On the Implementation of MIPS

Stefan Edelkamp and Malte Helmert
Institut für Informatik
Am Flughafen 17
D-79110 Freiburg
{edelkamp,helmert}@informatik.uni-freiburg.de

Abstract

Planning is a central topic of AI and provides solutions to problems given in a problem independent formalism. Recent successes in the exploration of model checking and single-agent search problems have led to a generalization of the symbolic exploration method with binary decision diagrams (BDDs).

In this paper we present the use, architecture, implementation and performance of our STRIPS planner MIPS abbreviating *intelligent model checking and planning system.*

With BDD refinements, symbolic and single state heuristic search engines we highlight recent improvements that have been added to the system.

Introduction

In classical planning an *object* is a unique name for an entity in the modeled domain and *predicates* are unique names for modeling attributes of objects. Instantiated predicates are *facts*. A *state* in the planning problem is a set of facts. An *operator* is a 3-tuple (P, A, D) of *preconditions*, *add-effects* and *del-effects*, with P, A and D being sets of facts. An operator is applicable, if all preconditions are fulfilled. The state is altered by adding the corresponding add-effects and removing the del-effects. Operators or actions are defined by *schemas* that contain free variables, to be instantiated by the facts given in a state.

Given a set P of predicates, a set A of actions, a set O of objects, an initial state s and set of goal states G the planning problem is to find a sequence of operators that transforms s into g in G. The algorithm is *admissible* if the sequence of operators is the shortest possible. The algorithm is *complete* if it always terminates returning a plan or with failure in case no solution exists.

In this paper we give details on the implementation and current status of our planner MIPS. The internal structure of BDDs and their operators is not discussed. We use BDDs as a black box compactly representing and operating on Boolean functions in our planner. We start with SAT-based planning which leads to representation issues resolved by precompilation, and reachability analysis. We give some insights in the complexity of precompilation, exploration and solution extraction. Further on we present the architecture of the system MIPS 1.0 and its usage. Then we turn to new inventions in version 1.1 - version 1.5 of our planner. Finally experiments on the AIPS-98 problem suite and some concluding remarks are given. Throughout the paper we have included some code fragments. Even though the code is not self-explanatory and only scratched in the text it might give a feeling of the implementation.

Planning as Satisfiability

Planning as satisfiability (as given in *Satplan* (Kautz and Selman 1996)) formalizes the truth of fact f after t time steps with a variable $x_{f,t}$ in propositional logic. The initial state is valid in time step $t = 0$ and leads to the following conjunction: $\mathit{start} = \bigwedge_{f\in s} x_{f,0} \wedge \bigwedge_{f\in F\setminus s} \neg x_{f,0}$, where F is the set of all facts. Reaching the goal set G that is represented by a set of facts which must be satisfied in any goal state and is formalized as $\mathit{goal}(t) = \bigwedge_{f\in G} x_{f,t}$ Finally, applying an operator $o = (P, A, D)$ at time t is formalized as

$$\mathit{apply}(o,t) = \bigwedge_{f\in P} x_{f,t} \wedge \bigwedge_{f\in A} x_{f,t+1} \wedge \bigwedge_{f\in D} \neg x_{f,t+1}$$

$$\wedge \bigwedge_{f\in F\setminus(A\cup D)} x_{f,t} \leftrightarrow x_{f,t+1}$$

The entire problem specification is now given by

$$\mathit{Problem}(t) = \mathit{start} \wedge \ \mathit{goal}(t) \wedge \bigwedge_{0\le i<t} \bigvee_{o\in Op} \mathit{apply}(o,i)$$

If a satisfying instantiation of the formula for a minimal time step $t = t_0$ is obtained, one can extract the sequence of states $x_{f,t}$, $t \in \{0, 1, ..., t_0\}$ that transforms the initial state into one goal state. This state solution path immediately implies the operator solution sequence.

The problem with planning as satisfiability is that even for small problem the number of variables is high. Since the action schemas generate similar operators, many similarly structured formulae are generated. Therefore, a compact data structure representation of actions is required and the utility of BDDs is apparent.

The Underlying BDD-Package

At first we had implemented a BDD-Library *StaticBdd* for efficient concise index based depth first search *BDD* operations from scratch. In about 1K lines all necessary features such as a unique table, a cache, a free list, a compressed state description, a dirty bit garbage collector, a reduction integrated synthesis and a relational product algorithm were provided.

During the implementation process we changed the BDD representation to improve performance mainly for small planning examples. We chose a quite recent public domain BDD package called *Buddy* (Library Package version 1.6) by Jørn Lind-Nielsen additionally providing a finite domain variable interface.

The Planning Process

The straight forward encodings of states by a variable for each fact are too long to solve larger planning problems. The question how to transform the STRIPS notation into a concise Boolean representation of the states and operators has been resolved by precompiling the domain to minimize the state description length. The process of finding a state description consists of four phases (Edelkamp and Helmert 1999). In the first phase we symbolically analyze the domain specification to determine constant and one-way predicates, i.e. predicates that remain unchanged by all operators or toggle in only one direction, respectively. In the second phase we symbolically merge predicates and exhibit important domain-specific invariants which lead to a drastic reduction of state encoding size, while in the third phase we constrain the domains of the predicates to be considered by enumerating the operators of the planning problem. The fourth phase combines the result of the previous phases.

Representation

States are represented by their characteristic function ϕ, which evaluates to *true* if and only if the binary encoding of the state is met. Similarly, we can represent relations on states by enumerating all tuples. Therefore, the three main parts of a planning problem are given by boolean formulae: start state, goal states and the transition relation encoding all possible state transition in the problem.

The BDD representation of the state space allows to reduce the planning problem to model checking: An iterative calculation of Boolean expressions has to be performed to verify the formula **EF** *Goal* in the temporal logic CTL. The computation of a (minimal) witness delivers a plan.

Exploration

In the context of the above formalization given a consistent assignment to the variables $x_{f,t}$ (the valid states at time t) the set of consistent assignments of the variable $x_{f,t+1}$ (the valid states at time $t+1$) is calculated. If one of the sets contains a goal state a satisfying assignment to the entire boolean planning formula is found. In case of a BDD representation the assignments have to be extracted.

Since exploration is the most important part of the BDD-based approach in Figure 1 we give the C++-code of the search engine. The main loop switches from forward to backward search according to a boolean flag `forward`. The encountered BDDs for representing the breadth–first search layers are kept in vector structures provided by the standard template library STL. The search step itself depicted in Figure 2.

```
int search() {
  vector<bdd> forwardBDD, backwardBDD;
  forwardBDD.push_back(init);
  backwardBDD.push_back(goal);
  int iteration = 0;
  bdd meet = bddfalse;
  while(meet == bddfalse) {
    bdd &front = forwardBDD.back();
    bdd &back = backwardBDD.back();
    ++iteration;
    if(forward
    ? searchStep(front,back,meet,
                 preVar,prePair,forwardBDD)
    : searchStep(back,front,meet,
                 effVar,effPair,backwardBDD))
      break;
  }
  return iteration;
}
```

Figure 1: The (bidirectional) breadth–first search process.

```
void searchStep
    (bdd front,bdd back,bdd &meet,bdd varset,
     bddPair *rename,vector<bdd> &bddVec) {
  bdd current = bdd_relprod(front,trans,varset);
  meet = current & back;
  current = bdd_replace(current, rename);
  bddVec.push_back(states);
  bdd states = front | current;
  if (states == front) return true;
  return false;
}
```

Figure 2: One iteration step in the symbolic breadth–first traversal.

Note, that the application of boolean formulae and BDDs from one level to the next is executed in one step and not state by state, as in most single-agent engines. As we search both from the start state onwards and from the goal state backwards we perform bidirectional breadth–first search.

Solution Extraction

If exploration terminates with success a plan exists. The last task is to extract a solution sequence. This is done by extracting one state u of the intersection `meet`. As in bidirectional search, the inverse of the transition relation can be used to build the solution in the opposite direction of the exploration process.

For both search frontiers we determine all states which may have led to u. Since the nodes on the searched solution path have been reached one iteration before termination we build a disjunction of this set and the set of states reached in this iteration. Now, we extract one state of the intersection and iterate.

When combining the path of the forward and backward solution path, we are left with a sequence of states $(v_0, v_1, \ldots, v_{n-1}, v_n)$, such that for all $i \in \{0, \ldots, n-1\}$ there exists an operator op_i that performs the transition from v_i to v_{i+1}. Finally we determine the corresponding operator sequence by simply trying all possibilities on the given pairs. Similar to the search process the implementation consists of two parts (depicted in Figure 3 and Figure 4): the outer loop and the individual solution step to apply the relational product and to successively extract one state for the next iteration.

```
void extractSolution(vector<bdd> &forward,
    vector<bdd> &backward,bdd meet) {
  bdd meetForward = meet;
  bdd meetBackward = bdd_replace(meet,effPair);
  list<bdd> states;
  states.push_front(meetForward);
  for(int i = forward.size() - 2;i >= 0;i--)
    states.push_front(solutionStep
      (meetBackward,forward[i],false));
  for(int i = backward.size() - 2;i >= 0;i--)
    states.push_back(solutionStep
      (meetForward,backward[i],true));
}
```

Figure 3: Solution extraction.

Complexity

Since STRIPS-planning is PSPACE complete we cannot expect an polynomial time planner. Therefore we exhibit, which parts of the algorithm are difficult and which parts are not.

The parsing process can be executed in linear time with respect to the length of the file. The data structures for referencing actions, predicates and objects by unique identifiers can be efficiently built by using balanced trees. For detecting constant predicates each effect list is considered at most once, resulting in a time bound linear in the total number of effects.

```
bdd solutionStep(bdd &current,bdd next,
    bool forward) {
  current = next &
    bdd_relprod(current,trans,
                forward ? preVar : effVar);
  current = bdd_fsatone(current);
  if(forward) {
    current = bdd_replace(current,prePair);
    return current;
  } else {
    bdd state = current;
    current = bdd_replace(current,effPair);
    return state;
  }
}
```

Figure 4: Calculating one solution step.

The balancing technique of predicates leads to an algorithm exponential in the number of (non-constant) predicates, since each combination of predicates in the effect lists can be considered in the recursive approach. However, in the benchmark problems we never achieved a recursion depth of more than two, which corresponds to the exponent. Moreover, the total number of (non-constant) predicates itself was small.

When exploring of the fact space (as proposed by utilizing a queue) the number of facts to be enqueued is bounded by the number total number of facts. Each instantiation step can take time linear in the number of operators. In theory both number can be large (exponential in the maximal parameter length of predicates and maximal number of operator schemes, respectively), but in well-modeled domains the exponents should be small constants leading to a polynomial time algorithm. To generate the encoding we systematically compare the different possibilities generated in the preceding steps. We prune the generation if the length of the next encoding exceeds the length of the currently minimal one. Nevertheless, the time required is exponential in the number of different possibilities to merge (non-constant) predicates. As mentioned above this number will be small in practice.

Building the BDD for the transition relation is critical in both time and space. In general the BDD size can be exponential in the number of bits in the encoding. However, the hope is that this will not be the case. The partial relation of each action is found by combining the representations of all possible operators.

The variable ordering of the encoded predicates is very important and can be improved by a precompiling step systematically trying some orders and measuring the growth of the transition relation. The optimal ordering is an NP hard problem so that we cannot expect an efficient algorithm. As said the number of non-constant predicates is small so that at least a greedy hill climbing strategy will do. The size of the transition relation is a predictor for the efforts in the exploration.

Tough we have faced two hard problems in building the transition function, the exploration includes an NP hard problem (the determination of the relational product) in each iteration step. Fortunately, there is a lot of practical evidence that this work is small with respect to the set of represented states.

Solution extraction takes time proportional to the product of the solution length, the number of operators plus the BDD size of the transition function, and the binary encoding lengths, since relational product for one state and the transition function is linear in the product of both BDD sizes.

In summary, pre- and postprocessing contribute substantially to the overall running time but the BDD operations in building the transition function and exploration are dominating.

Architecture and Use (Version 1.0)

Figure 1 depicts the coarse structure of the planner (Version 1.0). The parts of the planning system are as follows: `main`: Reads the command line, starts the planning system. `data.domain`: Maintains most data structures of the planner. `data.object`, `data.pred`, `data.fact`, `data.action`: Responsible for handling objects, action, predicates, and facts of the planning domain. `data.symbFact`: Maintains *symbolic* facts as found in precondition and effect lists of the operators. `step.parse`: Reads input files and builds the corresponding data structures. For this purpose a simple `parser` for processing *LISP* files is utilized. `step.constant`: Detects constant predicates. `step.merge`: Determines and unifies balanced predicates. `data.mergedPredicate` and `data.partPredicate`: Contain all methods for merging different predicates. `step.explore` Implements fact space exploration. `step.coding`: Realizes the generation of the state encoding. `bddEngine`: Employs BDDs, especially for the exploration phase. The transition relation is built in `transitionBuilder` and the extraction of the plan is delegated to `bdd.map`. `tools`: Some small routines that are used in different parts of the system: failure handling, time measurement, tuple, and some mathematical routines. `bitarray`: Maintains bitvectors. `option`: Parses the command line.

The command to invoke the planner is

```
mips [<options>] [<problem>] [<domain>].
```

Each parameter can be omitted. If only one file is given it is considered as the problem file and the domain description is read from `domain.pddl` in the current directory. If no file is specified, then the problem description is included from `problem.pddl` and the domain description `domain.pddl` is read. The following options are available: `-?`, `-h` (*help*): Outputs a small user manual. The planner itself is not invoked. `-p` (*preprocess*): Only the steps for generating the state encoding are processed. `-t` (*transition*): Only the steps for preprocessing and building the transition function are

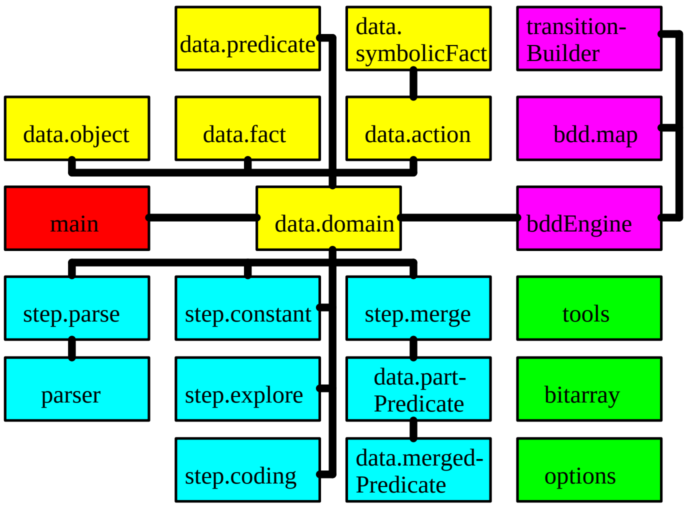

Figure 5: The architecture of MIPS 1.0.

executed, but no exploration takes place. `-u` (*unidirectional*): BDD exploration is invoked uni-directionally. The default is a bidirectional exploration. `-s` (*silent mode*): Less program information output is printed onto the screen. `-n` (*normal mode*): Normal program information is given (only useful in combination with `-s`, `-v` or `-d`) `-v` (*verbose output*): More program information is printed, e.g. the time and represented planning states for each exploration step is given. `-d` (*debug output*): Maximal information is given, e.g. the contents the LIFO-queue while exploring fact space.

The last four option can be appended by arbitrary letters from `pcmeobts` to impose restrictions on the phases in which the information should be printed (default is all phases): `p` abbreviates *parsing*, `c` *constant predicates*, `m` *merging*, `e` *exploring (fact space)*, `o` *coding*, `b` *BDD package handling*, `t` *transition function creation*, and `s` *BDD search.* For example:

```
mips -u -s -vo gripper.pddl gripper.10.pddl
```

invokes the planner with domain `gripper.pddl` and problem `gripper.10.pddl`. The search process is unidirectional (`-u`) with minimal output except for the coding phase in which additional information is given.

New Inventions (Version 1.1-1.5)

The extend of MIPS in its version 1.0 incorporates parsing, precomplilation, (bidirectional) breadth–first search, and solution extraction. In versions 1.1 we added BDD refinement techniques (*forward set simplification*, *constrain and restrict operators* and (simple) *transposition relation splitting.* In version 1.2 and 1.3 the symbolic heuristic search engines BDDA* and *Pure BDDA** were implemented. Subsequently, in version 1.4 we improved the heuristic estimate. In version 1.5

we extended symbolic search with a performant heuristic single-state search engine.

BDD Refinements (Version 1.1)

Several approaches have been proposed to improve the efficiency in calculating the image of a set of states according to a transition function. The daunting problem is that even it the input and the output representations are small the representation during the computation may be larger by magnitudes.

Forward Set Simplification The introduction of a list *Closed* containing all states ever expanded is very common in single state exploration to avoid duplicates in the search. Usually, the memory structure is realized as a hash table which in this context is referred to by the term *transposition table.* For symbolic search this technique is called *forward set simplification.* Let `reached` be the BDD representation of the set of reached nodes and `to` be the image of the current set. Then forward set simplification is implemented by `current = to & !reach`, i.e. the set of already reached nodes is subtracted from the set of nodes in the next iteration.

Constrain and Restrict Operators Note that any set R in between the successor set and the simplified successor set will be a valid choice for the horizon in the next iteration. Therefore, one may choose a set R that minimizes the BDD representation instead of minimizing the set of represented states. This is the idea of the *restrict operator* $\Downarrow$, which itself is a refinement to the *constrain operator* $\downarrow$. Without going into involved details we denote that both operators can be implemented efficiently and are available in several packages (Coudert, Berthet, and Madre 1990).

With the flags `optimize` for the restrict operator and `simplify` for forward set simplification and the BDD `reach` we get the code fragment of Figure 6 to determine the next preimage given the current one.

```
Image(bdd& current, bdd trans) {
  bdd to = bdd_relprod(trans,current,preVar);
  meet = back & to;
  to = bdd_replace(to,renamePair);
  if (!optimize & !simplify)
    current = to;
  if (!optimize & simplify)
    current = to & !reach;
  if (optimize & simplify)
    current = bdd_simplify(to,!reach);
}
```

Figure 6: Calculating th image of a set of states according to different simplification schemes.

Transition Function Splitting Fortunately, in the considered domains of search and planning problems the transition function is composed of smaller parts, called the operator functions: $T(x',x) = \bigvee_{o\in O} o(x',x)$. Therefore, we can simplify the calculation of the successor set: $\exists x'\ (\mathit{From}(x')\ \wedge\ T(x',x)) = \bigvee_{o\in O}\ \exists x'\ (\mathit{From}(x') \wedge\ o(x',x))$

A more efficient computation is obtained by partitioning the transition relation and performing the existential quantification of next state variables early in the calculation (Burch, Clarke, and Long 1991; Ranjan et al. 1995). To do this the transition relation T is split into a conjunction of partitions $T_1,\ldots,T_n$ allowing the modified calculation: $\exists x'\ \mathit{From}(x') \wedge T(x',x) = \exists x'_n T_n \wedge\ \ldots \wedge\ \exists x'_1 T_1 \wedge \mathit{From}(x')$ This approach has effectively been applied to the UMOP planning system (Jensen and Veloso 1999) and is planned to enrich MIPS in the near future.

Symbolic Heuristic Search (Version 1.2-1.4)

In the symbolic version of A* (Hart, Nilsson, and Raphael 1968), called BDDA* (Edelkamp and Reffel 1998), the relational product algorithm determines all successor states according to the set of states with minimal merit and the given heuristic estimate in one evaluation step. *Heuristic pattern databases* (Culberson and Schaeffer 1996) serve as heuristics and are combined to an overall heuristic by taking the maximum or the sum of subposition solutions. *Subpositions* are facts and the estimated distance of each single fact p to the goal is a heuristic value associated with p. In the *add*-version of a heuristic these values are added and in the *max*-version these values maximized. Especially in reversible problem spaces like *Logistics* this approach pays off. Searching with the *max*-Heuristic achieves better solutions compared to the *add*-Heuristic, but takes more time. A variant of BDDA*, called *Pure* BDDA* is obtained by ordering the priority queue only according to the h values. In this case the calculation of the successor relation simplifies to $\exists x'\ \mathit{Min}(x')\ \wedge\ T(x',x)\ \wedge\ \exists h\ H(h,x)$. As a code fragment this leads to the implementation given in Figure 7.

```
void pureHeuristicStep() {
  current = find_min(open);
  open = open & ! min;
  Image(current,trans);
  current = current & heuristic;
  current = bdd_replace(current,exchange);
  open = open | current;
  reach = reach | bdd_exist(from,preMeritVar);
  bddVec.push_back(open);
}
```

Figure 7: Source code for one exploration step with *Pure BDDA**.

We experimented with two symbolic estimates. The **HSP-Heuristic** of a fluent p is found through the fixed-point calculation of the relaxed planning problem according to the set of reachable facts. The fact exploration phase has been extended to output an HSP-like heuristic estimate $h(p)$ for each fact p. Since we omit facts to be generated twice with each explored fact we can associate a depth by adding the value 1 to its predecessor. The incorporated search knowledge is not as good as in HSPr since we do not consider mutual exclusions. Nevertheless, given the list of value/fact pairs and the heuristic in a symbolic representation we achieve good results. That is the symbolic representation compensates for a weaker knowledge.

For the **FF-Heuristic** we have implemented the FF planning approach (see below). The estimate for each fluent is calculated in a precompliation phase with respect to the fluent representing the goal. In the experiments the average heuristic value per fluent according to the *FF-Heuristic* is about 50 percent larger than in the *HSP-Heuristic*. However, by simplifying states to fluents a lot of the refined heuristic information in FF is lost. Therefore, in search spaces where the heuristic guidiance leads to a few thousand states to be evaluated, we stick to single-agent search.

Single-Agent Heuristic Search (Version 1.5)

To infer the heuristic estimate for each state FF (for fast-forward planning) approximates a relaxed planning problem (del-effect list omitted) in a combined forward and backward traversal (Hoffmann 2000). In terms of *Graphplan*, FF builds the plan graph *and* extracts a simplified solution by returning the number of instantiated operators that at least have to fire. Therefore, the heuristic estimate in FF is an elaboration to the heuristic in HSP (Bonet and Geffner 1999), since the latter one considers the first phase only. With *enforced hill climbing* it further employs another search strategy and drastically reduces the explored portion of search space at least for some important planning domains.

We have newly implemented the FF planning approach. The precompilation phase gives a list of facts, such that each state can be interpreted as a bitvector of facts. **Forward** traversal determines the set of operators and facts in the layered graph structure and is depicted in Figure 8.

To improve the performance in `preActions` the operators are kept in lists according to their preconditions. In the algorithm the number of not achieved preconditions is decreased until it equals zero. All operators that lead to new fluents are kept in a queue, on which **Backward** traversal extracts a relaxed solution (Figure 9). The relevant operator queue and the array `vis` (initialized with zeros) are used to reduce the branching factor and provide an ordering on the set of successors for expansion.

```
void Forward(State& from,State& to){
  enqueuedFacts = from.getVector();
  processedFacts = zeroFacts;
  markedFacts = to.getVector();
  while(!(markedFacts == zeroFacts)) {
    goals[depth] = enqueuedFacts;
    goals[depth++] -= processedFacts;
    processedFacts |= enqueuedFacts;
    goals[depth-1]->toIntArray(FactArray,size);
    for (int i=0; i<size;i++) {
      fact = FactArray[i];
      for(int o=0;o<preActions[fact.size()];o++){
        opererator = preActions[fact][o];
        (opererator->presize)--;
        if (operator->presize == 0) {
          enqueued = false;
          for (int k=0;k<operator->addsize;k++){
            succId = operator->add[k];
            if(!enqueuedFacts.get(succId)) {
              enqueued = true;
              enqueuedFacts.set(succId);
            }
          }
          if (enqueued)
            operQueue[operSentinal++] = operator;
        }
      }
    }
    markedFacts -= enqueuedFacts;
  }
}
```

Figure 8: Determining the fix-point on the set of facts.

Experiments

MIPS will compete on AIPS-00 as a fully automated planner in the combined Strips and ADL track. Since the problems have not been released yet we concentrate on results to the AIPS-98 competition.

The experiments were run on a Linux PC/450 MHz with 128 MByte. The precompiling and solution extraction times are included.

The observed time offset of MIPS 1.4 to MIPS 1.0 for small domains is due to different extractions algorithms of the solution paths. In MIPS 1.0 we traverse the path strictly backwards, whereas in MIPS 1.4 we start at the initial state and search for the first predecessor state in the list of BDDs.

In the *Movie* domain no planner has any difficulties in solving the problems. Moreover, precompiling exhibits that all problems in the domain have the same encoding so that the running times and solution lengths match (0.03s, length 7).

In *Gripper* we choose unidirectional search so that forward set simplification in *MIPS 1.4* scales better. The single-state heuristic search engine of MIPS 1.5 solves all problems to gripper in the optimal number of expansions in less than a tenth of a second.

In *Logistics Pure BDDA** with the *HSP-Heuristic* leads to a noticeable improvement in solvability by the cost of longer solutions. Further experiments indicate that on average the *FF-Heuristic* leads to shorter solutions and to smaller execution times (Edelkamp 2000).

```
int Backward(State& from) {
  matches = 0;
  markedFacts = zeroFacts;
  processedFacts = from.getVector();
  while (operSentinal > 0) {
    Operator* oper = operQueue[--operSentinal];
    for (int j=0; j< oper->addsize;j++) {
      if (processedFacts[oper->add[j]] &&
          ! markedFacts[oper->add[j]]) {
        matches++;
        for (int k=0; k< oper->addsize;k++)
          markedFacts.set(oper->add[k]);
        for (int k=0; k< oper->presize;k++) {
          vis[oper->pre[k]] = vis[oper->add[j]]+1;
          processedFacts.set(oper->pre[k]);
        }
        relevantQueue[operRelevant++] = oper;
        oper->pres = vis[oper->add[j]];
      }
    }
  }
  return matches;
}
```

Figure 9: Backward traversal.

Domain	Problem	MIPS 1.0		MIPS 1.4	
Gripper	1-1	11	0.24s	11	0.45s
	1-2	17	0.25s	17	0.46s
	1-3	23	0.29s	23	0.49s
	1-4	29	0.30s	29	0.53s
	1-5	35	0.34s	35	0.58s
	1-6	41	0.39s	41	0.65s
	1-7	47	0.45s	47	0.75s
	1-8	53	0.33s	53	0.86s
	1-9	59	1.45s	59	1.25s
	1-10	65	2.11s	65	1.70s
	1-11	71	2.48s	71	2.24s
	1-12	77	3.60s	77	2.92s
	1-13	83	4.71s	83	3.67s
	1-14	89	6.62s	89	4.56s
	1-15	95	7.86s	95	5.59s
	1-16	101	9.12s	101	6.78s
	1-17	107	11.71s	107	8.18s
	1-18	113	13.34s	113	9.73s
	1-19	119	34.03s	119	15.95s
	1-20	125	96.70s	125	20.81s

The results compete well with other symbolic approaches but are too weak to beat heuristic single-state planners. Our single-state engine, however, can solve the suite of 30 problem instances of round 1. The solution quality and the performance are almost as good as in original FF; in time we are off by a factor of about two and the achieved solution lengths are: 26 (-1), 32 (+0), 57 (+3), 62 (+4), 23 (+1), 71 (-2), 34 (-2), 45 (+4), 85 (-6), 105 (+2), 30 (+0), 42 (+1), 68 (+1), 95 (-3), 91 (-2), 56 (+1), 45 (+1), 174 (+7), 147 (-4), 138 (-1), 104 (+2), 295 (+13), 115 (-11), 40 (+0), 181 (+0), 208 (+25), 145 (+4), 266 (+1), 318 (-5), 131 (+0).

Domain	Problem	MIPS 1.0		MIPS 1.4	
Logistics	1-1	26	165.76s	38	7.03s
	1-2			37	15.23s
	1-3			70	70.18s
	1-4			72	87.67s
	1-5	22	37.11s	32	6.97s
	1-6			92	717.23s
	1-7			36	52.84s
	1-9			107	807.45s
	1-10			132	541.98s
	1-11			34	233.05s
	1-23			139	483.63s
	2-1	13	0.39s	16	3.29s
	2-2	20	21.29s	24	3.75s
	2-3			38	6.55
	2-4			59	17.99s
	2-5			42	13.78s

Mystery has some unsolvable problems (4,7,12) which are easily found by *MIPS 1.4* trough forward set simplification. These problems become tractable with the extra `drink` operator provided in *Mprime*. Single state search solves 16 problem of *Mprime* (first round), including the new ones 1-2 and 1-17.

Domain	Problem	MIPS 1.0		MIPS 1.4	
Mystery	1-1	5	0.19s	5	0.50s
	1-3			4	4.75s
	1-4			-1	179.89s
	1-7			-1	1.56s
	1-9	8	165.32s	8	27.08s
	1-11	7	0.69s	7	0.82s
	1-12			-1	65.91s
	1-17			4	242.79s
	1-19			6	205.06s
	1-25	4	0.17s	4	0.53s
	1-26			6	5.81s
	1-27	5	4.11s	5	2.19s
	1-28	7	0.28s	7	0.62s
	1-29	4	6.81s	4	2.10s

Domain	Problem	MIPS 1.0		MIPS 1.4	
Mprime	1-1	5	1.21s	5	1.32s
	1-3			4	16.23s
	1-4	8	12.31s	8	9.27s
	1-7	5	31.90s	5	20.70s
	1-8			6	215.27s
	1-9			8	91.17s
	1-11	7	7.71s	7	5.30s
	1-12	6	29.52s	6	12.08s
	1-16			6	98.62s
	1-21			6	408.87s
	1-25	4	0.36s	4	0.69s
	1-26			6	49.42s
	1-27	5	42.91s	5	24.06s
	1-28	7	2.51s	7	1.94s
	1-29	4	41.66s	4	21.79s
	2-1			4	36.93s
	2-2	7	17.04s	7	8.28s
	2-4			4	32.57s
	2-5			5	2.12s

Grid is still a hard problem for our approach. Although we can solve four problems with the μcke model checker applying BDDA* in the all-contained planners MIPS 1.0-1.4 up to now we are still restricted to two problems. However, with relaxations to the transition relation we can solve two further problems. Due to some yet unresolved problems single-state search fails to give fruitful results.

Domain	Problem	MIPS 1.0		MIPS 1.4	
Grid	2-1	14	23.01s	14	8.55s
	2-2			26	355.60s

Below we depict the encoding lengths, total number of facts (number of facts to be encoded is given in parenthesis) and the number reachable operators for some selected problems solved with MIPS 1.0.

Problem	Bits	Facts	Operators
Movie 1-28	7	160 (7)	809 (162)
Movie 1-29	7	165 (7)	834 (167)
Movie 1-30	7	170 (7)	859 (172)
Gripper 1-18	79	3.738 (156)	149.940 (460)
Gripper 1-19	83	4.092 (164)	172.304 (484)
Gripper 1-20	87	4.462 (172)	196.788 (508)
Logistics 1-01	42	3.264 (144)	1.212.416 (727)
Logistics 1-05	35	5.805 (151)	3.816.336 (699)
Logistics 2-02	28	1.449 (80)	240.786 (341)
Mystery 1-01	28	3.192 (58)	12.252.303 (269)
Mystery 1-07	82	12.558 (181)	392.073.696 (521)
Mystery 1-27	63	7.788 (152)	117.406.179 (2.280)
MPrime 1-07	126	12.558 (352)	$\approx 231 \cdot 10^9$ (3.291)
MPrime 1-11	61	4.862 (131)	$\approx 8 \cdot 10^9$ (3.189)
MPrime 1-28	41	4.152 (90)	$\approx 5 \cdot 10^9$ (2.544)
Grid 2-01	67	6.043 (276)	2.144.340 (4.295)

Conclusion

In this paper we presented the current status of the *intelligent model checking and planning system* (MIPS) project (cf. `http://www.informatik.uni-freiburg.de/~edelkamp/Mips` for more information.)

Further on, we investigated currently finished and unfinished work and present various experimental data on the AIPS-98 problem suite.

We intend to build a hybrid planner on heuristic search and BDD-exploration. The challenging question is how to combine the approach with other successful planning techniques *Graphplan* (Blum and Furst 1995), *Tim* (Fox and Long 1998), *Integer Programming* (Kautz and Walser 1999), and *Satplan* (Kautz and Selman 1996) to build a system that can solve really hard problems.

Acknowledgments We thank J. Hoffmann and B. Nebel for introducing us to the fast-forward planning approach.

References

Blum, A., and Furst, M. L. 1995. Fast planning through planning graph analysis. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI'95)*, 1636–1642. Morgan Kaufmann.

Bonet, B., and Geffner, H. 1999. Planning as heuristic search: New results. In Biundo, S., and Fox, M., eds., *Recent Advances in AI Planning. 5th European Conference on Planning (ECP'99)*, volume 1809 of *Lecture Notes in Artificial Intelligence*, 360–372. Heidelberg: Springer-Verlag.

Burch, J. R.; Clarke, E. M.; and Long, D. E. 1991. Symbolic model checking with partitioned transistion relations. In Halaas, A., and Denyer, P. B., eds., *Proceedings of the International Conference on Very Large Scale Integration (VLSI 1991)*, 49–58.

Coudert, O.; Berthet, C.; and Madre, J. C. 1990. Verification of synchronous sequential machines based on symbolic execution. In *Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, 365–373. Springer-Verlag.

Culberson, J. C., and Schaeffer, J. 1996. Searching with pattern databases. In *Proceedings of the Eleventh Biennial Conference of the Canadian Society for Computational Studies of Intelligence (CSCSI-96)*, volume 1081 of *Lecture Notes in Artificial Intelligence*, 402–416. Springer-Verlag.

Edelkamp, S., and Helmert, M. 1999. Exhibiting knowledge in planning problems to minimize state encoding length. In Biundo, S., and Fox, M., eds., *Recent Advances in AI Planning. 5th European Conference on Planning (ECP'99)*, volume 1809 of *Lecture Notes in Artificial Intelligence*, 135–147. Heidelberg: Springer-Verlag.

Edelkamp, S., and Reffel, F. 1998. OBDDs in heuristic search. In Herzog, O., and Günter, A., eds., *Proceed-*

ings of the 22nd Annual German Conference on Artificial Intelligence (KI 1998), volume 1504 of *Lecture Notes in Computer Science*, 81–92. Springer-Verlag.

Edelkamp, S. 2000. Heuristic search planning with BDDs. In *14th Workshop on New Results in Planning, Scheduling and Design (PuK 2000)*.

Fox, M., and Long, D. 1998. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research* 9:367–421.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2):100–107.

Hoffmann, J. 2000. A heuristic for domain independent planning and its use in an enforced hill-climbing algorithm. Technical Report 133, Albert-Ludwigs-Universität Freiburg, Institut für Informatik.

Jensen, R. M., and Veloso, M. M. 1999. OBDD-based universal planning: Specifying and solving planning problems for synchronized agents in non-deterministic domains. In Wooldridge, M., and Veloso, M. M., eds., *Artificial Intelligence Today*, volume 1600 of *Lecture Notes in Computer Science*. Springer-Verlag. 213–248.

Kautz, H., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, 1194–1201. AAAI Press.

Kautz, H. A., and Walser, J. P. 1999. State-space planning by integer optimization. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, 526–533. AAAI Press.

Ranjan, R. K.; Aziz, A.; Brayton, R. K.; Plessier, B.; and Pixley, C. 1995. Efficient BDD algorithms for FSM synthesis and verification. In *Workshop Notes of the International Workshop on Logic Synthesis (IWLS 1995)*.

Reffel, F., and Edelkamp, S. 1999. Error detection with directed symbolic model checking. In Wing, J. M.; Woodcock, J.; and Davies, J., eds., *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM 1999)*, volume 1708 of *Lecture Notes in Computer Science*, 195–211. Springer-Verlag.