Local search in model checking

A.W. Roscoe and P.J. Armstrong and Pragyesh

Oxford University Computing Laboratory
{Bill.Roscoe@comlab.ox.ac.uk}

Abstract. We introduce a new strategy for structuring large searches in model checking, called *local search*, as an alternative to depth-first and breadth-first search. It is designed to optimise the amount of checking that is done relative to communication, where communication can mean either between parallel processors or between fast main memory and backing store, whether virtual memory or disc files. We report on it in the context of the CSP refinement checker FDR.

1 Introduction

Recent years have seen an enormous increase in the power of model checking technology, thanks in part to the more powerful computers that we now have to run them on and in part to improved algorithms such as techniques for SAT checking [2, 8], partial order reduction [14] and state-space compression [13]. It is clear, however, that there is still a need for the basic function of searching through the states of a large but finite automaton to test whether each reachable state is satisfactory.

The first author has yet to find a problem where FDR, the refinement checker for CSP (using explicit searching combined with the compression of subprocesses) could not prove a property (i.e. the absence of a counter-example) faster than the SAT checking models of [10, 16]. Even when finding counter-examples, the option of using FDR in DFS mode is very competitive with these, as shown in [10].

It is well known that the state spaces of parallel systems tend to grow exponentially with the number of parallel components, and also grow rapidly with the sizes of the types used in defining systems. This means that many of the examples one might wish to run on a tool such as FDR either take only a trivial time or are well beyond the bounds of possibility. What we are going to investigate in this paper is the region between these two extremes where, for example, either it becomes desirable to split the effort of a particular run across several CPUs, the number of states needing to be stored exceeds the limits of fast memory on the computer, or both.

In any case it is clear that for the foreseeable future there will be demand for tools that can handle as large as possible a system as quickly as possible. In what follows we will examine how model-checking technology for explicitly searching through a state space has evolved with time, and has been affected by the developments in computing technology over the last two decades. In this

last respect technology has been hugely positive and is alone responsible for the complete transformation of the capabilities of FDR and other methods. However it is also true to say that, relative to the huge increase in processing power these years have brought, the rate at which large amounts of data can be moved around, in particular on and off backing store such as disc, has not developed so rapidly on typical workstations. There has therefore been a gradually increasing need for search algorithms that minimise and optimise the needs for such data shifting. One particular statistic that is immediately visible to the user of a tool like FDR is the change in speed that occurs when it comes to occupy more space than is available to it in fast memory. We will refer to this as the *slow down* factor: 1 implying none, 2 meaning that it goes to half speed, and so on.

A crude measure of the slow-down factor is the reciprocal of the proportion of CPU time that the search tool gets when operating on backing store. (In the authors' experience they always use 100% when not fettered in this way.)

This paper presents techniques that we believe will greatly improve this factor. It represents work in progress in the sense that only relatively primitive and experimental versions of our methods have been implemented at the time of writing. The relative importance and effectiveness of the heuristics we introduce will only become clear after a good deal more work.

The main concepts of local search were proposed by the first author, and developed by all three during a period when the third author was an intern in Oxford during the summer of 2009. We expect that an implementation of it will be released in version 2.91 of FDR towards the end of 2009.

In the next section we review the background to the problem, and see how model checking algorithms have evolved in relation to this slow-down, and how both have been affected by the developments in computing technology. We then introduce the main ideas of local searching, before concentrating on the problem of how to partition state spaces effectively. We then review the results we have obtained to date before reviewing how local searching will transfer to parallel architectures.

This paper focuses on FDR. While we describe those parts of its behaviour directly relevant to this paper, inevitably we leave out a large amount of related detail. The interested reader can discover much more in, for example, [4, 9], [11] (especially Appendices B and C) and [12] (forthcoming 2010).

Acknowledgements

We are grateful for discussions with Michael Goldsmith. The computing support staff at Oxford University Computing Laboratory were very helpful by providing the machines with a *small* amount of memory that we requested. This work has benefited from funding from the US Office of Naval Research and EPSRC.

2 Background

Early model checkers, including the first version of FDR, stored state information in hash tables. To discover if one of the successor states S' of the current state

S you are examining has been seen before, just see if it is in the table. This works well provided the hash table will fit in the main memory of the computer, but extremely badly otherwise because of the way in which hash tables tend to create truly random access into the memory they consume: a section of backing store that is fetched into main memory (and this happens in large blocks) is unlikely to be much used before being written back.

This problem generated a number of ingenious techniques such as *one bit hashing* [5, 6, 15] in which one did not worry about hash collisions (so the search one performed was not complete). In this, only one bit is allocated per hash value, which simply records whether a state with that value has been seen before.

Such techniques were never implemented in FDR1, which saw slow-down factors approaching 100, making it entirely unusable for checks exceeding the bounds of RAM. The release of FDR2 in 1994 saw the first introduction of a searching technique in which, instead of performing each membership test individually, they are grouped together into large batches in such a way that the stored state space can be accessed effectively, with all checks against a particular block of stored state space being performed together.

The specific technique used by early versions of FDR, and described in [9], was to store the entire explored state space in a sorted list. The search was then performed in breadth-first search order, so that all the membership tests of each level of the check can be done together as follows:

- Initially, the explored state list $E(0)$ is empty and we need to explore the root state r. The initial *ply* of the search $Ply(0)$ is therefore $\{r\}$.
- Each ply $Ply(n)$ is sorted and combined with the explored states $E(n)$ by merging to create $E(n+1)$, by adding only those members of $Ply(n)$ not in $E(n)$. $Ply(n+1)$ is created as the set of all successors of these added states.
- The search is over when $Ply(n)$ is empty.

At the same time FDR started to apply compression techniques to these sorted lists, storing them in blocks that were either delta-compressed (i.e. only the bytes that were different between consecutive states were stored, together with a table showing where to put them), or compressed using the `gzip` utility, which was more expensive in terms of computing, but created compressed files that were perhaps 30% smaller than delta-compression.

When this was implemented the typical slow-down was reduced to somewhere between 2 and 4, so this represented a considerable success. In many cases the more aggressive compression regime gave lower execution times for complete checks.

The fact that disc access speed did not keep up with processor speed had the effect, however, of gradually increasing the slow-down factor between 1994 and the early 2000's, so that by this time the typical slow-down associated with this method had increased to perhaps 8-12. This was particularly noticeable towards the end of a refinement check, when relatively small numbers of new states are introduced per ply of the BFS, but the above algorithm was still ploughing through the entire state space each time.

To counter that, from FDR 2.64 onwards, the simple sorted list structure was replaced by a B-tree, still a sorted structure, but where it is possible to gain rapid access to a particular state and to omit whole blocks of states that are not required in a particular pass. Thus the merge of the algorithm reported above was replaced by repeated insertion into the B-tree represented by the already-explored states. This was particularly effective towards the end of searches, and perhaps halved the slow-down factor overall.

Of course this was not as good as we would have liked, and we are naturally now once again seeing an increase in this factor. It is this issue that the present paper addresses: we are looking for ways to reduce the amount of data that has to be moved to and from memory for each state explored.

Although the memory size of modern workstations is huge compared to those of a few years ago (2-4Gb being typical at the time of writing) it nevertheless seems to be the case that this limit is reached more quickly by FDR than in previous years. This means that, if anything, the problems of a high slow-down factor has become greater.

Believing that the new technology of disc drives built from flash memory would help greatly with the problems identified here, we obtained such a machine in 2008. We were disappointed that, at least with that particular version, the performance was actually marginally worse than with the same machine's conventional disc. This led us to believe that, though this type of technology may well improve in the future, it is very unlikely to solve the slow down problem sufficiently.

As we will discuss later, may of the same issues also apply to the parallel execution of model checking, where a significant barrier may be amount of data that has to be communicated between processors.

3 Local search

As we have seen, BFS permits all of the membership tests in each ply of the search to be combined, which greatly reduces the amount of memory churn that is required. Nevertheless, even when improved by B-tree structures, it still more-or-less requires the entire accumulated state space to pass through the CPU on each ply. This would not matter if the memory bandwidth were sufficiently high that this happened without diminishing performance, but this is not true.

It gradually became apparent to the first author that the present BFS strategy required improvement, and so he proposed the following alternative method:

- Begin the search using the same BFS strategy as at present, with a parameter set for how much physical memory FDR is supposed to use. Maintain statistics on the states that are reached, the details of which will be discussed later.
- Monitor the position at the end of each play of the BFS, and when the memory used by the combination of the accumulated state space $X(n)$ and the unexplored successors $S(n+1)$ exceeds the defined limit, these two sets are

split into a small number (at least two) *blocks* by applying some *partitioning function*.
- This partitioning might either be done as a separate exercise, or be combined with the next ply of the search.
- From here on the search always consists of a number of these blocks, and a partitioning function Π that allocates states between them. Each such block $B(i)$ will have an explored state space $EB(i)$ and a set of unexplored states $UB(i)$, which may intersect $EB(i)$.
- At any time FDR pursues the search on a single block B only, as a BFS, but the successors generated are partitioned using Π. The successors belonging to $B(i)$ are pursued in this BFS. Those belonging to other blocks are added to $UB(j)$ in a way that does not require operations on the existing body of $UB(j)$ that require this to be brought into memory.
- The search on $B(i)$ might terminate because after some ply $UB(i)$ becomes empty. In this case begin or resume the search on some other block with a nonempty $UB(j)$: if there is none, the search is complete and in FDR the refinement check gives a positive result. If there is another block to resume we must realise that $B(i)$ may not be finished for all time since other $B(j)$ might create a new $UB(i)$ for the block that has just terminated.
- On the other hand the size of the search of $B(i)$ might grow beyond our limit. In this case Π is refined so that $B(i)$ is split as above. After that a block which may or may not be one of the results of this last split is resumed.
- On resuming a block $B(i)$ the set $UB(i)$ might have grown very large thanks to input from other blocks, and furthermore it is possible that there might be much repetition. There is therefore a strong argument for (repetition removing) sorting of $UB(j)$ before assessing whether to split $B(i)$ at this stage.

It is clear that this search will find exactly as many states and successor states as either BFS or DFS: we explore the successors of every state exactly once. The objective of this approach is to minimise the amount of transfer in and out of main memory during the search. The basic rationale is that it makes sense to explore the successors of some states without always backgrounding them before doing so. In other words,

- Like DFS, tend to explore states that are already in the foreground.
- Seek to reduce the amount of memory churning performed by BFS.

There are three major parameters to this search method:

- How many pieces to divide blocks into? There are clear arguments for making this depend on the evolution of the search. If it is growing quickly we would expect a larger number of pieces to be better.
- Where there is a choice of blocks to resume, which one one should one pick? One could, for example, pick the largest one, follow a depth-first strategy (where blocks are arranged into a stack, with new blocks being pushed onto the stack, and the top of the stack being chosen for resumption), a breadth

first search (where they are organised into a queue), or a hybrid in which, when a split occurs, we pursue one of the resulting blocks but push the rest to the back of a queue. Suppose, for example, an initial split generates blocks A and B, pursuing A generates AA and AB, and that pursuing AA at that point means that it terminates before splitting. Then the three strategies would initially follow $\langle A, AA, AB\rangle$, $\langle A, B\rangle$ and $\langle A, AA, B\rangle$.

- Perhaps most importantly, what algorithm should we use for partitioning the state space? There are two desiderata here: firstly that the parts we divide the state space into are the sizes we want. While we might well want them to be equal in size, this is not inevitable. The second objective is that as large a proportion of the transitions from a state in each block should be to the same block: that way there will be relatively little state space to transfer to other blocks, and the search of a generated block is unlikely to peter out quickly. We will term this property that transitions tend to be to "nearby" states as *locality* and measure it as a percentage: the percentage of computed transitions that lie inside a single block. It follows that we should hope to do better than we would with a randomised partition.
  How successful we are with this second objective might well influence the other choices that have to be made.

Whilst we hope that this approach will give us advantages thanks to better memory management, we also need to be aware of the extra work it introduces.

- Based on the above, we would expect each state to have partitioning functions applied to it as often as the blocks it happens to be in are split.
- Similarly we would expect there to be a cost in re-organising the data structures used to store states each time a block is split.
- There will also be costs in devising partitioning functions and collecting and analysing statistics to help in this.

4 Partitioning the state space

We will assume in what follows that the overall state space is stored in a sorted structure in the general style of FDR as described earlier. We do not, however, discount the possibility that the same ideas might work in conjunction with hash tables.

Given this choice, there is an obvious way of partitioning a state space quickly: to break it into k pieces choose $k-1$ states as pivots, with the pieces being the $k-2$ sections between consecutive pairs of pivots, and the other two being the those less than, and greater than, all pivots. We might conventionally include the pivots themselves in the section immediately above them.

The most obvious way of picking these pivots is to spread them evenly either in the relevant $UB(i)$ or $EB(i)$ or both. The most obvious advantage of this approach is that the partitioning and restructuring work essentially disappears.

A clear disadvantage is that there is no good reason to suspect that this partitioning approach will give good locality properties: for this to happen we

would need to design an order where most transitions are to states that are close (in the order) to their origins. We will show later how this can be achieved.

If we are to look for ways of improving the locality of a partition we must understand how one's tool, in our case FDR, represents states and generates transitions.

Since FDR is checking the refinement of a "specification" process *Spec* by an 'implementation" process *Impl*, the things it searches through are not single states but actually "state-pairs": the pairs (ss, is) where ss is a state of the specification and is is a state of the implementation that are reachable on some trace t.

ss is an integer index into an *explicit* representation of the *normal form* of *Spec*. An explicit state machine is an array of states, each of which carries a list of transitions, with each transition being the label of the action and the index of the target state. In some cases the states might have additional labelling. A normal form state machine is one in which (i) there are no τ (invisible) actions and (ii) no state has more than one action with any given label. Thus given the root state of a normal form machine, and a trace, then if it can perform the trace t then after t its state is completely determined.

If the refinement check is over any model other than $\mathcal{T}$, the traces model, then the normal form state will be labelled with information representing, for example, divergence and refusal-set information.

In most cases FDR computes the normal form in advance, but there is also the option to use the function `lazynorm(Spec)`, which means that only those normal form states relevant to traces of *Impl* are explored. The motivation for `lazynorm` is that sometimes the normalisation of *Spec* is a time-consuming activity, and `lazynorm` means that only the necessary parts of the normalisation are done. This is potentially an important distinction for our partitioning activities, since preliminary analysis can be carried out on a complete *Spec*, but less on one that has yet to be evaluated.

In a large proportion of FDR checks, the specification is extremely simple, frequently having just one or two states. For example, the specifications for "deadlock-free" and "the event a never occurs" each have just one state. There is clearly very little potential for partitioning the space of state pairs based on the specification state in cases like this[1].

The situation with the implementation state *is* is analogous to that with lazy normalisation: at the start of the check we have no idea which states will be reached or what their transitions will be. They are both examples of *implicit* state machines: a representation of the root state together with a set of recipes for computing the actions and resulting target states from any given state.

To understand the structure of an implementation state we need to understand the two-level treatment of CSP's operational semantics in FDR. The

[1] It is possible in these cases artificially to increase the number of states in the specification, and indeed in our trial implementations we have sometimes done this. There are various good reasons such as loss of compression and the difficulty of automating this process that make this option very unattractive as a long-term solution.

CSP_M input language of FDR allows the user to lay out a network using functional programming. Broadly speaking FDR will reduce (in functional programming terms) the description of a network until it either hits a true recursion or gets below the level of all "high-level" operators, the most important of which are parallel, hiding and renaming. The syntax it encounters below that dividing line is compiled into explicit state machines using a symbolic representation of the operational semantics: these must be finite state and will be called *components*. Above that level it devises one or more recipes for combining together the states and actions of the components into states and actions of the whole implementation. We call these recipes *supercombinators*: see below.

Components do not only arise from compiling low-level syntax. They can also be generated by one of the various operators in CSP that compresses the state space of a process, typically because it has been applied to the parallel/hiding combination of a proper subset of what would otherwise have been the components of the complete system. Components can vary in size from a single state to hundreds of thousands or even millions.

Each of the recipes for combining components is called a *format*. In the majority of practical checks, there is just a single format that is effectively a parallel composition of the N (say) components, with perhaps some hiding and renaming mixed in.

Where there is more than one format, it is because one or more CSP operators that are usually low level get pushed up to high level by having some parallel operator beneath them in the reduced syntax tree, as in

$$(P \parallel Q);\ R \quad \text{and} \quad (a \rightarrow (P \parallel Q)) \,\Box\, (b \rightarrow R)$$

In both of these there will be format that consists of a state of P and one of Q, and a format that consists of a state of R. In the right-hand process there is also a format representing the initial state that has no components.

FDR already stores the different formats separately, but where there are more than one they will frequently have very different numbers of states. So while it may well make sense to incorporate formats into a partitioning strategy, they are unlikely ever to be an important part of it. For the rest of this section, therefore, we will concentrate on partitioning an implementation process that has only a single format.

FDR calculates the actions of such a state in one of two ways. Firstly, any τ action of one of the components is promoted to be an action of the complete system, in which only the component that performs the τ changes state. Secondly, FDR produces a number of *supercombinator rules*. Each of these can be described by a partial function ϕ from the component indices to Σ, the visible event labels, together with a result action, which may either be in Σ or τ. This rule can fire just when, for each $i \in dom(\phi)$, component i can perform the event $\phi(i)$. Just the components that are in $dom(\phi)$ change state.

Thus each component processes simply moves around its own state space as the complete implementation progresses. One of the best prospects for partitioning the complete state space is to partition one of the components, and

simply assign each state or state pair to a group determined by what state that component is in. This can clearly be generalised to look at the states of a small collection of the components, particularly in the case where the individual components each have very few states.

The mapping that sends each overall state to the state of the jth component may be far from uniform: some states may be represented many more times than others. It follows that what seems like a well-balanced partition of this component may not yield a well-balanced partition of the complete space. It is not at all unusual, for example, for a component process to have one or more error states that we hope will never be reached in the entire system. It therefore seems unlikely that there will be good *automatic* ways of deciding what component-based partition will be used in advance of running the search. We will, however, later show how to create a range of *options* for this in advance.

In our single format case, where the format has N components, we can think of each state pair as an $N+1$-tuple of states, the extra one being the specification. There is no reason why partitions cannot be based on the specification state just as for a component of the implementations, but nor is there any reason to expect specification state distributions to be any more uniform.

We offer two different strategies for partitioning the state space, which we term *pre-computed* and *dynamic*.

4.1 Pre-computed partitioning

In pre-computed partitioning, we decide on some partitioning options in advance, based on the state spaces of one or more of the $N + 1$ processes that represent a state pair.

We need an algorithm which will take a state machine and divide it into two or more state machines that are relatively self-contained, in the sense that as few transitions as possible pass between these parts.

This algorithm will come in two parts. The first part will assign weights to the various nodes and transitions of the component state machine. These will assess how likely it is that the machine will be in a given state or take a given transition. The second part will be to choose partition(s) of the state machine that attempt to deliver roughly equal node weight in each part and minimal edge weight between them so as to improve the locality of the search.

Each of these two algorithms will inevitably only give approximations to optimal results. In the first case this is because we cannot predict the weights that apply to a given search without running the search, and in the second case because even if we could formulate the correct balance between the two criteria, it would almost certainly be NP-hard to optimise them. Since the size of components we will want to partition will vary from 2 to millions, we will have to choose extremely efficient algorithms, place an upper bound on the size of component we will apply our algorithms to, or have different algorithms for different sizes of component.

This is not the place to go into great detail about these algorithms, but we give a few ideas below.

Weighting algorithms In the absence of evidence about how a system behaves in practice, we can make intelligent guesses about how often particular transitions and nodes are used.

The state explosion problem arises because potentially, and frequently, the state spaces of the component processes multiply to give the state space of the entire system. Though in many cases this is not literally true, it is hard to think of a general way other than running the search to identify statistical information about which of these potential states are reachable and which are not. What we will therefore do is to build up a stochastic model built on the assumption that all combinations of component states are reachable. We exclude the specification from the computation of this model since it participates in *every* visible action.

What we will estimate is the probability, for each state γ of each component i of the system, that if the ith component of a global state Γ is γ, and Γ's other components are random, a randomly chosen action of Γ leads component i being in each of γ and the states immediately reachable from it. The sum of the probabilities of those states that are in the same partition as γ provides a measure of the locality of the actions starting from a randomly chosen Γ, conditional on this ith component.

We will call γ an *i-state*. We note that it might stay in γ either because it took an action that did not change the state, or because the global action did not involve component i.

For each labelled action x that each component j can make, we can compute $E_j(x)$ the expected number of times j can perform x from a random state: this is simply the number of x actions from its states divided by the number of states i has. Note that some of the states might have more than one x, so this expectation can be greater than 1.

We can therefore calculate, for each supercombinator $\rho = (\phi, x)$ the expected number of times $E(\rho)$ it can fire in an random state, namely the product of $E_j(\phi(j))$ as j ranges over $dom(\phi)$. We can similarly calculate $E_j(\rho \mid \gamma)$ the expected number of times that ρ can fire given that the i-state is γ.

- If $i \notin dom(\phi)$, it is $E(\rho)$.
- If $i \in \mathrm{dom}(\phi)$, then it is the product of the number of $\phi(i)$ actions γ has and all the $E_j(\phi(j))$ as j ranges over $\mathrm{dom}(\phi) - \{i\}$.

The total number of actions $EA_i(\gamma)$ we can expect a random state Γ whose i-state is some fixed γ to perform is

$$ct(\tau, \gamma) + \Sigma\{E_j(\tau) \mid j \in \{1..N\} - \{i\}\} + \Sigma\{E_i(\rho) \mid \rho \in SC\}$$

where $ct(x, \gamma')$ is the number of different x actions that the state γ' can perform from its initial state.

The expected number of times $E1_i(x \mid \gamma)$ that component i has a single one of its x actions enabled is then

- 1 if $x = \tau$.
- If $x \neq \tau$, it is the sum, over all supercombinators (ϕ, y) with $\phi(i) = x$, of the product of $E_j(\phi(j))$ as j ranges over $\mathrm{dom}(\phi) - \{i\}$.

We can now say that the probability of a specific action (x, γ') of γ firing from Γ whose other components are randomly chosen is $E1_i(x \mid \gamma)/EA_i(\gamma)$, namely the number of times we expect it to fire as an action of such a Γ divided by the total number of actions of Γ. If $EA_i(\gamma) = 0$ then it is also 0.

The probability $PT_i(\gamma, \gamma')$ of γ making a transition to $\gamma' \neq \gamma$ is therefore the sum of this value over all labels x such that $\gamma \xrightarrow{x} \gamma'$. The probability the state is unchanged is

$$PT_i(\gamma, \gamma) = 1 - \Sigma_{\gamma' \neq \gamma} PT(\gamma, \gamma')$$

which takes into account the possibility of some x with $\gamma \xrightarrow{x} \gamma$ firing, and component i not being involved in the transition.

The above model depends crucially on our simplifying assumption that all states of the Cartesian product of the component state spaces are reachable. Under that assumption we know that every state of a given component i is reached equally often, so there is no point in trying to assign weights to these states representing how often each is represented in the complete system. There may, however, be reason to assign weights to them which include the number of times we expect that state to appear as a successor in the search, since that will affect the memory consumption of each block. This is because some component states may be represented more often than others in successors, even though they all appear equally often in the final state space. We do not propose a specific method at this time.

4.2 Example

In Chapter 15 of [11], the first case study given is peg solitaire, see Figure 1. We have added the letters A–G to show typical examples of the seven symmetry classes of slot under rotation and reflection. Games like this are useful benchmarks for FDR because they demonstrate the complexity of the problems being solved, because they usually give a counter-example (which is much more appealing than demonstrating an example where refinement holds) and because in FDR's standard breadth-first search the counter-example is found (as in solitaire) at the very end of the search, meaning that the search profile is almost identical to a check that succeeds. The model consists of 33 two-state processes, one representing each peg. A solitaire move is the in one of the directions *up, down, left* and *right*, and hops a peg over another peg into an empty slot. The complete alphabet has 19 moves in each of the four directions, plus a special event *done* that all the processes synchronise on when they reach their target state. The diagram shows the initial configuration, and the target is the opposite one – a single peg in the middle.

None of the processes can perform a τ. The nine central slots (in classes E, F and G) can each perform 12 different moves: they can be hopped out of, over, or into in any of the four directions. The twelve slots adjacent to them (classes C and D) can each perform 6 different move events, and the twelve slots around the edges (A and B) can each perform only 4. Since there is one supercombinator

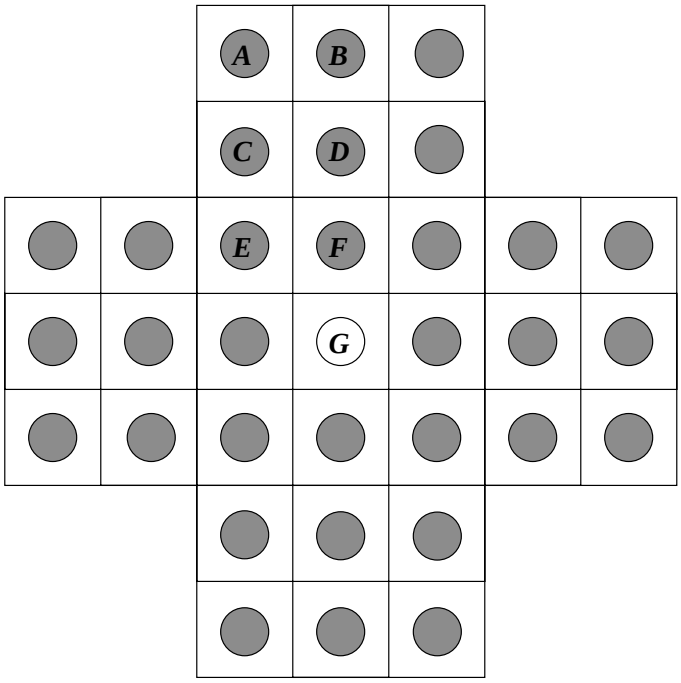

Fig. 1. Solitaire board with representatives of symmetry classes

for each event – the one that demands that each of the 3 or 33 processes that can perform the event do so – and since only one of the two states of each event can perform it, our probability model calculates the probability of each move event being enabled as 2^{-3} and the probability that *done* is enabled as 2^{-33}. In other words $E_i(x) = 0.5$ for every event x and slot i.

The model predicts that the probability of any component process being involved in an arbitrary transition is close to the number of moves it is involved in, divided by the total number of move (76). This model, of course, is based on the assumption that all the 2^{33} states are reachable[2] whereas in fact $187,636,299$ are (about 2.2%). (

By symmetry of the puzzle, it follows immediately from the fact that not all reachable configurations can be carried forward to a solution that more of the states will be encountered beyond the half-way point (i.e the 16th move of 31) than before it. This means, that $E_i(x)$ is, on average, less than 0.5 since each move requires two pegs present and one absent. Thus it is not surprising that

[2] As discussed in [1], the 2^{33} states break down naturally into 16 equal-size equivalence classes such that every reachable state is in the same class as the starting one. Thus the 2.2% reachability actually represents about 35% of the equivalence class of the starting state.

the actual number of successor states found (1,487M) is less than the 1,781M that the model of $(76/8) \times 187M$ suggests. While the model suggests that all 76 moves will be enabled roughly equally often, in fact they vary from 12.4M times to 24.1M.

Both the model and the statistics from the check itself suggest that the outer pegs (classes A and B) are the best ones to partition on, since they change state in the smallest proportion of the transitions.

Partitioning a weighted graph Now suppose that we have computed, for each component i, a *node weight* for each state γ, representing the proportion by which we expect to encounter Γs with this γ during the search, and an *edge weight* on each pair (γ, γ') such that $\gamma \neq \gamma'$ and γ' is reachable in one step from γ. The edge weight represents the probability that, given that the ith component of Γ is γ, a successor Γ will have γ'.

The previous section we assumed that all node weights are equal, and that the edge weights are $PT_i(\gamma, \gamma')$. We will suggest some more possibilities later. In a case where node weights are not equal, then the edge weights from a state γ should be an estimate of the product $w(\gamma)$ and the probabilities of the various actions *given* we are in γ.

In partitioning component i into roughly equal weight but localised parts (localised meaning that most of its transitions lie within the part), there is a tension between these two objectives. In general, of course, we will want to divide a component machine into more than two pieces, or two be able to subdivide a machine that has already been partitioned. Some possibilities are as follows.

- We could deploy algorithms for finding a *minimum cut* between two regions of the state machine that have been identified an have non-trivial node weight. An example of this (and the one we have deployed in our preliminary implementation) is the Kerninghan-Lin algorithm [7].
- We could successively remove inter-node edges in reverse order of weight until the machine becomes disconnected, again with restrictions to ensure the pieces do not become too large.
- Starting from a number of well-spaced points in the graph of nodes, build up components C_i. Successively add a node to the lightest C_i: the node for which the sum of edge weights between C_i and it (both ways) is maximised.

Where we have partitioned a component into more than two parts $C_1, \ldots, C_n$, whatever algorithm we use for the above task should also generate a linear order on the C_i with a view to each of the $\bigcup_{i=r}^{s} C_i$ also making sense in terms of locality. One way of achieving this is to join together the two C_i with the largest (sum of edge weights) connection between them, and then successively add on another whose connection at either end makes most sense.

To follow the pre-computed partitioning algorithm, we then arrange the components into order by the quality of the partitioning, judged in some way. Suppose the jth part (in its linear order) of the ith component process (ordered in decreasing order of quality) is $C_{i,j}$. We then define the *partitioning function* Π, when applied to a state Γ to be a tuple whose ith component is j, where the ith component $\gamma(i)$ of Γ lies in $C_{i,j}$.

The range of Π is naturally linearly ordered by lexicographic order (i.e., the first component takes precedence, then the second, and so on).

Recalling that FDR stores the accumulated state space in a sorted structure, we can make $\Pi(\Gamma)$ the primary sorting key for the states Γ. Then to partition the state space all we have to do is find the point(s) in the range of $\Pi(\Gamma)$ which is closest to giving the desired sizes of the sections, or (probably) better, find points determined by as few components $\gamma(i)$ as possible that are within some tolerance of the desired division points.

The effect of this technique is to pre-compute many potential ways to partition the state space, and then structure the search so that it is trivial to split either B-trees or sorted lists of states.

Only experience can tell us how effective the probability-based weight calculations set out above are. It is reasonable to expect that we will find methods that are effective for searches that are in some sense "typical". It is also very likely that it will be possible to construct possibly pathological examples for which it works badly.

There are two alternatives to this: one is to use Monte Carlo methods to conduct a partial exploration of the state space in advance, simply to be able to give estimates for the node and edge weights of the components. Using this is will still be possible to use the partitioning function Π that stays constant during a search. The other alternative is to use a dynamic partitioning method.

4.3 Dynamic partitioning

We have seen that there is a significant data structuring advantage in using a pre-computed, linearly ordered, partitioning function. There are also the disadvantages that we have to work to give predictions about the structure of the check, and that we are committed to using the $N + 1$ component processes in a fixed order regarding partitioning.

The alternative is to gather information about the node and edge weights of the various components as the check progresses: we can actually count how many times each of the states of these processes is represented in the accumulated state space, and how often each of their transitions is possible in these states.

This is very easy and relatively cheap to collect during a search, and we can then use whatever part of it we wish when a particular block of states has to be partitioned. A possible approach to this is, for each block that is searched, to collect information about those states in it that arose since that block either began as the initial one with the root state, or was last split.

We should, however, notice that the statistics from the part of the check already done may not be a good predictor of the future. For example, in the solitaire example, we would expect that the probability that any given slot is empty will increase during a run.

It is likely that in many cases it will be possible to choose a better-performing partition of such a block, probably based on only one component, that using the fixed function.

The algorithms for choosing the best ways to partition a given component process will, however, follow the same lines as those above.

Whether using dynamic or pre-computed partitioning, we always have the option of deciding how many pieces to break a block into at the point when the split is made. We note that it may be wise to estimate the eventual size of the block to be partitioned in order to guide this decision.

5 Preliminary practical results

There are two main options as to how to use FDR in respect of memory management. The default mode is to have it run as a simple UNIX process, storing everything as part of process state. Typical Linux implementations limit the size of a process to 2Gb or 4Gb. A modern processor typically fills this up in less than an hour when running FDR, in many cases without needing backing store at all.

The other option is via `FDRPAGEDIRS` and `FDRPAGESIZE`, environment variables which direct the tool to store blocks of states, whose size are determined by the latter, in directories specified by the former. The latter, naturally, represent backing store, but of course in many cases these are buffered in main memory.

Our main experiments have been based on two classes of example: variants of the peg solitaire puzzle discussed earlier, and the distributed database example set out in Chapter 15 of [11].

Our implementations to date use a partitioning algorithm based on the specification process only, without edge or node weighting calculations of the sort set out earlier. Nevertheless they have shown reasonable locality on the above examples: 93% (of states leading to same partition) in the database example, and 84% for solitaire.

They have shown considerable promise in speeding up large checks, in particular in the database example where we found better locality, but are still considerably sub-optimum in the way they handle the sets of states that are passed around between blocks.

For this reason we have decided not to publish a performance table in this version of this paper, but will instead include one in a later version to appear on the web.

6 Parallel implementation

A parallel version of FDR was reported in [3], the *modus operandi* of which is to allocate states between processors using a randomising hash function whose range is the same size as the number of processors, and which implements the usual BFS by having the processors "trade" successors at the end of each ply.

Though its performance was excellent in terms of speed up (linear in most cases), this parallel version has never been included in the general FDR release

because its usability would be restricted to just one of the many parallel architectures in existence. It is clear, however, that a parallel version for at least multi-core architectures is now required.

The concepts of local search clearly transfer extremely well to parallel execution, and offer the prospects of reducing the amount of communication between processors and reducing the amount of time that one process will spend waiting for another. One possible implementation would be to use the idea of *processor farming*: keep a set of search blocks in a queue and, each time a processor becomes free, give it the next member of this queue.

The effectiveness of local search versus BFS, and the most efficient way to use it, is likely to differ between parallel architectures. Let us examine the difference between a cluster of processors with independent memory and discs, and multi-core architectures that share these things. It is, of course, likely that we will have two-level architectures consisting of clusters of multi-core processors.

- In a shared-resource multi-core architecture we should have very fast inter-core communication. On the other hand the imbalance (from the point of view of model checking) between processor power and fast memory (i.e. the problem that local search is designed to alleviate) will simply be multiplied when running a check on more than one core, and disc bandwidth-per-core will be divided. It may therefore prove a good idea to concentrate on high locality between the *set* of blocks currently being explored on all the cores, and the set of blocks presently stored on disc. There might also be an argument for performing different aspects of the search on different cores, though that might be harder to balance and scale.
- The behaviour of a cluster of processors is likely to be governed by the relative rates at which data (transfers of states) is passed between processors, and the speeds at which the processors can (a) transfer data in and out of their own disc store and (b) process the results.

Our initial aim will be to release a multi-core parallel implementation, hopefully by the end of 2009.

7 Conclusions

In this paper we have analysed the issue of keeping the usage of slow forms of memory down to the level where this does not force processing to be delayed, and seen that this challenge has gradually increased as processing capability has grown faster, and will increase further with the number of processor cores.

We have developed the *local search* strategy in an attempt to solve this problem, which is crucially dependent on our ability to partition the state space into pieces whose transitions are primarily *local*, namely to other members of this partition.

We have shown that by analysing the transition patterns of the component processes that are combined by FDR generate the transitions of a particular

refinement check, there is a good prospect that we can choose good partitioning algorithms.

Since local search is not BFS it removes the guarantee that the first counterexample it finds will be a shortest one. There is, however, still a choice akin to the distinction between DFS and BFS that we have to make, namely in which order do we process the outstanding blocks of our search. We discussed three possibilities for this earlier.

The next state of our work will be to implement these possibilities and a variety of partitioning algorithms in FDR and experiment with their effectiveness. Our objective is to provide a limited range of options so that the user does not have to understand all of this technology to use it.

References

1. J.D. Beasley, *The ins and outs of peg solitaire*, Oxford University Press, 1985.
2. N. Een and N. Sorenson, *An extensible SAT solver*, Proc SAT 2003.
3. M.H. Goldsmith and J.M.R. Martin, *The parallelisation of FDR*, Proc Workshop on Parallel and Distributed model Checking, 2002.
4. M.H. Goldsmith and others, *Failures-Divergences Refinement (FDR) manual*, 1991-2009.
5. G.Holzmann, *An improved reachability analysis technique*, Software P&E **18**(2), pp137-161, 1988.
6. G.Holzmann, *Design and validation of computer protocols*, Prentice Hall, 1991.
7. B.W. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, Bell System Tech. Journal, **49**, pp. 291-307, 1970.
8. K.L. McMillan, *Interpolation and SAT-based model-checking*, Proc CAV 2003.
9. A.W. Roscoe, *Model checking CSP*, in 'A classical mind: essays in honour of C.A.R. Hoare', Prentice Hall, 1994.
10. H. Palikareva, J. Ouaknine and A.W. Roscoe, *Faster FDR counter-example generation using SAT solving*, To appear in proceedings of AVoCS 2009.
11. A.W. Roscoe, *The theory and practice of concurrency*, Prentice-Hall, 1997.
12. A.W. Roscoe, *Understanding concurrent systems*, Springer, Forthcoming 2010.
13. A.W. Roscoe, P.H.B. Gardiner, M.H. Goldsmith, J.R. Hulance, D.M. Jackson and J.B. Scattergood, *Hierarchical compression for model-checking CSP* or *how to check* 10^{20} *dining philosophers for deadlock*, Proceedings of the 1st TACAS (1995), Springer LNCS 1019.
14. A. Valmari, *Stubborn sets for reduced state space generation*, Proceedings of 10^{th} International conference on theory and applications of Petri nets, 1989.
15. P.L. Wolper, Pierre and D. Leroy, *Reliable hashing without collision detection*, Proc. CAV 1993.
16. J. Sun, Y. Liu, J. S. Dong, *Bounded model checking of compositional processes*, Proc. 2nd IEEE International Symposium on Theoretical Aspects of Software Engineering. Pp. 23.30. IEEE, 2008.