BDD–based Reasoning in the Fluent Calculus — First Results (Extended Abstract)

Steffen Hölldobler and Hans–Peter Störr
Artificial Intelligence Institute
Department of Computer Science
Dresden University of Technology
{sh,hans-peter}@inf.tu-dresden.de

Abstract

The paper reports on first preliminary results and insights gained in a project aiming at implementing the fluent calculus using methods and techniques based on binary decision diagrams. After reporting on an initial experiment showing promising results we discuss our findings concerning various techniques and heuristics used to reduce the search space.

1 Introduction

In recent years we have seen highly advanced and novel implementations of propositional calculi and systems like, for example, GSat and its variants [13], Smodels [12] or Dlv [8], to mention just a few. The implementations were applied to many interesting fields in Intellectics like, for example, planning or non–monotonic reasoning. On the other hand, very few results are reported so far on applying another propositional method in these fields, viz., model checking using binary decision diagrams (BDDs), with [5, 6] being an exception. This comes to a surprise because model checking using binary decision diagrams has significantly improved the performance of algorithms and enabled the solution of new classes of problems in related areas like formal

verification and logic synthesis (see e.g. [3, 4]). Can we adopt the technology developed for model checking of finite state machines using binary decision diagrams for the solution of planning problems and, more generally, problems occurring in reasoning about situations, actions and causality? Can we enrich these techniques by exploiting the experiences made in the state of the art implementations of propositional logic calculi and systems mentioned at the beginning of this paragraph?

In order to answer these and related questions a sound and complete mapping from (a fragment of) the fluent calculus [9, 14] to propositional logic is specified in [10] such that the entailment problem in the fluent calculus can be solved by finding models for the corresponding propositional logic formula. The propositional logic formulas are represented by reduced and ordered binary decision diagrams and techniques from model checking are applied to search for models.

Here we report on initial results, findings and insights gained with the BDD–based implementation of the fluent calculus. After briefly discussing the fluent calculus and the implementation using an example from the so–called GRIPPER–class, we concentrate on two heuristics and techniques which can be applied to speed up the solution of the planning problem. In particular, we discuss some results on variable ordering and partitioning of the transition relation.

In this extended abstract we assume the reader to be familiar with the fluent calculus and binary decision diagrams, and refer to [14, 10] and [2] as references respectively.

2 Gripper Planning Problems

In a contest held at AIPS98, planners had to solve various problems, among which were the problems of the so–called GRIPPER class:

> *A robot equipped with two grippers G_1 and G_2 can move between two rooms A and B. Initially the robot is in room A together with a number of balls $B_1, \ldots, B_n$. The task is to transport these balls into room B.*

The problems differ wrt the number of balls and are then called GRIPPER–1, GRIPPER–2 etc.

We will specify GRIPPER class problems in the fluent calculus in a moment. Before doing so, however, some notational conventions are helpful. Words starting with an upper letter denote constants, whereas words starting with a lower letter denote predicate symbols, non–nullary function symbols and variables. Additionally we assume that each variable a denotes an action, s a situation, f a fluent and z a state. The latter conventions are encoded as sorts in the fluent calculus (see [14, 10]). All symbols may be indexed. We also make frequently use of the abbreviation

$$holds(f, s) \equiv (\exists z)\ state(s) = f \circ z \ . \tag{1}$$

The initial state of a reasoning problem in the fluent calculus is specified by an axiom of the form

$$\mathcal{F}_{S_0} = \{state(S_0) = t\}, \tag{2}$$

relating the initial situation S_0 to a state t. t itself is a so–called *constructor state term* of the form $\emptyset \circ f_1 \circ \ldots \circ f_n$, $n \geq 0$ where the f_i's are pairwise distinct. If an equation like (2) is given, then $\Phi_I(z)$ denotes the equation $z = t$. Turning to the example, the initial state of a GRIPPER class problem is specified by

$$\begin{aligned} \mathcal{F}_{S_0} = \{state(S_0) \quad = \quad & at(B_1, A) \circ \ldots \circ at(B_n, A) \\ & \circ free(G_1) \circ free(G_2) \circ at\text{-}robby(A) \quad \}, \end{aligned}$$

where n is instantiated to some number. Informally, *state* is a function mapping situations to states. The fluent $at(b, r)$ states that ball b is at room r, $free(g)$ states that gripper g is free and $at\text{-}robby(r)$ states that the robot is at room r.

There are three actions in the GRIPPER class:

- the robot may *move* from one room to the other.
- the robot may *pick* up a ball if it is in the same room as the ball and one of its grippers is empty.
- the robot may *drop* a ball if it is carrying one.

These actions are specified by means of state update axioms:

$$\begin{aligned}
\mathcal{F}_{su} \;=\; \{\;& holds(at\text{-}robby(r_1), s) \wedge \neg holds(at\text{-}robby(r_2), s) \\
& \rightarrow \; state(do(move(r_1, r_2)), s) \circ at\text{-}robby(r) \\
& \quad = \; state(s) \circ at\text{-}robby(r_2) \qquad ,\\
& holds(at(b, r), s) \wedge holds(at\text{-}robby(r), s) \\
& \wedge \; holds(free(g), s) \wedge \neg holds(carry(b, g), s) \\
& \rightarrow \; state(do(pick(b, r, g)), s) \circ at(b, r) \circ free(g) \\
& \quad = \; state(s) \circ carry(b, g) \qquad ,\\
& holds(carry(b, g), s) \wedge holds(at\text{-}robby(r), s) \\
& \wedge \; \neg holds(at(b, r), s) \wedge \neg holds(free(g), s) \\
& \rightarrow \; state(do(drop(b, r, g)), s) \circ carry(b, g) \\
& \quad = \; state(s) \circ at(b, r) \circ free(g) \qquad \}
\end{aligned}$$

In addition, we need the axioms $\mathcal{F}_{un}$ specifying the unique name assumption for fluents, $\mathcal{F}_{mset}$ specifying that all state–terms denote multisets of fluents and $\mathcal{F}_{ms}$ specifying that in each state each fluent may occur at most once. Altogether, the axioms of the fluent calculus are

$$\mathcal{F} = \mathcal{F}_{un} \cup \mathcal{F}_{mset} \cup \mathcal{F}_{S_0} \cup \mathcal{F}_{ms} \cup \mathcal{F}_{su}.$$

Reasoning problems themselves are specified as entailment problems in the fluent calculus. For the GRIPPER class we obtain the entailment problem

$$\mathcal{F} \models (\exists s)\; holds(at(B_1, B), s) \wedge \ldots \wedge holds(at(B_n, B), s).$$

Using abbreviation (1) this can be reformulated as

$$\mathcal{F} \models (\exists s)\; [(\exists z)\; state(s) = at(B_1, B) \circ z] \wedge \ldots \wedge [(\exists z)\; state(s) = at(B_n, B) \circ z],$$

which itself is equivalent to

$$\mathcal{F} \models (\exists z)\; [(\exists s)\; state(s) = z] \wedge [(\exists z')\; z = at(B_1, B) \circ \ldots \circ at(B_n, B) \circ z'].$$

In general, reasoning in the fluent calculus amounts to solving an entailment problem of the form

$$\mathcal{F} \models (\exists z)\; [(\exists s)\; state(s) = z] \wedge \Phi_G(z),$$

where $\Phi_G(z)$ is a goal formula with z as the only free variable. Such problems have a solution if we find a substitution σ for z such that

$$\mathcal{F} \models [(\exists s)\ state(s) = z\sigma] \tag{3}$$

and

$$\mathcal{F} \models \Phi_G(z\sigma)\ . \tag{4}$$

It is sufficient to restrict us to substitutions σ which actually denote states of our reasoning problem, i.e., substitutions which contain solely bindings of variables of type STATE to constructor state terms. Such substitutions are called *constructor state substitutions*. In the sequel, σ will always denote a constructor state substitution.

The main idea of the algorithm presented in [10] is to calculate a sequence $(S_i \mid i \geq 0)$ of solutions to (3) which correspond to the sets of states reached after executing $0,\ 1,\ 2,\ \ldots$ actions starting in the initial state, until a state is found which is a goal state, i.e. fulfills (4), or, if no new states are reached, in which case there is no plan. The implementation of this algorithm is done by representing these sets by binary decision diagrams (BDDs).

3 The Algorithm

The algorithm for solving entailment problems in the fluent calculus follows in spirit the algorithm to find reachable states presented in [4]. As mentioned, the aim is to find the sets S_i of solutions for (3) representing states which can be reached from the initial state after the execution of i actions. The first crucial question to tackle is how to represent these sets using BDDs.

Each solution to (3) is a constructor state substitution $\{z/t\}$ with a term t of the form $\emptyset \circ f_1 \circ \ldots \circ f_n$, where the f_i's are pairwise distinct. On first glance it seems impossible to represent substitutions by finite BDDs because there are infinitely many terms. Fortunately, however, if there are only finitely many fluents then there are also only finitely many terms t such that $\{z/t\}$ satisfies (3) due to $\mathcal{F}_{ms}$. Furthermore, because $\circ$ is an AC1–symbol in the fluent calculus we do not have to distinguish between terms which are equivalent under the AC1 equational theory. In other words, a term t occurring in the codomain of a constructor state substitution is uniquely characterized by the set of fluents occurring in t.

This observation opens a possibility for encoding sets of solutions for the entailment problem in the fluent calculus into a BDD: for each of the finitely many fluents f which may occur in the binding for a variable z in a constructor state substitution we introduce a propositional variable z_f. A constructor state substitution $\sigma = \{z/t\}$ is represented by a valuation $\mathcal{B}(\sigma)$ for these variables such that z_f is mapped to $\top$ by $\mathcal{B}(\sigma)$ iff f occurs in t. [1] Hence, a set S of constructor state substitutions is represented by a set of valuations. The set of valuations itself is represented by a propositional formula Z such that the set of models for Z is the set of valuations. Finally, Z is represented by a BDD. For example, if the alphabet underlying the fluent calculus contains precisely the fluent symbols a, b and c, then the set $\{\{z/a \circ b\}, \{z/c \circ b\}\}$ is represented by the formula $(z_a \wedge z_b \wedge \neg z_c) \vee (\neg z_a \wedge z_b \wedge z_c)$.

The crucial point of our application of methods and techniques based on BDDs to reasoning in the fluent calculus is the following: We could identify a class $\mathcal{C}$ of formulas over the alphabet underlying the fluent calculus and a transformation $\mathcal{B}$ mapping each $F \in \mathcal{C}$ to a propositional logic formula $\mathcal{B}(F)$ such that (i) the class is expressive enough to represent interesting entailment problems wrt the fluent calculus and (ii) the following result holds.

Lemma 1. *Let $F \in \mathcal{C} \cup \{\Phi_I(z),\ \Phi_G(z)\}$ and σ a constructor state substitution such that $F\sigma$ does not contain any free variables. Then,*

$$\mathcal{F}_{un} \cup \mathcal{F}_{mset} \models F\sigma \quad \textit{iff} \quad \mathcal{B}(\sigma) \models \mathcal{B}(F).$$

The precise definition of $\mathcal{C}$ and $\mathcal{B}$ as well as the proof of this lemma is beyond the scope of this extended abstract and we refer the interested reader to [10] for all the details.

We turn now to the process of calculating the sequence $(S_i \mid i \geq 0)$. S_0 can be immediately derived from $\Phi_I(z)$. But how can we compute S_{i+1} given S_i and $\mathcal{F}_{su}$? In order to answer this question we define

$$\mathbf{T}_{\phi(a)}(z, z') = [\Delta(z) \wedge z' \circ \vartheta^- = z \circ \vartheta^+] \ . \tag{5}$$

for each state update axiom $\Phi(a) \in \mathcal{F}_{su}$ of the form

$$(\forall)[\Delta(state(s)) \rightarrow state(do(a, s) \circ \vartheta^- = state(s) \circ \vartheta^+]$$

[1] A substitution containing more than one binding is represented similarly: for each variable we introduce a separate set of propositional variables which encodes the binding of that variable.

Furthermore, for the set $\mathcal{F}_{su}$ we define

$$\mathbf{T}(z, z') = \bigvee_{\phi(a) \in \mathcal{F}_{su}} \mathbf{T}_{\phi(a)}(z, z') \ . \tag{6}$$

This definition is motivated by the following result, whose proof can again be found in [10]

Lemma 2. *Let t and t' be two constructor state terms and $\mathcal{F} \models \mathrm{state}(s) = t$. Then,*

$$\mathcal{F} \models \mathrm{state}(\mathrm{do}(a, s)) = t' \textit{ iff } \mathcal{F}_{un} \cup \mathcal{F}_{mset} \models \mathbf{T}_{\phi(a)}(t, t') \textit{ for some } \phi(a) \in \mathcal{F}_{su}.$$

Because $\mathbf{T}(z, z') \in \mathcal{C}$ we can compute $\mathcal{B}(\mathbf{T}(z, z'))$ and apply Lemma 1. Altogether we obtain a procedure for calculating the sequence $(S_i \mid i \geq 0)$ as follows. Let $\{f_1, \ldots, f_n\}$ be the finite set of fluents in the alphabet underlying the fluent calculus. Furthermore, let $F[z_1, \ldots, z_n]$ denote a propositional logic formula F built over the propositional variables $z_1, \ \ldots, \ z_n$. The sequence $(Z_i \mid i \geq 0)$ of propositional logic formulas corresponding to $(S_i \mid i \geq 0)$ is defined by

$$Z_0[\vec{z}] \ = \ \mathcal{B}(\Phi_I(z)) \tag{7}$$

$$Z_{i+1}[\vec{z}\,'] \ = \ (\exists \vec{z})\ Z_i[\vec{z}] \ \wedge \ \mathcal{B}(\mathbf{T}(z, z'))[\vec{z}, \vec{z}\,'], \tag{8}$$

where $\vec{z}$ is the vector $z_{f_1}, \ldots, z_{f_n}$ of propositional variables used to encode z and $(\exists \vec{z})\ F$ is an abbreviation for $(\exists z_1) \ \ldots (\exists z_n)\ F$ with

$$(\exists z_i)\ F = F\{z_i/\bot\} \ \vee \ F\{z_i/\top\} \ .$$

From (7) and (8) the so called *forward pass* of our planning algorithm for computing the sequence $(S_i \mid i \geq 0)$ can be derived:

1. Define S_0, i.e., the BDD–representation of Z_0, such that it contains only the initial state of the reasoning problem.

2. Recursively calculate S_{i+1}, i.e., the BDD-representation of Z_{i+1} based on Z_i and $\mathcal{B}(\mathbf{T}(z, z'))$, until either S_i overlaps with the set G of goal states, in which case the reasoning problem is successfully solved or until a fixpoint is reached, in which case the reasoning problem is unsolvable.

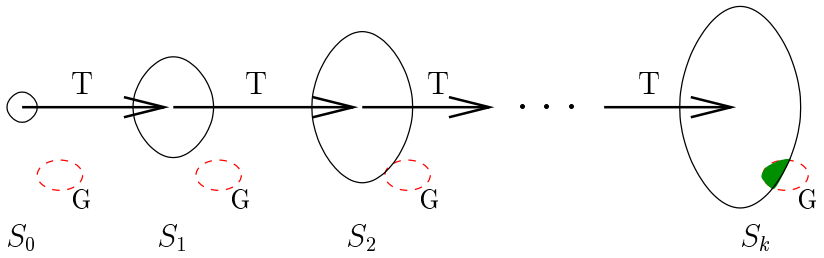

Figure 1: An illustration of the forward pass of our algorithm. After three steps the sets S_3 and G overlap.

The algorithm is illustrated in Fig. 1. Starting from the initial state all reachable states are generated. The algorithm terminates as soon as this set of states overlaps with the set of goal states or can no longer be expanded.

If the forward pass terminates successfully, then in a second step a shortest plan is constructed. This is done by choosing a state from $G \cap S_k$ and searching for a chain of states through which this state can be reached from the initial state. This is done by iterating backwards through the sets S_i generated by the forward pass algorithm. Because this second step is a computationally (relatively) inexpensive part, we refer the interested reader to [10], where also the soundness and completeness of the combined algorithm is established.

4 Optimizations

The planning approach described above is an implicit[2] breadth first search. In each single step we search the whole breadth of the search tree in depth i. The sets S_i can get quite complex and their BDDs quite large. Even more so, the size of the BDD for $\mathcal{B}(\mathbf{T}(z, z'))$, which describes the relation between the S_i, can quickly become too large to be handled in a graceful manner. Thus, a number of techniques were invented to limit a potential explosion in its size. In the sequel some of these techniques and their effects are discussed.

[2]It is called implicit because the calculated sets of states are never explicitly enumerated, but represented as a whole by a BDD, whose size depends more on the structure of the set, than on its actual size.

4.1 Variable Order

It is well known that the variable order used in a BDD has a large influence on the size of the BDD. Unfortunately it is still a difficult problem to find even an near optimal variable order.[3] Often, a good and acceptable variable order is found by empiric knowledge and experimentation. In our experiments it has turned out that fluents, which directly influence each other, should be grouped together. In particular, the variables z_f and z'_f occurring in $\mathcal{B}(\mathbf{T}(z, z'))$ should be ordered next to each other order. But how should these variable groups be arranged? An ordering we call *sort ordering* led to good results in several reasoning problems (see Tab. 1). The idea underlying the sort ordering is to group fluents by their arguments. For example, in the GRIPPER class the fluents $at(B_1, A), at(B_1, B), carry(B_1, G_1), carry(B_1, G_2)$ should be grouped together, because they share the argument B_1. Remember that the fluent calculus is sorted. The sort ordering works as follows. First one considers the argument of each fluent which belongs to the largest sort and sorts the fluents according to this argument. The remaining ambiguities are resolved by considering the argument of the second largest sort and so forth as well as the leading function symbol. For some domains Tab. 1 shows some almost dramatic improvements in the size of the BDDs for sort ordering if compared to a simple lexicographical ordering. The latter results in grouping fluents with the same leading function symbol together. For some domains, however, there is little or no improvement; this is usually the case when there are no large sets of objects as parameters for fluents.

Problem	GRIPPER–10	BLOCKSWORLD–8	GET–PAID–4
lexicographical	217409	206995	25633
sort ordered	3087	23373	38367

Table 1: The Size of the BDD for $\mathcal{B}(\mathbf{T}(z, z'))$ with an ordering of the variables by name (lexicographical) or with the sort ordering heuristic. The problems are from the planning problem repository [11].

[3] The problem to find the optimal variable order is NP-complete.

4.2 Partitioning of the Transition Relation

The maximal size of a BDD is exponential in the number of propositional variables it contains. Thus, the BDD representing $\mathcal{B}(\mathbf{T}(z,z'))$, which contains twice as many propositional variables as the BDDs representing the S_i, is prone to get very large. A way to reduce this problem is to divide the disjunction $\mathbf{T}(z,z')$ into several parts $\mathbf{T}_1,\ldots,\mathbf{T}_n$, which correspond to subsets of the state update actions. Let $\mathcal{F}_{su,1}$, ..., $\mathcal{F}_{su,k}$ be a partition of $\mathcal{F}_{su}$ and define for all $1 \leq i \leq k$

$$\mathbf{T}_i(z,z') = \bigvee_{\phi(a)\in\mathcal{F}_{su,i}} \mathbf{T}_{\phi(a)}(z,z')$$

such that

$$\mathbf{T}(z,z') = \bigvee_{i=1}^{k} \mathbf{T}_i(z,z') \ .$$

Thus, (8) is modified to

$$Z_{i+1}[\vec{z}'] = \bigvee_{i=1}^{k}((\exists\vec{z})\ Z_i[\vec{z}] \wedge \mathcal{B}(\mathbf{T}_k(z,z'))[\vec{z},\vec{z}']) \ . \tag{9}$$

Fig. 2 illustrates the partitioning of the transition relation.

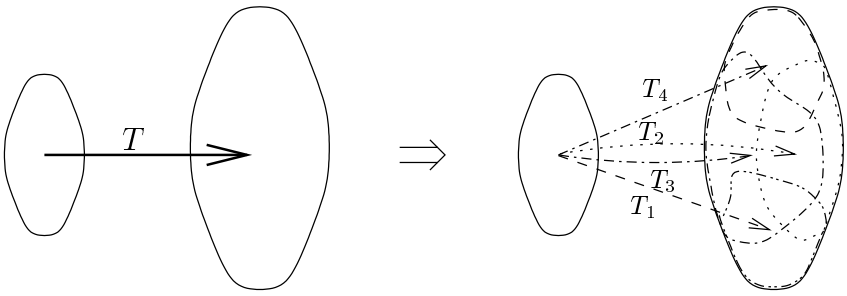

Figure 2: The partitioning of the transition relation. Each of the codomains of T_1, T_2, T_3 and T_4 is significantly smaller than the codomain of T.

The positive effect of the partitioning is that the actions in each subset effect only a subset of all fluents. Because the maximal size of a BDD is

exponential in the number of propositional variables, the sum of the sizes of the BDDs corresponding to the partition may be significantly smaller than the size of the original BDD.

In our implementation the number of partitions is adaptive: first the BDDs $\mathcal{B}(\mathbf{T}_{\phi(a)}(z, z'))$ for every single action are constructed, then they are combined until a parameter "partition treshold" is exceeded. In the experiments, partitioning led to a reduction of needed memory in most of the tested problems as shown in Fig. 3.

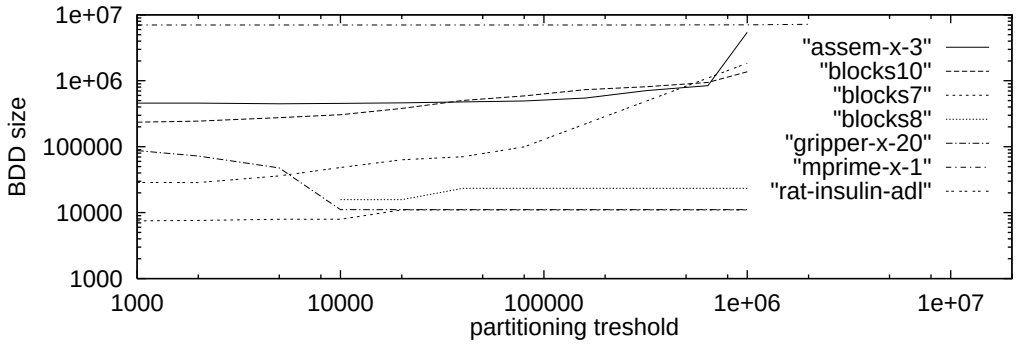

Figure 3: The sum of the sizes of the BDDs used to represent the transition relation in dependence on the parameter "partitioning treshold".

On the other hand, a reduction in memory size does not necessarily lead to a reduction in calculation time as the results depicted in Fig. 4 indicate. According to equation (9) the various parts of the partitioned transition relation have to be put together, and this takes time. Nevertheless splitting can be useful even if the computation time increases, because of the reduction of the needed memory to store the BDDs. For example in the case of MPRIME–X–1 the problem was not manageable under our memory constraints without partitioning the transition relation.

The idea to partition BDDs can also be applied to the BDDs representing Z_i. We have not yet explored this idea, because in our test problems these BDDs were only moderately large (i.e., up to 100.000 nodes).

We have also implemented an optimization technique called *frontier simplification* [7]. This technique explores the fact, that the algorithm for solving the entailment problem in the fluent calculus works also if the following two conditions are enforced for all $i \geq 0$:

- The set S_i contains all states which may be reached by executing i

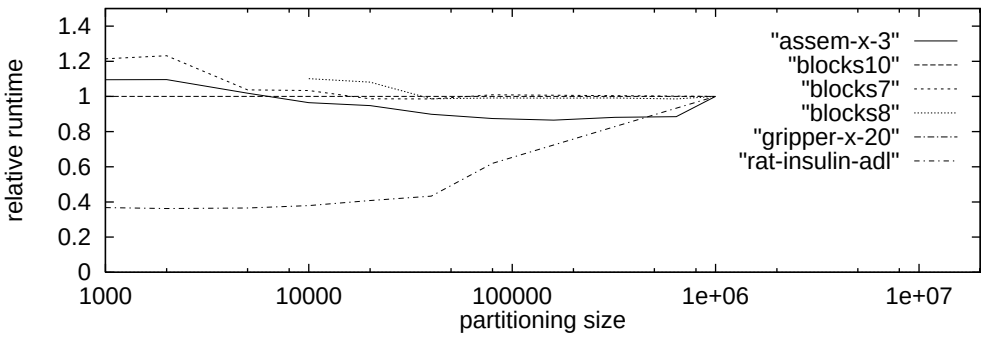

Figure 4: Effects of the parameter "partitioning treshold" on the calculation time for several problems. The time is relative to the time taken when no partitioning is done.

actions, but not by executing less than i actions.

- the set S_i does not contain any states which cannot be reached by executing at most i actions.

The sets S_i can be chosen freely within these limitations. Hence, it is desirable that the algorithm chooses the S_i such that their BDD representations are as small as possible. Frontier simplification promises to lead to moderate improvements but we have no experimental data on planning problems yet.

5 Results on the Gripper Class

The problems of the GRIPPER class were quite hard problems for the planners taking part in the AIPS98 competition. Their difficulty is rooted in the combinatorial explosion of alternatives due to the existence of two grippers. In Fig. 5 the runtimes of these planners[4] are compared to our system, BDDPLAN.[5] Only one planner (HSP) was able to solve all of the problems of this class, but it generated only suboptimal plans by using only one of the two grippers, whereas BDDPLAN generates the shortest possible plan by design.

[4]See `http://ftp.cs.yale.edu/pub/mcdermott/aipscomp-results.html`.

[5]The runtime of BDDPLAN is measured on a different machine, so it is only accurate up to a constant factor.

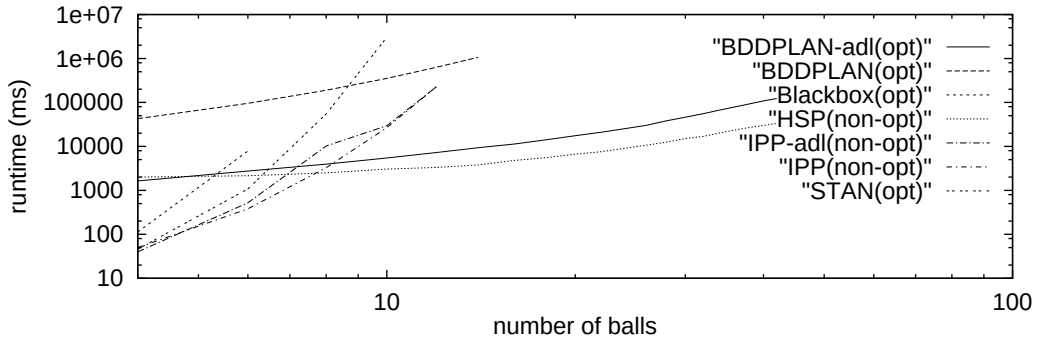

Figure 5: Runtimes different planners on the `Gripper` problem (in milliseconds) with different numbers of balls. Planners marked with `opt` provided optimal (i.e. shortest) plans, planners marked with `-adl` work on the sorted version of the domains, the others on the STRIPS-version.

6 Discussion

We have presented in this paper our preliminary findings in applying BDD techniques as an implementation tool for reasoning about situations, actions and causality in the fluent calculus, and discussed several techniques that have been successfully used to improve the performance of the implementation.

We tested our implementation using the problems of the planning contest on AIPS98 and have received mixed results so far. As discussed in section 5, our planner performed very good in the GRIPPER class: It was able to provide the shortest solutions to even the most difficult problems posed in this class, whereas the planners which have participated in the competition were only able to solve but the simplest problems or, in the exceptional case of HSP, provided sub–optimal solutions ignoring the second gripper of the robot. In some other problem classes, however, our implementation did not outperform existing systems. On the other hand, we have just started to investigate optimization techniques and will continue to do so in the future. In particular, optimization techniques used in GRAPHPLAN [1] and other propositional reasoning systems are potential candidates to investigate.

The optimization techniques presented in this paper do not change the principle of breadth first search the algorithm is based on. This has the pleasant effect that

- the algorithm always finds the shortest plan, and

- it is possible to reuse the results of the computationally intensive forward pass stage, in which the sequence of sets of reachable states $(S_i \mid i \geq 0)$ is constructed, to either create many possible solutions to the same reasoning problem or to solve multiple reasoning problems with the same initial state.

On the other hand, in order to speed up the search it seems one should give up the concept of breadth first search and explore *interesting* parts of the search space first. This can be done without giving up completeness by stepwise adding actions to the transition relation, which seem heuristically relevant for reaching the goal, and explore the subtrees of the search space generated by these actions first.

It should be noted that although we have presented our algorithm in such a way that there is only a single initial state (i.e., the set S_0 is unitary), the algorithm itself is by no means restricted to this case. If the initial situation is only incompletely specified then there are several initial states, which leads to a set S_0 containing more than one element.

We are aware of only another approach to reasoning about situations, actions and causality using BDDs, viz. the system presented in [5, 6]. This system generates so–called *universal* plans, which consist of a state–action table that contains for each state the action, which when executed leads to the goal in the shortest way. This has the advantage that it works for non–deterministic problems as well. We conjecture, that this approach is limited to less complex reasoning problems, because the executed actions have to be encoded into the transition relation. This leads to a considerable increase in the number of propositional variables and, consequently, in the maximal size of the BDDs. But we have not yet performed direct comparisons to bolster this conjecture.

To sum up, our BDD based implementation shows some promising initial results but it is too early to completely evaluate it yet.

References

[1] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.

[2] Randal E. Bryant. Graph–based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 8(C-35):677–691, 1986.

[3] J. Burch, E. Clarke, K. McMillan, and D. Dill. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[4] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(4):401–424, April 1994.

[5] A. Cimatti, E. Giunchiglia, F. Giunchiglia, and P. Traverso. "planning via model checking: A decision procedure for ar. In S. Steel and R. Alami, editors, *Proceedings of the Fourth European Conference on Planning (ECP97)*, number 1348 in Lecture Notes in Artificial Intelligence, pages 130–142, Toulouse, France, Sept. 1997. Springer-Verlag.

[6] Alessandro Cimatti, Marco Roveri, and Paolo Traverso. Automatic OBDD-based generation of universal plans on non–deterministic domains. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI98)*, Madison, Wisconsin, July 26-30 1998. (to appear).

[7] E. Clarke, O. Grunberg, and D. Long. Model checking. In *Proceedings of the International Summer School on Deductive Program Design*, Marktoberdorf, 1994.

[8] T. Eiter, N. Leone, C. Mateis, G. Pfeier, and F. Scarnello. The KR system DLV: Progress report, comparisons and benchmarks. In *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning*, pages 406–417. Morgan Kaufmann Publishers, 1998.

[9] S. Hölldobler and J. Schneeberger. A new deductive approach to planning. *New Generation Computing*, 8:225–244, 1990. A short version appeared in the Proceedings of the German Workshop on Artificial Intelligence, Informatik Fachberichte *216*, pages 63-73, 1989.

[10] S. Hölldobler and H.-P. Störr. Solving the entailment problem in the fluent calculus using binary decision diagrams. Technical Report WV-99-05, Artificial Intelligence Institute, Computer Science Department, Dresden University of Technology, 1999. http://pikas.inf.tu-dresden.de/publikationen/TR/1999/wv-99-05.ps.

[11] Drew McDermott. Planning problem repository. `ftp://ftp.cs.yale.edu/pub/mcdermott/domains/`.

[12] I. Niemelä and P. Simons. Smodels — an implementation of the well–founded and stable model semantics. In *Proceedings of the 4th International Conderence on Logic Programming and Non–monotonic Reasoning*, pages 420–429, 1997.

[13] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 440–446, 1992.

[14] M. Thielscher. Introduction to the fluent calculus. *Electronic Transactions on Artificial Intelligence*, 2(3-4):179–192, 1998.