sequences of moves, which need not yet be decided.

∀belard wins

1. The play is $(P_0, \Phi_0) \ldots (P_n, \Phi_n)$ and $\Phi_n = \mathbf{F}$.
2. The play is $(P_0, \Phi_0) \ldots (P_n, \Phi_n)$ and $\Phi_n = \langle K \rangle \Psi$ and $\{Q : P \xrightarrow{a} Q \text{ and } a \in K\} = \emptyset$.
3. The play $(P_0, \Phi_0) \ldots (P_n, \Phi_n) \ldots$ has infinite length and the unique variable X which occurs infinitely often and which subsumes all other variables that occur infinitely often identifies a least fixed point subformula $\mu X.\Psi$.

∃loise wins

1. The play is $(P_0, \Phi_0) \ldots (P_n, \Phi_n)$ and $\Phi_n = \mathbf{T}$.
2. The play is $(P_0, \Phi_0) \ldots (P_n, \Phi_n)$ and $\Phi_n = [K]\Psi$ and $\{Q : P \xrightarrow{a} Q \text{ and } a \in K\} = \emptyset$.
3. The play $(P_0, \Phi_0) \ldots (P_n, \Phi_n) \ldots$ has infinite length and the unique variable X which occurs infinitely often and which subsumes all other variables that occur infinitely often identifies a greatest fixed point subformula $\nu X.\Psi$.

Figure 2. Winning conditions

Theorem 2.1. *$P \models \Phi$ iff ∃loise has a winning strategy for $\mathcal{G}(P, \Phi)$.*

In fact, the strategy can be required to be deterministic and history-free, but this will follow from what we prove here so we don't need to quote it. The proof follows closely the soundness and completeness proofs from [10]: indeed winning strategies and successful tableaux are closely related.

3 Techniques: indexes and open games

In this section we introduce the definitions and basic results which will enable us to prove our algorithm (and others in the same family) correct.

When we have to think about the play that lead to a configuration, we are often only interested in the positions at which fixed points are unwound (indeed, it is this that leads to the equivalent formulations such as MC games and Boolean equation systems). This motivates:

3.1 Indexes

Fix a "root" formula with fixed point variables $X_1, \ldots X_n$. We have a natural partial order on the variables, given by $X \leq Y$ if X subsumes Y. Now given a play p with final configuration (P, Φ), the *index of* p written index(p), or the *index of* (P, Φ) *with respect to* p is a map i from the fixed point variables $\{X_1, \ldots X_n\}$ to natural numbers, in which $i(X)$ is the number of times in the play p the fixed point X has been unwound *since any other fixed point which subsumes* X *was*

last unwound. We think of i as being a concise description of the features of p that we may have to care about. We will write $\mathbf{0}$ for the everywhere-zero index.

Let (Q, Ψ) be any legal next configuration after (P, Φ). Then, of course, the index of (Q, Ψ) relative to the extended path, $\mathrm{index}(p(Q, \Psi))$, can be determined from i and Φ without further reference to the play: it is i itself unless Φ was a fixed point variable X (that is, Rule 6 was applied), in which case it is i modified by adding 1 to $i(X)$ and setting $i(Y)$ to 0 for any $Y \neq X$ subsumed by X. Let us denote this new index by $\mathrm{next}(i, \Phi)$.

It is useful to think of the index as being part of the configuration of the game (writing (P, Φ, i)). The index is determined by the play that led to (P, Φ) and by the initial index, which in the case of the standard model-checking game is $\mathbf{0}$.

Given a transition system, any index which can arise in a legal play from the root is called *reachable*. If i is a reachable index then j is reachable from i if there is some play pq such that $\mathrm{index}(p) = i$ and $\mathrm{index}(pq) = j$. Note that any reachable index has the property that its non-zero entries are those corresponding to the variables on some prefix of a single branch of the partial order on variables; if i is reachable and $i(X) \neq 0$ and $i(Y) \neq 0$ then either X subsumes Y or vice versa.

We shall need a notion of the leading difference of two different indexes i and j, which should be the outermost variable X such that $i(X) \neq j(X)$. However, this is not yet well-defined: for example, considering the formula

$$\nu X.(\nu Y.[a]X) \wedge (\mu Z.[a]X)$$

indexes $\{X \mapsto 1, Y \mapsto 1, Z \mapsto 0\}$ and $\{X \mapsto 1, Y \mapsto 0, Z \mapsto 1\}$ may both arise as indexes for $[a]X$; is their leading difference Y or Z? Either may be chosen, but (for example to make $\leq_A$ below transitive) we need to make consistent choices.

Let us fix a total order on the variables compatible with the natural partial order (that is, if X subsumes Y then $X \leq Y$), and so we can implement the index as an array of natural numbers (from outermost variables on the left to innermost on the right) as an array of natural numbers. We can implement the unwinding of a variable by incrementing its entry and zeroing every entry to the right of that one. For every such entry is either one that we must zero because its variable is subsumed by the one unwound, or must in fact be already 0. We define:

Definition *The* leading difference *of two different indexes i and j is the least variable X such that $i(X) \neq j(X)$.*

Notice that if i is reachable from j then regardless of which order on variables we choose the leading difference of i and j will be the same.

3.2 The open game

Given a root configuration, i.e. a transition system and formula we wish to check, we will need to consider *open games*, in which play proceeds as in the standard model-checking game, except that certain plays are made to terminate earlier

than they would in the standard game, and that we may start from some position which is notionally part way through the standard model-checking game. The idea is to have a disciplined way of exploring what may happen in a limited part of the standard game, as a proof technique for proving the correctness of algorithms which generate winning strategies for the standard game.

An *assumption* (the motivation for the terminology will become clearer later) is a triple (P, Φ, i), where (P, Φ) is a configuration as usual, and i is an index.

A *decision for player A* is either (P, Φ, i) where Φ is not a formula from which A must move, or else $((P, \Phi, i), (Q, \Psi))$ where (Q, Ψ) is a configuration which A may choose from the A-choice point (P, Φ). With a slight abuse of terminology we will sometimes talk about (P, Φ, i) as the decision in this case too, and refer to (Q, Ψ) as the move which justifies the decision (P, Φ, i).

Let Γ be a set of assumptions, (P, Φ) a configuration, i an index. The game $G(\Gamma, (P, \Phi, i))$ begins at configuration (P, Φ) and proceeds with moves as described above. As described, we maintain the index recording how many times each fixed point has been unwound; but in $G(\Gamma, (P, \Phi, i))$ the index is initialised to i, not to the everywhere-0 index. The winning conditions are as before, with the addition of the extra rule that if play reaches a configuration (Q, Ψ) with index k, where $(Q, \Psi, j) \in \Gamma$ and $j \neq k$, then play terminates. The winner is A, the player for whom $j <_A k$, according to:

Definition *$i <_A j$, where A is a player and i and j are indexes, means that either:*

1. *The leading difference of i and j corresponds to a variable X which belongs to player A, and $i(X) < j(X)$, or*
2. *The leading difference of i and j corresponds to a variable X which belongs to player B, and $i(X) > j(X)$.*

We write $i \leq_A j$ for $i <_A j$ or $i = j$.

(Recall that we say maximal fixed points belong to ∃loise and minimal fixed points to ∀belard.) Given the total order on variables which made "leading difference" well defined, $\leq_A$ is a total order: it is "partially reversed lexicographic order" in the sense that B variables are compared "upside down". We will of course get different total orders depending on which order on variables we choose. (But again, if j is reachable from i then $i <_A j$ in every possible order or none.)

Of course $<_B$ is just the reverse of $<_A$: $i <_A j$ iff $j <_B i$. (But it's convenient to have separate symbols for the two related orders.)

The relationship between open games and the standard games is that $G(\emptyset, (P, \Phi, \mathbf{0}))$ is the standard model-checking game starting at (P, Φ). We say that σ *A-wins* game G if σ is a winning strategy for player A for G.

Lemma 3.1. *σ A-wins $G(\Gamma \cup \{(P, \Phi, i)\}, (P, \Phi, i)) \Rightarrow \sigma$ A-wins $G(\Gamma, (P, \Phi, i))$*

Lemma 3.2. *(σ A-wins $G(\Gamma, (P, \Phi, i)) \wedge i \leq_A j) \Rightarrow \sigma$ A-wins $G(\Gamma, (P, \Phi, j))$.*

This lemma can be strengthened for particular classes of formulae (by a weaker relationship between i and j): for example the whole algorithm specialises in the

case of model-checking a non-alternating formula to one which does not have to store indexes with decisions at all. However, we defer discussion of this to the full paper. For now we will say that an A-decision (P, Φ, i) applies at (P, Φ, j) exactly when $i \leq_A j$.

Lemma 3.3. *If σ A-wins $G(\Gamma \cup \{(D, \Phi, j)\}, (C, \Psi, i))$ and for some $k \leq_A j$ τ A-wins $G(\Gamma, (D, \Phi, k))$, then we can construct σ' such that σ' A-wins $G(\Gamma, (C, \Psi, i))$. If $\sigma = \tau$ then $\sigma' = \sigma = \tau$ will do.*

4 A practical algorithm

The core of the final algorithm is shown in Figure 3.

In brief, we explore depth first, maintaining a *playList* which records the sequence of configurations which lead from the root to the *current node* (empty if the current node *is* the root), along with, for each choice-point, which choices remain unexplored. We consider whether to stop exploring at the current node, which we do if winning conditions 1 or 2 apply, or if we hit a repeat – that is, the configuration of the current node already appears on the *playList* – or if there is some "applicable decision" (see below). If none of these conditions apply, we add the current node to the *playList*, and choose some so-far unexplored successor to be the new current node. If one of the stopping conditions does apply, then for the appropriate player, say A, we retrace our steps along the *playList*, or *backtrack for Player A*, adding decisions for A in a way to be described. Notionally we are trying to build an A-winning strategy. 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 for Player A to a B-choice-point n, we see whether the *playList* entry records any unexplored B-choices; notionally, we must see whether Player B could have made a better choice than the one that has just led to an A-win. If there are any unexplored B-choices, we must explore one, replacing the record for n on the *playList* having altered it to reflect the fact that one fewer B-choices remain unexplored. Eventually we either backtrack for B through n in which case n will get decided for B, or run out of unexplored B-moves in which case we may mark n as decided for A: Player B has tried all the possibilities and none succeeded.

The application of a decision as a stopping condition is the reuse of previously calculated information.

At any stage in the execution of the algorithm we have a single set of current assumptions for each player, say $\Gamma_\forall$ and $\Gamma_\exists$. Each assumption set will be a subset of the positions on the current *playList*; since we stop exploring any time we encounter a repeat, this implies in particular that for any configuration (Q, Ψ) and player A, there will be at most one k such that $(Q, \Psi, k) \in \Gamma_A$, though the same configuration may appear once in each assumption set.

Moreover at each step for each player A we have a collection of decisions Δ_A. More precisely, for each configuration (P, Φ) and player A we have a stack (possibly empty) of decisions for A at (P, Φ) with different indexes, where if

```
function explore (config as (state, formula)) index playList
-- take the first case which applies:
case:
-- first dispose of trivial cases
  (formula is T) then
     backtrack Eloise playList config
  (formula is F) then
     backtrack Abelard playList config
-- then see if any decision can be used
  (there is a valid decision, (config, i, playerA) say,
   which is applicable at index) then
     backtrack playerA playList config
-- then see if this is a repeat
  (config is in playList with index i: repeat won by playerA) then
     record that (config, i) used as an assumption for playerA
     backtrack playerA playList config
-- otherwise we have to try to keep playing
  (there are no legal next moves for the player who chooses a move
   from this config, say playerA) then -- A can't go, so B wins
     backtrack playerB playList config
  (else if none of the above apply) then
     pick a next move, say to config2
     create index2, index = index2 unless we just unwound a fixpoint
     record that we've played through (config, index)
     create r, a new entry for play list recording untaken choices
     explore config2 index2 (r::playList)
end case

function backtrack playerA playList whereWeCameFrom
  case
    (playList is []) then
       playerA -- playList was empty, we're done
    (playList is (h::t)) then
       if ((config recorded in h) is a playerB choicept & h shows a
            non-empty list (h'::t') of unexplored pl.B choices) then
         alter h to give t' as the unexplored playerB choices
         explore h' playList
       else
         add decision for playerA at h justified by whereWeCameFrom
         if (h was used as an assumption for playerB)
         then (forget all decisions which may have relied on h)
         backtrack playerA t (config recorded at h)
  end case
```

Figure 3. The core of the algorithm

$d_{n+1} = (P, \Phi, i)$ is a decision above $d_n = (P, \Phi, j)$ in the stack, then $i <_A j$. The decision at the top of the stack is the best decision in the sense that when we ask whether any decision applies at our current node, we only have to examine this top element; if it does not apply, neither does any other decision in the stack. If the configuration is an A-choice point, then the "best" move for A to make is that justifying the top decision. Given the current collection of A-decisions Δ_A, the collection of these best moves for each configuration forms a history-free partial strategy (for any of the relevant games), which we call σ_A.

The reader may wonder why we keep decisions other than the top ones at all: the reason is that we may have to *invalidate* sets of decisions, when assumptions on which they rely are removed from the current assumption set. These decisions are then removed from the stacks, which may cause lower decisions to be exposed as the new top decisions, changing the current strategy σ_A.

We do not wish to maintain a complete dependency graph of assumptions and the decisions that rest on them: we use a simple "time-stamping" mechanism instead. (This design decision will doubtless surprise some readers. There are various reasons for it: simplicity, space, and experience with profiling earlier algorithms which suggested that in fact, checking the dependencies in detail was in practice costing more time than it saved. Nevertheless, it would be interesting to return to this topic and see whether we can do better.) At each step of the algorithm we increment the "time" on a single global "clock". When we add a configuration to the *playList* we stamp it with the time when it was added. When we add a decision to $\Delta_\exists$ or $\Delta_\forall$ we stamp it with the time when it was added. Also, for a given decision d we will want to talk about $playList(d)$, being the value of *playList* at the time when the decision d was added. (The algorithm does not need to record this path, it's for purposes of proof only.) Then at any stage, given a decision $d \in \Delta_A$ we can define the current *branching time* of d as the time-stamp on the earliest element of *playList* which is not a member of $playList(d)$. The idea is that if d depends directly or indirectly on another decision, that decision must have been added before the branching time of d. (Care is needed, to ensure that if d depended on an assumption which was later discharged and turned into a decision d', we record properly the fact that d depends on d' and on any decisions on which d' depends, even though d' was added after d.)

Because a decision which currently does not contribute to σ_A may later do so, our invariant has to refer to more general strategies than σ_A. Suppose that at some stage in the execution of the algorithm we have an A-decision $d = (P, \Phi, k) \in \Delta_A$, and let $\Delta_d \subseteq \Delta_A$ and $\Gamma_d \subseteq \Gamma_A$ be the sets of A-decisions and A-assumptions respectively which were time-stamped before the branching time of d. That is, Δ_d is the set of current A-decisions which were added before the branching time, and Γ_d is the set of assumptions which were *played through* (not necessarily used as assumptions) before then, in other words, the current A-assumptions which appear on $playList(d) \cap playList$. Then we will define a (partial, non-deterministic) strategy τ_d of $G(\Gamma_d, d)$ by saying that a play $p = p_1 \ldots p_n$ (where each $p_i = (P_i, \Phi_i)$) of $G(\Gamma_d, d)$ follows τ_d iff there is some

sequence $k_1, \ldots, k_n$ of indexes satisfying:

- $(P_1, \Phi_1, k_1) = d$
- $\forall\, i : 0 < i < n\ (P_i, \Phi_i, k_i) \in \Delta_d$; moreover if (P_i, Φ_i) is an A-choice point then the justifying move for this decision is (P_{i+1}, Φ_{i+1}).
- $\forall\, i : 0 < i < n\ k_{i+1} \leq_A \text{next}(k_i, \Phi_i)$

Very roughly, the intuition behind this is that play may move up the stack of decisions to a decision which is at least as safe as the starting point, but never down. A play following τ_d is a path through the part of the game tree which has been decided by the algorithm, following the justifying moves, with the modification that we are allowed to jump to another equally safe branch if it is valid at the present position to reuse the calculation it represents.

Our algorithm will maintain the invariant:

(I) for every Player A decision $d \in \Delta_A$, τ_d is a winning strategy for $G(\Gamma_d, d)$.

Therefore if the algorithm terminates by adding a decision for player A at the root configuration (P, Φ), then since $\Gamma_{(P,\Phi,0)} \subseteq \Gamma_A \subseteq \mathit{playList} = \emptyset$, $\Gamma_{(P,\Phi,0)}$ must then be empty, so I gives that $\tau_{(P,\Phi,0)}$ is a winning strategy for the standard model-checking game $G(\emptyset, (P, \Phi, \mathbf{0}))$ starting at the root. To get a deterministic history-free strategy, observe that σ_A is a substrategy of $\tau_{(P,\Phi,0)}$, in the sense that the plays following σ_A are exactly the plays following $\tau_{(P,\Phi,0)}$ which always follow the move prescribed by the top decision on the stack. Moreover completeness of σ_A follows from completeness of $\tau_{(P,\Phi,0)}$, which gives the result. Thus we have to show

1. that the algorithm does indeed maintain I
2. that it terminates with no assumptions, making a decision at the root.

We start with no decisions or assumptions for either player. This establishes the invariant. In what follows we will use primes to denote the old values of things, for example defining the new A-assumption set Γ_A in terms of the old one Γ'_A.

Here we omit the proofs that the invariant is maintained, but we give the relevant settings of assumption and decision sets, and have presented all the important ingredients of the proof.

Exploring Exploring doesn't change the decision sets or the assumption sets, so there's nothing to prove about the invariant. What it does do is to set up the recorded structure so that when we backtrack along the *playList* we always are moving backwards along a real play. Moreover, because we always stop playing and start backtracking when we encounter a repeat configuration, and because we never explore a branch more than once, for finite LTSs termination is automatic.

Stopping the play Play stops when we encounter (a) a position where winning conditions 1 or 2 apply, or (b) a repeat of a configuration on the *playList*, or (c) a current applicable decision. Suppose play stops at (Q, Ψ, j).

(a) If the winning condition says that A wins, we add a decision, setting $\Delta_A = \Delta'_A \cup \{(Q, \Psi, j)\}$. Backtrack for A.

(b) If we stop because of a repeat – say (Q, Ψ, i) occurs on the *playList*– we determine for which player, say A, $i <_A j$. We add an assumption, setting $\Gamma_A = \Gamma'_A \cup \{(Q, \Psi, i)\}$. Backtrack for A.

(c) If there is some $(Q, \Psi, m) \in \Delta_A$ with $m \leq_A j$, backtrack for A.

Backtracking: building the strategies, adding decisions As we have seen, when the algorithm backtracks for player A from (Q, Ψ, j) we know that there is either an assumption $(Q, \Psi, m) \in \Gamma_A$ with $m <_A j$, or some decision $(Q, \Psi, m) \in \Delta_A$ such that $m \leq_A j$.

If (Q, Ψ, j) is the initial position, Γ_A is empty and we must have a decision which applies at the root (in fact, it's easy to see that we must have a decision *at* the root itself), so we're done: we return the answer A and the current A-strategy σ_A. Otherwise, we backtrack to the end of the old *playList*, say to (P, Φ, i).

(a) If (P, Φ, i) is a B-choice point and there are any unexplored B-choices, then we pick one and play to it, returning (P, Φ, i) to the end of the *playList* after updating the record of unexplored moves from (P, Φ, i). We do not alter decisions or assumptions.

(b) Otherwise, we continue to backtrack for A from (P, Φ, i), after having updated decisions and assumptions as follows:

- We add an A decision: $\Delta_A = \Delta'_A \cup \{(P, \Phi, i)\}$; if (P, Φ) is an A-choice point this is justified by move (Q, Ψ).
- If $(P, \Phi, i) \in \Gamma'_A$ we remove (discharge) it: that is, $\Gamma_A = \Gamma'_A \setminus \{(P, \Phi, i)\}$.
- If $(P, \Phi, i) \in \Gamma'_B$ then we must not only remove this assumption, but also remove B-decisions that "may have depended on this assumption". There are various sound ways of doing this: let $\Delta_B \subseteq \Delta'_B$ be the decisions which we decide to keep. We set $\Gamma_B = \Gamma'_B \setminus \{(P, \Phi, i)\}$.

Since in this version of the algorithm we are not keeping full dependency information, we use a time-stamping mechanism to identify which decisions to discard. Specifically, we set Δ_B to be those decisions $d \in \Delta'_B$ whose time-stamps are earlier than the time-stamp t_1 on (P, Φ, i): that is, which were added to Δ_B before we played through (P, Φ, i). (We implement the forgetting of decisions outside Δ_B by unioning the interval $[t_1, now]$ with the set of B-invalid times; then the procedure that looks for decisions discards and ignores any decisions whose time-stamps are invalid times. This simply saves us the time-expensive procedure of going through all our decisions removing invalid ones every time an assumption is removed.) By keeping more information about the dependencies (up to a complete dependency graph) we could of course be less pessimistic; but this approach works surprisingly well in practice.

5 The tool

We have integrated the algorithm described above into the Edinburgh Concurrency Workbench (CWB), which is a well-established and powerful tool for the analysis of concurrent systems expressed in CCS. The CWB is written in Standard ML, and the model-checker is an ML module. ML provides strong type-checking in a convenient high-level functional language, but imposes severe performance penalties. Experience has shown that it is crucial that CWB code be readable: because of the importance of dependability and ease of maintenance, this is even more important than efficiency.

The CWB's WWW home page, from which the current public version can be obtained, is http://www.dcs.ed.ac.uk/home/cwb. The version described here is 7.1beta.

5.1 Interactive diagnostics

We offer the user the chance to play against the CWB, which takes the winning strategy. Of course the user will lose, and the ways in which the CWB defeats the user can help the user to understand the problem, and especially to dispel mistaken intuition. For example, suppose the user believes that the property "there exists an infinite path on which, infinitely often, a is the only action possible", i.e. the formula in the Introduction with $P = \langle a \rangle \mathbf{T} \wedge [-a]\,\mathbf{F}$, holds of the system A given in CCS in Figure 4. The user must have some path in mind; perhaps the specification should define D as $b.A$, not $b.B$, and the path the user has in mind is that given by following a b transition whenever there is one, and expecting P to be true only at A. In playing against the CWB, the user can test this intuition by trying to follow this path, and will find that A isn't reached infinitely often.[4] Sections from one possible play are shown in figure 5.1.[5] Notice that there are many infinite paths through the system, and the CWB could not (at any rate, without considerable "intelligence"!) tell which one the user had in mind; so no individual error trace suffices as a debugging aid.

5.2 Usability

Initial feedback on the usability of the games have been encouraging. In the light of comments from users, we have made a number of improvements in the interface. For example:

[4] Since users cannot be expected to enjoy playing infinite games, a play terminates when it repeats, and the winner is the owner of the outermost variable unwound on the repeat section. Provided that the user is thinking of a *history-free* strategy – and it is difficult to imagine why this would not be so – s/he will get the same information out of this game as out of the original.

[5] We have deleted some blank lines in order to fit the example on the page! The original is more readable.

```
Command: agent A = a.B;
Command: agent B = a.C + b.D;
Command: agent C = a.C + c.B;
Command: agent D = b.B;
Command: prop P = max (X.min (Y.((<a>T & [-a]F) & <->X) | <->Y));
Command: checkprop (A, P);
false
Would you like to play (and lose!) a game against the CWB? (y or n) y
The CWB will choose Abelard's moves.
You can choose Eloise's.
1: Current position:
A
X
The referee unwinds the fixpoint.
2: Current position:
A
Y
The referee unwinds the fixpoint.
3: Current position:
A
(<a>T & [-a]F & <->X) | <->Y
Your turn (playing Eloise)
1:
A
<a>T & [-a]F & <->X
2:
A
<->Y
Which move?
1
4: Current position:
A
<a>T & [-a]F & <->X

[...the play must continue to:]

The referee unwinds the fixpoint.
7: Current position:
B
Y

[... user follows the b-path and chooses disjunct <->Y at B and at D...]

13: Current position:
B
Y
The CWB (playing Abelard) won, because of a repeat
Another game? (y or n) n
```

Figure 4. Playing a game

- Different users, with different problems, need different degrees of automation of the game playing process. Initially, the CWB did not ask users to acknowledge its own moves or the moves of the referee, and it did not ask users to choose a move in circumstances where only one legal move was available. Some users like this, for example, where plays are long and only short sections are of interest. However, sometimes such behaviour is disorienting, and it's easier to follow the game if you take some action on every move, even when it's only an acknowledgement. Therefore the CWB now makes this behaviour configurable (via command "toggle;").
- It's important to be able to examine past portions of the play after the fact. Running the CWB using its Emacs mode is the easiest way to do this.
- We experimented with a graphical user interface for playing games, but in fact for non-trivial examples, this seemed to be less usable than the text version, partly because of the greater ease of finding relevant sections of a long play when running under a powerful editor.

5.3 Implementation and performance

We have done considerable experimentation with correct variants on the algorithm presented, with the aim of finding which theoretical improvements work well in practice. In many cases our major improvements have been achieved by making changes which have no effect on the worst case complexity of the algorithm. For example, when we arrive at a configuration for which there is a decision, but where we cannot prove that it is safe to apply the decision, we must continue to play. Originally, we played using the first move that came to hand, as though we were meeting the configuration for the first time. Later we modified this so that we try first the move that is recorded in the strategy against the decision we weren't allowed to use. Because it often happens that the same strategy will work, even though we didn't have the means to prove it, this often reduces the re-exploration procedure to a process of checking a strategy, rather than finding one. The old strategy *may* not work – so there is no improvement in the worst-case complexity – but in practice, it usually does, so there is a worthwhile improvement in practical performance.

We have already mentioned another example of the same kind: that rather than process decisions every time an assumption is invalidated, we keep a record of which time-stamps correspond to valid decisions, and simply update this record whenever an assumption is invalidated. The check for whether a decision is still valid is made immediately before the decision is used. Because most decisions are never used, this is very helpful.

We use structure-sharing techniques to avoid keeping multiple copies of data structures representing states, formulae etc, and we calculate the game graph and the LTS on the fly as needed. When model-checking a non-alternating formula, we use specialised versions of the decision lookup functions to take advantage of the fact that (as mentioned previously, though proof and detail is omitted) there is no need to store indexes with decisions in this case, as a decision always applies.

We compare the implementation of the new algorithm with the model-checker from the CWB version 7.0, which is an optimised version of a tableau based method. (We ran both in the same executable, so both used the same infrastructure for calculating LTSs etc.)

Practically the game-based implementation vastly outperforms the old tableau-based implementation. We show a few examples. The numbered rows represent that number of parallel copies of a cycler being tested for deadlock-freedom, an alternation-free formula. Peterson uses a complex fairness property with twelve fixed points and two alternations. Where a figure is absent for CWB v7.0 this is because we have not been able to run the example to completion.

We wish to emphasise the *difference* between the figures, not the actual performance, which suffers from inefficiencies elsewhere in the CWB code (and the fact that it's written in ML). Time spent in the model-checking algorithm itself – rather than calculating transitions from CCS terms (on the fly, once only per state) – is given as (MC: n). Times are in user cpu seconds.

Name	Subformulae	States	CWB v7.0 (u-cpu-s)	new CWB (u-cpu-s)
four	4	626	6 (MC: 5)	1 (MC: < 1)
five	4	3126	419 (MC: 413)	9 (MC: 6)
six	4	15626	-	64 (MC: 33)
seven	4	78126	-	371 (MC: 190)
peterson	112	241	-	15 (MC: 14)

6 Conclusions and further work

We have developed and implemented a model-checking algorithm which produces automatically a "proof object" which can interactively help the user to understand and debug systems. The game techniques which we have used seem promising both because of this interaction paradigm and as a way of developing efficient algorithms. In future we intend to gather and build upon information from CWB users about what kinds of interaction are practically useful, and to improve and extend the interface accordingly. We will also continue to investigate variations on the algorithm, with the intention of improving its performance on practical examples further. We should also like to use these game-based techniques to investigate complexity issues, in particular, to compare the problem of finding a winning strategy with the model-checking problem, which can be seen as that of proving that a winning strategy exists without necessarily constructing it. We are in the process of applying similar techniques to classes of infinite state processes, in particular, to well-behaved value-passing CCS processes. This work will be incorporated into a future version of the CWB.

References

1. Andersen, H. (1994). Model checking and boolean graphs. *Theoretical Comp. Science*, **126**, 3-30.

2. Bernholtz, O., Vardi, M. and Wolper, P. (1994). An automata-theoretic approach to branching-time model checking. *Lecture Notes in Computer Science*, **818**, 142-155.
3. Emerson, E., Jutla, C., and Sistla, A. (1993). On model checking for fragments of μ-calculus. *Lecture Notes in Computer Science*, **697**, 385-396.
4. Long, D., Browne, A., Clarke, E., Jha, S., and Marrero, W. (1994) An improved algorithm for the evaluation of fixpoint expressions. *Lecture Notes in Computer Science*, **818**, 338-350.
5. Mader, A. (1996) Verification of Modal Properties Using Boolean Equation Systems. *Doctoral dissertation, Institut für Informatik, Technische Universität München.*
6. Milner, R. (1989) Communication and Concurrency *Prentice Hall*
7. Moller, F. and Stevens, P. (1996). The Edinburgh Concurrency Workbench user manual. http://www.dcs.ed.ac.uk/home/cwb.
8. Stirling, C. (1995). Local model checking games. *Lecture Notes in Computer Science*, **962**, 1-11.
9. Stirling, C. (1996). Modal and temporal logics for processes. *Lecture Notes in Computer Science*, **1043**, 149-237.
10. Stirling, C. and Walker, D. (1991) Local model checking in the modal mu-calculus. *Theoretical Computer Science*, **89**, 161-177.