OBDD-based Universal Planning in Multi-Agent, Non-Deterministic Domains

Master's Thesis

Rune Møller Jensen
Technical University of Denmark
Department of Automation
Building 326/327
DK-2800 Lyngby
Denmark

June 7, 1999

Preface

With a few changes Manuela Veloso and I have published Section 1, 3, 4, 5, 6, 8.1.1, 8.1.2, 8.2, 9 and 10 to the Journal of Artificial Intelligence Research in June 1999. For this reason the thesis uses an article-oriented language, which I hope, the reader will enjoy.

Rune M. Jensen
Pittsburgh, PA, USA

Acknowledgements

The thesis work has been carried out in the Spring 1999, while I was visiting Manuela Veloso at the School of Computer Science, Carnegie Mellon University. I am very indebted to her for her hospitality and guidance on the work.

I am also indebted to my supervisor Morten Lind. Without his help and support, this work, or any of the work leading to it, would not have been possible.

A special thanks to Marco Roveri for introducing me to MBP and for many rewarding discussions on OBDD techniques and OBDD-based planning.

Also a special thanks to Randal E. Bryant and Ed Clarke for showing interest in the project and giving advice on suitable model checking techniques for OBDD-based planning.

For additional advice on OBDD issues, formal representation and proof reading I wish to thank Henrik R. Andersen, Jørn Lind-Nielsen, Lars Birkedal and Kelly Richmond.

Finally, I wish to thank the CORAL group and people associated to the Robotics Soccer Lab.: Anna H. R. Costa, Sorin Achim, Laurie Hiyakumoto, Peter Stone, Tucker Balch, Jan Koehler, Belinda Thom, William Uther, Kwun Han, Michael Bowling, Bryan Singer, Scott Lenser and Elly Winner.

Abstract

Recently model checking representation and search techniques were shown to be efficiently applicable to planning, in particular to non-deterministic planning. Such planning approaches use Ordered Binary Decision Diagrams (OBDDs) to encode a planning domain as a non-deterministic finite automaton (NFA) and then apply fast algorithms from model checking to search for a solution. OBDDs can effectively scale and can provide universal plans for complex planning domains. This thesis presents UMOP[1], a new universal OBDD-based planning framework for non-deterministic, multi-agent domains, which is also applicable to deterministic single-agent domains as a special case. A new planning domain description language, *NADL*[2], is introduced to specify non-deterministic, multi-agent domains. The language contributes the explicit definition of controllable agents and uncontrollable environment agents. The syntax and semantics of *NADL* is described, and it is shown how to build an efficient OBDD-based representation of an *NADL* description. The UMOP planning system uses *NADL* and different OBDD-based universal planning algorithms. It includes the previously developed strong and strong cyclic planning algorithms (Cimatti et al., 1998a, 1998b). In addition, a new optimistic planning algorithm is introduced, which relaxes optimality guarantees and generates plausible universal plans in some domains where no strong or strong cyclic solution exist. Empirical results are presented from domains ranging from deterministic and single-agent with no environment actions to non-deterministic and multi-agent with complex environment actions. UMOP is shown to be a rich and efficient planning system.

[1] UMOP stands for Universal Multi-agent OBDD-based Planner.
[2] *NADL* stands for Non-deterministic Agent Domain Language.

Contents

1 Introduction

Classical planning is a broad area of research which involves the automatic generation of the appropriate choices of actions to traverse a state space to achieve specific goal states. A variety of different algorithms have been developed to address the state-action representation and the search for action selection.

Traditionally these algorithms have been classified according to their search space representation as either state-space planners (e.g., PRODIGY, Veloso et al., 1995) or plan-space planners (e.g., UCPOP, Penberthy & Weld, 1992).

A new research trend has been to develop new encodings of planning problems in order to adopt efficient algorithms from other research areas, leading to significant developments in planning algorithms, as surveyed by Weld (1999). This class of planning algorithms includes GRAPHPLAN (Blum & Furst, 1995), which uses a flow-graph encoding to constrain the search and SATPLAN (Kautz & Selman, 1996), which encodes the planning problem as a satisfiability problem and uses fast model satisfaction algorithms to find a solution.

Recently, another new planner MBP (Cimatti et al., 1997) was introduced that encodes a planning domain as a non-deterministic finite automaton (NFA) represented by an Ordered Binary Decision Diagram (OBDD) (Bryant, 1986). In contrast to the previous algorithms, MBP effectively extends to non-deterministic domains producing universal plans as robust solutions. Due to the scalability of the underlying model checking representation and search techniques, it can be shown to be a very efficient universal planner (Cimatti et al., 1998a, 1998b).

A universal plan is a set of state-action rules that aim at covering the possible multiple situations in the non-deterministic environment. A universal plan is executed by interleaving the selection of an action in the plan and observing the resulting effects in the world. Universal planning resembles the outcome of reinforcement learning (Sutton & G., 1998), in that the state-action model captures the uncertainty of the world. Universal planning is a precursor approach[3], where all planning is done prior to execution, building upon the assumption that a non-deterministic model can be acquired, and leading therefore to a sound and complete planning approach.

[3] The term *precursor* originates from Dean et al. (1995) in contrast to *recurrent* approaches which replan to recover from execution failures.

However, universal planning has been criticized (e.g., Ginsberg, 1989), due to a potential exponential growth of the universal plan size with the number of propositions defining a domain state. An important contribution of MBP is thus the use of OBDDs to represent universal plans. In the worst case, this representation may also grow exponential with the number of domain propositions, but because OBDDs are very compact representations of boolean functions, this is often not the case for domains with a regular structure (Cimatti et al., 1998a). Therefore, OBDD-based planning seems to be a promising approach to universal planning.

An interesting problem is to extend the OBDD-based planning approach to multi-agent, non-deterministic domains, where the environment is explicitly modelleds. *The goal of this thesis is to define a new domain description language suitable for modelling such domains and implement an* OBDD*-based universal planning system for solving planning problems defined in this language.*

The developed planner is called UMOP (UMOP stands for Universal Multi-agent OBDD-based Planner.). The overall approach for designing UMOP is similar to the approach introduced by Cimatti et al. (1998a, 1998b). The main contribution is the domain description language, *NADL* (*NADL* stands for Non-deterministic Agent Domain Language.). *NADL* has more resemblance with previous planning languages than the action description language $\mathcal{AR}$ currently used by MBP. It has powerful action descriptions that can perform arithmetic operations on numerical domain variables. Domains comprised of synchronized agents can be modelled by introducing concurrent actions based on a multi-agent decomposition of the domain.

In addition, *NADL* introduces a separate and explicit environment model defined as a set of *uncontrollable* agents, i.e., agents whose actions cannot be a part of the generated plan. *NADL* has been carefully designed to allow for efficient OBDD-encoding. Thus, in contrast to MBP, UMOP can generate a partitioned transition relation representation of the NFA, which is known from model checking to scale up well (Burch et al., 1991; Ranjan et al., 1995). Empirical experiments suggest that this is also the case for UMOP.

UMOP includes the previously developed algorithms for OBDD-based universal planning. In addition, a new "optimistic" planning algorithm is introduced, which relaxes optimality guarantees and generates plausible universal plans in some domains, where no solution can be found by the previous algorithms.

The thesis is organized as follows. Section 2 gives an introduction to classical Artificial Intelligence planning (AI planning). It may be skipped by

readers already familiar with the subject. Section 3 gives a brief overview of OBDDs. Section 4 introduces *NADL*, shows how to encode a planning problem, and formally describes the syntax and semantics of this description language in terms of an NFA. The properties of the language are also discussed based on an example and arguments are given for the design choices. Section 5 presents the OBDD representation of *NADL* domain descriptions. Section 6 describes the different algorithms that have been used for OBDD-based planning and introduces the optimistic planning algorithm. Section 7 describes the implementation of UMOP and its facility for analyzing universal plans. Section 8 presents empirical results in several planning domains, ranging from single-agent and deterministic ones to multi-agent and non-deterministic ones. Section 9 discusses previous approaches to planning in non-deterministic domains. Finally, Section 10 draws conclusions and discusses directions for future work.

2 Introduction to Classical AI Planning

This section gives a brief introduction to the basic concepts of AI planning and presents two classical approaches. It may be skipped by readers already familiar with the topic.

2.1 Basic Concepts

A planning problem consists of finding some ordering of actions that changes the state of a domain from some initial state to some goal state. As an example consider the problem of finding a plan to move a robot with actions *up*, *down*, $left$ and $right$ from grid position $(1,1)$ to $(2,3)$. One solution is the totally ordered plan $(up, up, right)$, but any ordering of the three actions is a solution.

The input to a planning system is a specification of the initial and goal state and a *domain theory*. The domain theory defines the state space and the actions of the domain and is a discrete representation of the target world, where actions are modelled as transitions between states.

Classical planners use the STRIPS language (Fikes & Nilsson, 1971), or STRIPS inspired languages, to represent the domain theory. In the STRIPS language states are described in logic by conjunctions of function-free ground literals. In the classical blocks world the initial state shown in Figure 1 could

be described by:

$$on(C, A) \wedge on(B, Table) \wedge on(A, Table)$$

A STRIPS action has three parts: A *precondition* that defines when the

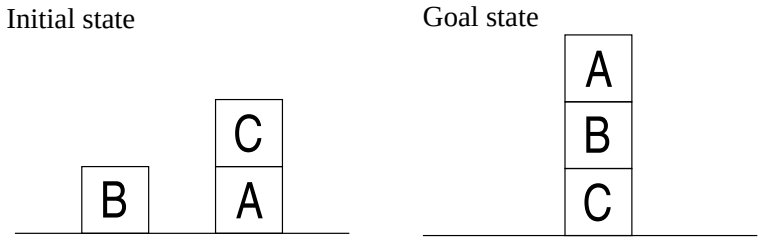

Figure 1: A blocks world domain planning problem.

action is applicable, a set of atoms made true by the action (the *add-list*) and a set of actions made false by the action (the *delete list*). Only atoms mentioned in the add and delete list are assumed to change truth value. An action in the blocks world for stacking block x onto block y is shown below:

ACTION:Stack(x,y),
PRECOND:holding(x) $\wedge$ clear(y),
ADD:{armempty,on(x,y),clear(x)}
DELETE:{holding(x),clear(y)}

For the early approaches to planning (e.g. planning as theorem proving in situation calculus) a major problem, known as the *frame problem*, was how to efficiently represent the unchanged knowledge by actions. It is STRIPS implicit solution of the frame problem that is the main reason for its popularity, as the amount of knowledge changed by an action normally is small compared to the amount of unchanged knowledge.

2.2 State Space Planners

A *state space planner* searches in the state space of a domain. A planner starting from an initial state and applying actions successively until a goal state is reached is called a *progression planner* or a *forward chaining planner*. Often search trees with a lower branching factor can be obtained be searching backward from the goal by applying actions that can achieve some goal literal. Planners using this strategy are called *regression planners* or *backward chaining planners*. Backward chaining planners only considering

orderings of subgoals are known as linear planners. Not all problems can be solved using this strategy. Consider for example the planning problem shown in Figure 1. Suppose a planner first achieves $on(A, B)$ by taking C off A and putting A on B, it then cannot achieve the subgoal $on(B, C)$ without undoing the action. A similar situation arises, if it starts with the goal $on(B, C)$. The problem is known as the *Sussman anomaly.*

Planners able to interleave the plan steps from each subgoal in a way that satisfies all subgoals are called *nonlinear planners.* A subset of these planners can avoid having to totally order all steps in a plan. These planners are known as *partial-order planners.*

2.3 Plan Space Planners

A partial-order planner searches through the space of plans rather than the space of states. The planner starts with a simple, incomplete plan, which is modified until a complete plan is found that solves the problem. The operators in this search are operators on plans: Adding an action, adding an ordering constraint on actions, binding previously unbound variables etc.. The solution is a plan consisting of a partial-order of actions.

Partial-order planners are capable of solving the Sussman anomaly. A solution showing causal and ordering links between actions is depicted in Figure 2.

3 Introduction to OBDDs

An Ordered Binary Decision Diagram (Bryant, 1986) is a canonical representation of a boolean function with n linear ordered arguments $x_1, x_2, ..., x_n$.

An OBDD is a rooted, directed acyclic graph with one or two terminal nodes of out-degree zero labeled 1 or 0, and a set of variable nodes u of out-degree two. The two outgoing edges are given by the functions $high(u)$ and $low(u)$ (drawn as solid and dotted arrows). Each variable node is associated with a propositional variable in the boolean function the OBDD represents. The graph is ordered in the sense that all paths in the graph respect the ordering of the variables.

An OBDD representing the function $f(x_1, x_2) = x_1 \wedge x_2$ is shown in Figure 3. Given an assignment of the arguments x_1 and x_2, the value of f is determined by a path starting at the root node and iteratively following the high edge, if the associated variable is true, and the low edge, if the

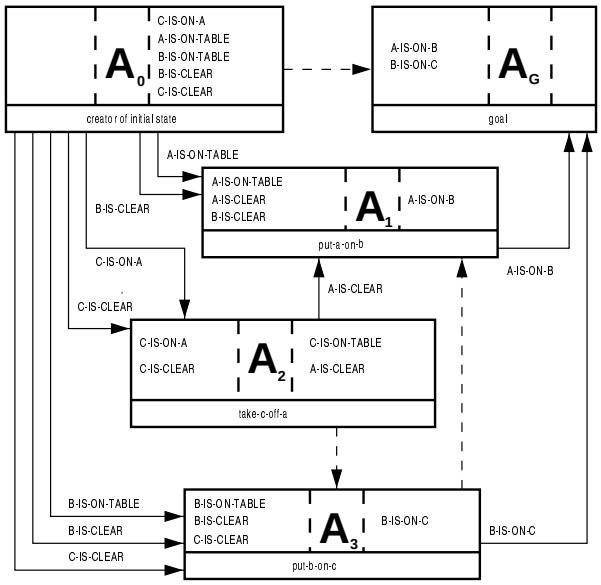

Figure 2: The final plan generated by a partial-order planner for solving the Sussman anomaly. Solid arrows denote causal links between actions (and thus also ordering links) while dashed arrows denote ordering links.

associated variable is false. The value of f is *True* if the label of the reached terminal node is 1; otherwise it is *False*.

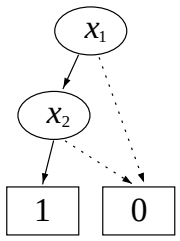

Figure 3: An OBDD representing the function $f(x_1, x_2) = x_1 \wedge x_2$. High (true) and low (false) edges are drawn solid and dotted, respectively.

An OBBD graph is reduced so that no two distinct nodes u and v have the same variable name and low and high successors (Figure 4(a)), and no variable node u has identical low and high successors (Figure 4(b)).

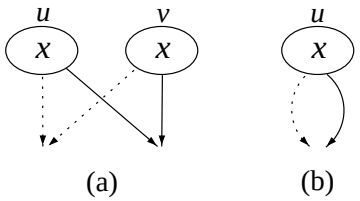

Figure 4: Reductions of OBDDs. (a): nodes associated to the same variable with equal low and high successors will be converted to a single node. (b): nodes causing redundant tests on a variable, are eliminated.

The OBDD representation has two major advantages: First, it is an efficient representation of boolean functions because the number of nodes often is much smaller than the number of truth assignments of the variables. The number of nodes can grow exponential with the number of variables, but most commonly encountered functions have a reasonable representation (Bryant, 1986). Second, any operation on two OBDDs, corresponding to a boolean operation on the functions they represent, has a low complexity bounded by the product of their node counts.

A disadvantage of OBDDs is that the size of an OBDD representing some function is very dependent on the ordering of the variables. To find an optimal variable ordering is a co-NP-complete problem in itself, but fortunately a good heuristic is to locate dependent variables near each other in

the ordering.

OBDDs have been successfully applied to model checking. In model checking the behavior of a system is modelled by a finite state automaton with a transition relation represented as an OBDD. Desirable properties of the system is checked by analyzing the state space of the system by means of OBDD manipulations.

As introduced by Cimatti et al. (1998a, 1998b), a similar approach can be used for a non-deterministic planning problem. Given an NFA representation of the planning domain with the transition relation represented as an OBDD, the algorithms used to verify CTL properties in model checking (Clarke et al., 1986; McMillan, 1993) can be used to find a universal plan solving the planning problem.

4 *NADL*

In this section, the properties of *NADL* are discussed based on an informal definition of the language and a domain encoding example. The formal syntax and semantics of *NADL* is then described.

An *NADL* domain description consists of: a definition of *state variables*, a description of *system* and *environment agents*, and a specification of an *initial* and *goal conditions*.

The set of state variable assignments defines the state space of the domain. An agent's description is a set of *actions*. The agents change the state of the world by performing actions, which are assumed to be executed synchronously and to have a fixed and equal duration. At each step, all of the agents perform exactly one action, and the resulting action tuple is a *joint action*. The system agents model the behavior of the agents controllable by the planner, while the environment agents model the uncontrollable world. A valid domain description requires that the system and environment agents constrain a disjoint set of variables.

An action has three parts: a set of *state variables*, a *precondition* formula, and an *effect* formula. Intuitively the action takes responsibility of constraining the values of the set of state variables in the next state. It further has exclusive access to these variables during execution. In order for the action to be applicable, the precondition formula must be satisfied in the current state. The effect of the action is defined by the effect formula which must be satisfied in the next state. To allow conditional effects, the effect expression can refer to both current and next state variables, which need

to be a part of the set of variables of the action. All next state variables not constrained by any action in a joint action maintain their value. Furthermore only joint actions containing a set of actions with consistent effects and a disjoint set of state variable sets are allowed. System and environment agents must be independent in the sense that the two sets of variables, their actions constrain, are disjoint.

The initial and goal conditions are formulas that must be satisfied in the initial state and the final state, respectively.

There are two sources of non-determinism in *NADL* domains: non-determinism caused by actions not restricting all their constrained variables to a specific value in the next state, and non-determinism caused by a non-deterministic selection of environment actions.

A simple example of an *NADL* domain description is shown in Figure 5[4]. The domain describes a planning problem for Schoppers' (1987) robot-baby domain. The domain has two state variables: a numerical one, *pos*, with range $\{0, 1, 2, 3\}$ and a propositional one, *robot_works*. The robot is the only system agent and it has two actions *Lift-Block* and *Lower-Block*. The baby is the only environment agent and it has one action *Hit-Robot*. Because each agent must perform exactly one action at each step, there are two joint actions (*Lift-Block*,*Hit-Robot*) and (*Lower-Block*,*Hit-Robot*).

Initially the robot is assumed to hold a block at position 0, and its task is to lift it up to position 3. The *Lift-Block* (and *Lower-Block*) action has a conditional effect described by an if-then-else operator: if *robot_works* is true, *Lift-Block* increases the block position with one, otherwise the block position is unchanged. Initially *robot_works* is assumed to be true, but it can be made false by the baby. The baby's action *Hit-Robot* is non-deterministic, as it only constrains *robot_works* by the effect expression $\neg robot\_works \Rightarrow \neg robot\_works'$. Thus, when *robot_works* is true in the current state, the effect expression of *Hit-Robot* does not apply, and *robot_works* can either be true or false in the next state. On the other hand, if *robot_works* is false in the current state, *Hit-Robot* keeps it false in the next state. The *Hit-Robot* models an uncontrollable environment, in this case a baby, by its effects on *robot_works*. In the example above, *robot_works* stays false when it, at some point, has become false, reflecting that the robot cannot spontaneously be fixed by a hit of the baby.

[4] Unquoted and quoted variables refer to the current and next state, respectively. Another notation like v_t and v_{t+1} could have been used. We have chosen the quote notation because it is the common notation in model checking.

variables
  nat(4) *pos*
  bool *robot_works*
system
  agt: Robot
    Lift-Block
      con: *pos*
      pre: $pos < 3$
      eff: $robot\_works \rightarrow pos' = pos + 1, pos' = pos$
    Lower-Block
      con: *pos*
      pre: $pos > 0$
      eff: $robot\_works \rightarrow pos' = pos - 1, pos' = pos$
environment
  agt: Baby
    Hit-Robot
      con: *robot_works*
      pre: *true*
      eff: $\neg robot\_works \Rightarrow \neg robot\_works'$
initially
  $pos = 0 \wedge robot\_works$
goal
  $pos = 3$

Figure 5: An *NADL* domain description.

An NFA representing the domain is shown in Figure 6. The calculation of the next state value of *pos* in the *Lift-Block* action shows that numerical variables can be updated by an arithmetic expression on the current state variables. The update expression of *pos* and the use of the if-then-else operator further demonstrate the advantage of using explicit references to current state and next state variables in effect expressions. *NADL* does not restrict the representation by enforcing a structure separating current state and next state expressions. The if-then-else operator has been added to support complex, conditional effects that often are efficiently and naturally represented as a set of nested if-then-else operators.

The explicit representation of constrained state variables enables any non-deterministic or deterministic effect of an action to be represented, as the constrained variables can be assigned to any value in the next state that

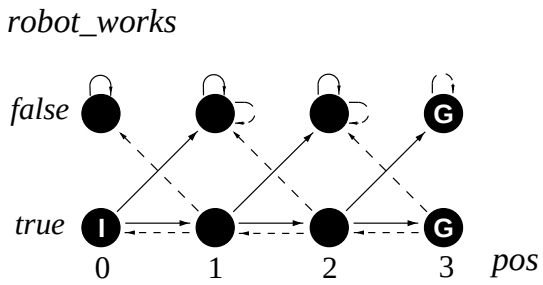

Figure 6: The NFA of the robot-baby domain (see Figure 5). There are two state variables: a propositional state variable *robot_works* and a numerical state variable *pos* with range $\{0, 1, 2, 3\}$. The (*Lift-Block*,*Hit-Robot*) and (*Lower-Block*,*Hit-Robot*) joint actions are drawn with solid and dashed arrows respectively. States marked with "I" and "G" are initial and goal states.

satisfies the effect formula. It further turns out to have a clear intuitive meaning as the action takes the "responsibility" of specifying the values of the constrained variables in the next state.

Compared to the action description language $\mathcal{A}$ (Gelfond & Liftschitz, 1993) and $\mathcal{AR}$ (Giunchiglia et al., 1997) that are the only prior languages used for OBDD-based planning (Di Manzo et al., 1998; Cimatti et al., 1998a, 1998b, 1997), *NADL* introduces an explicit environment model, a multi-agent decomposition and numerical state variables. It can further be shown that *NADL* can be used to model any domain that can be modelled with $\mathcal{AR}$ (see Appendix B).

The concurrent actions in *NADL* are assumed to be synchronously executed and to have fixed and equal duration. A general representation allowing partially overlapping actions and actions with different durations has been avoided, as it requires more complex temporal planning (see e.g., O-PLAN or PARCPLAN Currie & Tate, 1991; Lever & Richards, 1994). Our joint action representation has more resemblance with $\mathcal{A}_C$ (Baral & Gelfond, 1997) and $\mathcal{C}$ (Giunchiglia & Lifschitz, 1998), where sets of actions are performed at each time step. In contrast to these approaches, though, we model multi-agent domains.

An important issue to address when introducing concurrent actions is synergetic effects between simultaneously executing actions (Lingard & Richards, 1998). A common example of destructive synergetic effects is when two or

more actions require exclusive use of a single resource or when two actions have inconsistent effects like $pos' = 3$ and $pos' = 2$. In *NADL* actions cannot be performed concurrently if: 1) they have inconsistent effects, or 2) they constrain an overlapping set of state variables. The first condition is due to the fact that state knowledge is expressed in a monotonic logic which cannot represent inconsistent knowledge. The second rule addresses the problem of sharing resources. Consider for example two agents trying to drink the same glass of water. If only the first rule defined interfering actions both agents, could simultaneously empty the glass, as the effect *glass_empty* of the two actions would be consistent. With the second rule added, these actions are interfering and cannot be performed concurrently.

The current version of *NADL* only avoids destructive synergetic effects. It does not include ways of representing constructive synergetic effects between simultaneous acting agents (Lingard & Richards, 1998). A constructive synergetic effect is illustrated in Baral and Gelfond (1997), where an agent spills soup from a bowl when trying to lift it up with one hand, but not when lifting it up with both hands. In $\mathcal{C}$ and $\mathcal{A}_\mathcal{C}$ this kind of synergetic effects can be represented by explicitly stating the effect of a compound action. A similar approach could be used in *NADL*, but is currently not supported.

4.1 Syntax

A BNF definition of the concrete syntax of *NADL* is shown in Appendix A. Identifiers are alphanumeric sequences starting with a letter. Numbers are natural numbers. The syntax of formulas is defined in the following formal definition of an *NADL* description.

An *NADL* description is a 7-tuple $D = (SV, S, E, Act, d, I, G)$, where:

- $SV = PVar \cup NVar$ is a finite set of state variables comprised of a finite set of propositional variables, $PVar$, and a finite set of numerical variables, $NVar$.
- S is a finite, nonempty set of system agents.
- E is a finite set of environment agents.
- Act is a set of action descriptions (c, p, e) where c is the state variables constrained by the action, p is a precondition state formula in the set $SForm$ and e is an effect formula in the set $Form$. Thus $(c, p, e) \in Act \subset 2^{SV} \times SForm \times Form$. The sets $SForm$ and $Form$ are defined below.

- $d : Agt \rightarrow 2^{Act}$ is a function mapping agents ($Agt = S \cup E$) to their actions. Because an action is associated to one agent, d must satisfy the following conditions:

$$\bigcup_{\alpha \in Agt} d(\alpha) = Act$$

$$\forall \alpha_1, \alpha_2 \in Agt \,.\, d(\alpha_1) \cap d(\alpha_2) = \emptyset$$

- $I \in SForm$ is the initial condition.
- $G \in SForm$ is the goal condition.

For a valid domain description, we require that actions of system agents are independent of actions of environment agents:

$$\bigcup_{\substack{e \in E \\ a \in d(e)}} c(a) \;\cap\; \bigcup_{\substack{s \in S \\ a \in d(s)}} c(a) = \emptyset,$$

where $c(a)$ is the set of constrained variables of action a. The set of formulas $Form$ are constructed from the following alphabet of symbols:

- A finite set of current state v and next state v' variables, where $v \in SV$.
- The natural numbers $\mathbf{N}$.
- The arithmetic operators $+$, $-$, $/$, $*$ and mod.
- The relation operators $>$, $<$, $\leq$, $\geq$, $=$ and $\neq$.
- The boolean operators $\neg$,$\vee$,$\wedge$,$\Rightarrow$,$\Leftrightarrow$ and $\rightarrow$.
- The special symbols *true*, *false*, parenthesis and comma.

The set of arithmetic expressions is constructed from the following rules:

1. Every numerical state variable $v \in NVar$ is an arithmetic expression.
2. A natural number is an arithmetic expression.
3. If e_1 and e_2 are arithmetic expressions and $\oplus$ is an arithmetic operator, then $e_1 \oplus e_2$ is an arithmetic expression.

Finally, the set of formulas *Form* is generated by the rules:

1. *true* and *false* are formulas.
2. Propositional state variables $v \in PVar$ are formulas.
3. If e_1 and e_2 are arithmetic expressions and $\mathcal{R}$ is a relation operator then $e_1 \; \mathcal{R} \; e_2$ is a formula.
4. If f_1, f_2 and f_3 are formulas, so are $(\neg f_1)$, $(f_1 \vee f_2)$, $(f_1 \wedge f_2)$, $(f_1 \Rightarrow f_2)$, $(f_1 \Leftrightarrow f_2)$ and $(f_1 \rightarrow f_2, f_3)$.

Parenthesis have their usual meaning and operators have their usual priority and associativity with the if-then-else operator "$\rightarrow$" given lowest priority. $SForm \subset Form$ is a subset of the formulas only referring to current state variables. These formulas are called *state formulas.*

4.2 Semantics

All of the symbols in the alphabet of formulas have their usual meaning with the if-then-else operator $f_1 \rightarrow f_2, f_3$ being an abbreviation for $(f_1 \wedge f_2) \vee (\neg f_1 \wedge f_3)$. Each numerical state variable $v \in NVar$ has a finite range $rng(v) = \{0, 1, \cdots, t_v\}$, where $t_v > 0$.

The formal semantics of a domain description $D = (SV, S, E, Act, d, I, G)$ is given in terms of an NFA M:

Definition 1 (NFA) *A Non-deterministic Finite Automaton is a 3-tuple, $M = (Q, \Sigma, \delta)$, where Q is the set of states, Σ is a set of input values and $\delta : Q \times \Sigma \rightarrow 2^Q$ is a next state function.*

In the following construction of M we express the next state function as a transition relation. Let $\mathcal{B}$ denote the set of boolean values $\{True, False\}$. Further, let the *characteristic function* A: $B \rightarrow \mathcal{B}$ associated to a set $A \subseteq B$ be defined by: $A(x) = (x \in A)$[5]. Given an NFA M we define its *transition relation* $T \subseteq Q \times \Sigma \times Q$ as a set of triples with characteristic function $T(s, i, s') = (s' \in \delta(s, i))$.

The states Q of M equals the set of all possible variable assignments $Q = (PVar \rightarrow \mathcal{B}) \times (Nvar \rightarrow \mathbf{N})$. Σ of M is the set of joint actions of system agents represented as sets. That is, $\{a_1, a_2, \cdots, a_{|S|}\} \in \Sigma$ if and only

[5]Note: the characteristic function has the same name as the set.

if $(a_1, a_2, \cdots, a_{|S|}) \in \prod_{\alpha \in S} d(\alpha)$, where $|S|$ denotes the number of elements in S.

To define the transition relation $T : Q \times \Sigma \times Q \to \mathcal{B}$ of M we constrain a transition relation $t : Q \times J \times Q \to \mathcal{B}$ with the joint actions J of all agents as input by existential quantification to the input Σ.

$$T(s, i, s') = \exists j \in J \,.\, i \subset j \wedge t(s, j, s')$$

The transition relation t is a conjunction of three relations A, F and I. Given an action $a = (c, p, e)$ and a current state s, let $P_a(s)$ denote the value of the precondition formula p of a. Similarly, given an action $a = (c, p, e)$ and a current and next state s and s', let $E_a(s, s')$ denote the value of the effect formula e of a. $A : Q \times J \times Q \to \mathcal{B}$ is then defined by:

$$A(s, j, s') = \bigwedge_{a \in j} \Big(P_a(s) \wedge E_a(s, s') \Big)$$

A defines the constraints on the current state and next state of joint actions. A further ensures that actions with inconsistent effects cannot be performed concurrently as A reduces to false if any pair of actions in a joint action have inconsistent effects. Thus, A also states the first rule for avoiding interference between concurrent actions.

$F : Q \times J \times Q \to \mathcal{B}$ is a frame relation ensuring that unconstrained variables maintain their value. Let $c(a)$ denote the set of constrained variables of action a. We then have:

$$F(s, j, s') = \bigwedge_{v \notin C} (v = v'),$$

where $C = \bigcup_{a \in j} c(a)$.

$I : J \to \mathcal{B}$ ensures that concurrent actions constrain a non overlapping set of variables and thus states the second rule for avoiding interference between concurrent actions:

$$I(j) = \bigwedge_{(a_1, a_2) \in j^2} \Big(c(a_1) \cap c(a_2) = \emptyset \Big),$$

where j^2 denotes the set $\{(a_1, a_2) \mid (a_1, a_2) \in j \times j \wedge a_1 \neq a_2\}$. The transition relation t is thus given by:

$$t(s, j, s') = A(s, j, s') \wedge F(s, j, s') \wedge I(j)$$

5 OBDD Representation of *NADL* Descriptions

To build an OBDD $\tilde{T}$ representing the transition relation $T(s, i, s')$ of the NFA of a domain description $D = (SV, S, E, Act, d, I, G)$, we must define a set of boolean variables to represent the current state s, the joint action input i and the next state s'. As in Section 4.2 we first build a transition relation with the joint actions of both system and environment agents as input and then reduces this to a transition relation with only joint actions of system agents as input.

Joint action inputs are represented in the following way: assume action a is identified by a number p and can be performed by agent α. a is then defined to be the action of agent α, if the number expressed binary by a set of boolean variables A_α, used to represent the actions of α, is equal to p. Propositional state variables are represented by a single boolean variable, while numerical state variables are represented binary by a set of boolean variables.

Let A_{e_1} to $A_{e_{|E|}}$ and A_{s_1} to $A_{s_{|S|}}$ denote sets of boolean variables used to represent the joint action of system and environment agents. Further, let $x^k_{v_j}$ and $x'^k_{v_j}$ denote the *k'th* boolean variable used to represent state variable $v_j \in SV$ in the current and next state. An ordering of the boolean variables, known to be efficient from model checking, puts the input variables first followed by an interleaving of the boolean variables of current state and next stat variables:

$$\begin{aligned} A_{e_1} \prec \cdots \prec A_{e_{|E|}} \prec A_{s_1} \prec \cdots \prec A_{s_{|S|}} \\ \prec x^1_{v_1} \prec x'^1_{v_1} \prec \cdots \prec x^{m_1}_{v_1} \prec x'^{m_1}_{v_1} \\ \cdots \\ \prec x^1_{v_n} \prec x'^1_{v_n} \prec \cdots \prec x^{m_n}_{v_n} \prec x'^{m_n}_{v_n} \end{aligned}$$

where m_i is the number of boolean variables used to represent state variable v_i and n is equal to $|SV|$.

The construction of an OBDD representation $\tilde{T}$ is quite similar to the construction of T in Section 4.2. An OBDD representing a logical expression is built in the standard way. Arithmetic expressions are represented as lists of OBDDs defining the corresponding binary number. They collapse to single OBDDs when related by arithmetic relations.

To build an OBDD $\tilde{A}$ defining the constraints of the joint actions we need to refer to the values of the boolean variables representing the actions. Let

$i(\alpha)$ be the function that maps an agent α to the value of the boolean variables representing its action and let $b(a)$ be the identifier value of action a. Further let $\tilde{P}(a)$ and $\tilde{E}(a)$ denote OBDD representations of the precondition and effect formula of an action a. $\tilde{A}$ is then given by:

$$\tilde{A} = \bigwedge_{\substack{\alpha \in Agt \\ a \in d(\alpha)}} \Big(i(\alpha) = b(a) \Rightarrow \tilde{P}(a) \wedge \tilde{E}(a) \Big)$$

Note that logical operators now denote the corresponding OBDD operators. An OBDD representing the frame relation $\tilde{F}$ changes in a similar way:

$$\tilde{F} = \bigwedge_{v \in SV} \Big(\big(\bigwedge_{\substack{\alpha \in Agt \\ a \in d(\alpha)}} (i(\alpha) = b(a) \Rightarrow v \notin c(a)) \big) \Rightarrow s'_v = s_v \Big),$$

where $c(a)$ is the set of constrained variables of action a and $s_v = s'_v$ expresses that all current and next state boolean variables representing v are pairwise equal. The expression $v \notin c(a)$ evaluates to *True* or *False* and is represented by the OBDD for *True* or *False*.

The action interference constraint $\tilde{I}$ is given by:

$$\begin{aligned} \tilde{I} \;=\; & \bigwedge_{\substack{(\alpha_1, \alpha_2) \in S^2 \\ (a_1, a_2) \in c(\alpha_1, \alpha_2)}} \Big(i(\alpha_1) = b(a_1) \Rightarrow i(\alpha_2) \neq b(a_2) \Big) \wedge \\ & \bigwedge_{\substack{(\alpha_1, \alpha_2) \in E^2 \\ (a_1, a_2) \in c(\alpha_1, \alpha_2)}} \Big(i(\alpha_1) = b(a_1) \Rightarrow i(\alpha_2) \neq b(a_2) \Big), \end{aligned}$$

where $c(\alpha_1, \alpha_2) = \{(a_1, a_2) \mid (a_1, a_2) \in d(\alpha_1) \times d(\alpha_2) \wedge c(a_1) \cap c(a_2) \neq \emptyset\}$.

Finally the OBDD representing the transition relation $\tilde{T}$ is the conjunction of $\tilde{A}$, $\tilde{F}$ and $\tilde{I}$ with action variables of the environment agents existentially quantified:

$$\tilde{T} = \exists A_{e_1}, \cdots, A_{e_{|E|}} \,.\, \tilde{A} \wedge \tilde{F} \wedge \tilde{I}$$

Partitioning the transition relation

The algorithms we use for generating universal plans all consist of some sort of backward search from the states satisfying the goal condition to the

states satisfying the initial condition (see Section 6). Empirical studies in model checking have shown that the most complex operation for this kind of algorithms normally is to find the preimage of a set of visited states V.

Definition 2 (Preimage) *Given an NFA $M = (Q, \Sigma, \delta)$ and a set of states $V \subseteq Q$, the preimage of V is the set of states $\{s \mid s \in Q \wedge \exists i \in \Sigma, s' \in \delta(s, i) \,.\, s' \in V\}$.*

Note that states already belonging to V can also be a part of the preimage of V. Assume that the set of visited states are represented by an OBDD expression $\tilde{V}$ on next state variables and that we for iteration purposes, want to generate the preimage $\tilde{P}$ also expressed in next state variables. For a monolithic transition relation $\tilde{T}$ we then calculate:

$$\begin{aligned} \tilde{U} &= (\exists \vec{x}' \,.\, \tilde{T} \wedge \tilde{V})[\vec{x}/\vec{x}'] \\ \tilde{P} &= \exists \vec{i}' \,.\, \tilde{U} \end{aligned}$$

where $\vec{i}$, $\vec{x}$ and $\vec{x}'$ denote input, current state and next state variables, and $[\vec{x}'/\vec{x}]$ denotes the substitution of current state variables with next state variables. The set expressed by $\tilde{U}$ consists of state input pairs (s, i), for which the state s belongs to the preimage of V and the input i may cause a transition from s to a state in V. In the universal planning algorithms presented in the next section, the universal plans are constructed from elements in $\tilde{U}$.

The OBDD representing the transition relation $\tilde{T}$ and the set of visited states $\tilde{V}$ tend to be large, and a more efficient computation can be obtained by performing the existential quantification of next state variables early in the calculation (Burch et al., 1991; Ranjan et al., 1995). To do this the transition relation has to be split into a conjunction of partitions $T_1, T_2, ..., T_n$ allowing the modified calculation:

$$\begin{aligned} \tilde{U} &= (\exists \vec{x_n}' \,.\, \tilde{T}_n \wedge \cdots (\exists \vec{x_2}' \,.\, \tilde{T}_2 \wedge (\exists \vec{x_1}' \,.\, \tilde{T}_1 \wedge \tilde{V})) \cdots)[\vec{x}/\vec{x}'] \\ \tilde{P} &= \exists \vec{i}' \,.\, \tilde{U} \end{aligned}$$

That is, $\tilde{T}_1$ can refer to all variables, $\tilde{T}_2$ can refer to all variables except $\vec{x_1}'$, $\tilde{T}_3$ can refer to all variables except $\vec{x_1}'$ and $\vec{x_2}'$ and so on.

As shown by Ranjan et al. (1995) the computation time used to calculate the preimage is a convex function of the number of partitions. The reason for this is that, for some number of partitions, a further subdivision of the partitions will not reduce the total complexity, because the complexity

introduced by the larger number of OBDD operations is higher than the reduction of the complexity of each OBDD operation.

NADL has been carefully designed to allow a partitioned transition relation representation. The relations A, F and I all consist of a conjunction of subexpressions that normally only refer to a subset of next state variables. A partitioned transition relation that enables early variable quantification can be constructed by sorting the subexpressions according to which next state variables they refer to and combining them in partitions with near optimal sizes that satisfy the above requirements.

6 OBDD-based Universal Planning Algorithms

In this section two prior algorithms for OBDD-based universal planning are described. Furthermore it is discussed which kind of domains they are suitable for. Based on this discussion a new algorithm called *optimistic planning* is presented that seems to be suitable for some domains not covered by the prior algorithms.

The three universal planning algorithms discussed are all based on an iteration of preimage calculations. The iteration corresponds to a parallel backward breadth first search starting at the goal states and ending when all initial states are included in the set of visited states (see Figure 7). The main difference between the algorithms is the way the preimage is defined.

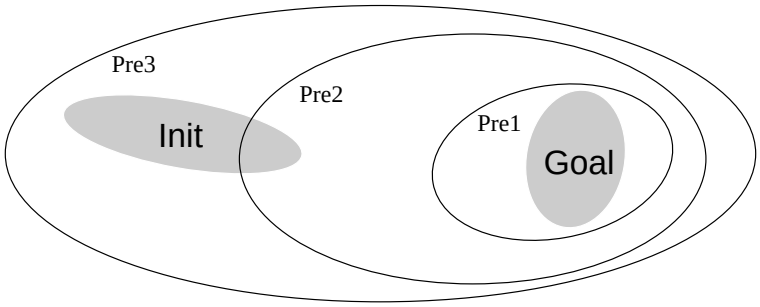

Figure 7: The parallel backward breadth first search used by universal planning algorithms studied in this article.

6.1 Strong Planning

Strong Planning (Cimatti et al., 1998b) uses a different preimage definition called strong preimage. For a state s belonging to the strong preimage of a set of states V, there exists at least one input i where all the transitions from s associated to i leads into V. When calculating the strong preimage of a set of visited states V, the set of state input pairs U represents the set of actions for each state in the preimage that, for any non-deterministic effect of the action, causes a transition into V. The universal plan returned by strong planning is the union of all these state-action rules. Strong planning is complete. If a strong plan exists for some planning problem the strong planning algorithm will return it, otherwise, it returns that no solution exists. Strong planning is also optimal due to the breadth first search. Thus, a strong plan with the fewest number of steps in the worst case is returned.

6.2 Strong Cyclic Planning

Strong cyclic planning (Cimatti et al., 1998a) is a relaxed version of strong planning, as it also considers plans with infinite length. Strong cyclic planning finds a strong plan if it exists. Otherwise, if the algorithm at some point in the iteration is unable to find a strong preimage it adds an ordinary preimage (referred to as a weak preimage). It then tries to prune this preimage by removing all states that have transitions leading out of the preimage and the set of visited states V. If it succeeds, the remaining states in the preimage are added to V and it again tries to add strong preimages. If it fails, it adds a new, weak preimage and repeats the pruning process. A partial search of strong cyclic planning is shown in Figure 8. A strong cyclic plan only guarantees progress towards the goal in the strong parts. In the weak parts, cycles can occur. To keep the plan length finite, it must be assumed that a transition leading out of the weak parts eventually will be taken. The algorithm is complete as a strong solution will be returned if it exists. If no strong or strong cyclic solution exist the algorithm returns that no solution exists.

6.3 Strengths and Limitations

An important reason for studying universal planning is that universal planning algorithms can be made generally complete. Thus, if a plan exists for painting the floor, an agent executing a universal plan will always avoid to paint itself into the corner or reach any other unrecoverable dead-end.

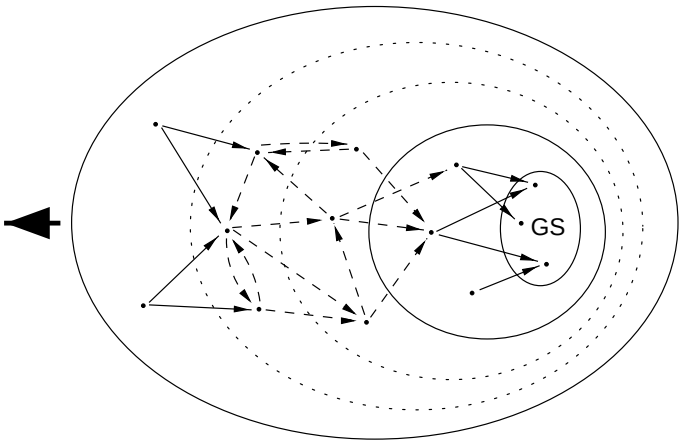

Figure 8: Preimage calculations in strong cyclic planning. Dashed ellipses denote weak preimages while solid ellipses denote strong preimages. Only one action is assumed to exist in the domain. All the shown transitions are included in the universal plan. Dashed transitions are from "weak" parts of the plan while solid transitions are from "strong" parts of the plan.

Strong planning and strong cyclic planning algorithms contribute by providing complete OBDD based algorithms for universal planning.

A limitation of strong and strong cyclic planning is their criteria for plan existence. If no strong or strong cyclic plan exist, these algorithms fail. The domains that strong and strong cyclic planning fail in are characterized by having unrecoverable dead-ends that cannot be guaranteed to be avoided.

Unfortunately, real world domains often have these kinds of dead-ends. Consider, for example, Schoppers' robot-baby domain described in Section 4. As depicted in Figure 6 no universal plan represented by a state-action set can guarantee the goal to be reached in a finite or infinite number of steps, as all relevant actions may lead to an unrecoverable dead-end.

A more interesting example is how to generate a universal plan for controlling, for example, a power plant. Assume that actions can be executed that can bring the plant from any bad state to a good state. Unfortunately the environment can simultaneously fail subsystems of the plant which makes the resulting joint action non-deterministic, such that the plant may stay in a bad state or even change to an unrecoverable failed state (see Figure 9). No strong or strong cyclic solution can be found because an unrecoverable state can be reached from any initial state. An *NADL* description of a power

plant domain is studied in Section 8.2.2.

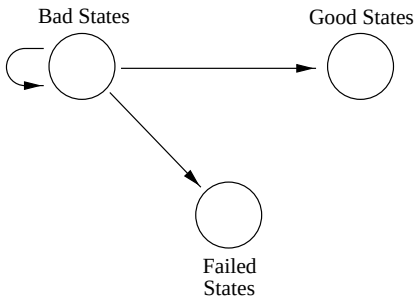

Figure 9: Abstract description of the NFA of a power plant domain.

Another limitation of strong and strong cyclic planning is the inherent pessimism of these algorithms. Strong cyclic planning will always prefer to return a strong plan if it exists, even though a strong cyclic plan may exist with a shorter, best case plan length. Consider for example the domain described in Figure 10. The strong cyclic algorithm would return a strong

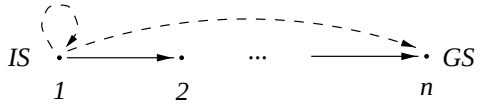

Figure 10: The NFA of a domain with two actions (drawn as solid and dashed arrows) showing the price in best case plan length when preferring strong solutions. IS is the initial state while GS is the goal state.

plan only considering solid actions. This plan would have a best and worst case length of n. But a strong cyclic plan considering both solid and dashed actions also exists and could be preferable because the best case length of 1 of the cyclic solution may have a much higher probability than the infinite worst case length.

By adding a unrecoverable dead-end for the dashed action and making solid actions non-deterministic (see Figure 11) strong cyclic planning now returns a strong cyclic plan considering only solid actions. But we might still be interested in a plan with best case performance even though the goal is not guaranteed to be achieved.

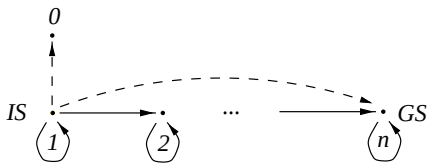

Figure 11: The NFA of a domain with two actions (drawn as solid and dashed arrows) showing the price in best case plan length when preferring strong cyclic solutions. IS is the initial state while GS is the goal state.

6.4 Optimistic Planning

The analysis in the previous section shows that there exist domains and planning problems for which we may want to use a fully relaxed algorithm, that always includes the best case plan and returns a solution even if it includes dead-ends which cannot be guaranteed to be avoided. An algorithm similar to the strong planning algorithm, that adds an ordinary preimage in each iteration has these properties. Because state-action pairs that can have transitions to unrecoverable dead-ends are added to the universal plan, this algorithm is called *optimistic planning*. The algorithm is shown in Figure 12.

The optimistic planning algorithm is incomplete because it does not necessarily return a strong solution if it exists. Intuitively, optimistic planning only guarantees that there exists some effect of a plan action leading to the goal, where strong planning guarantees that all effects of plan actions lead to the goal.

The purpose of optimistic planning is not to substitute strong or strong cyclic planning. In domains where strong or strong cyclic plans can be found and goal achievement has the highest priority these algorithms should be used. On the other hand, in domains where goal achievement cannot be guaranteed or the shortest plan should be included in the universal plan, optimistic planning might be the better choice.

Consider again, as an example, the robot-baby domain described in Section 4. For this problem an optimistic solution makes the robot try to lift the block as long as it is working. A similar optimistic plan is generated in the power plant domain. For all bad states the optimistic plan recommend an action that brings the plant to a good state in one step. This continues as long as the environment keeps the plant in a bad state. Because no strategy can be used to avoid the environment from bringing the block lifting robot and power plant to an unrecoverable dead-end, the optimistic solution is

```
procedure OptimisticPlanning(Init, Goal)
        VisitedStates := Goal
        UniversalPlan := ∅
        while (Init ⊄ VisitedStates)
                StateActions := Preimage(VisitedStates)
                PrunedStateActions := Prune(StateActions, VisitedStates)
                if StateActions ≠ ∅ then
                        UniversalPlan := UniversalPlan ∪ PrunedStateActions
                        VisitedStates := VisitedStates ∪ StatesOf(PrunedStateActions)
                else
                        return "No optimistic plan exists"
return UniversalPlan
```

Figure 12: The optimistic planning algorithm. All sets in this algorithm are represented by their characteristic function, which is implemented as an OBDD. Preimage(*VisitedStates*) returns the set of state-action pairs U associated with the preimage of the visited states. Prune(*StateActions*, *VisitedStates*) removes the state-action pairs, where the state already is included in the set of visited states. StatesOf(*PrunedStateActions*) returns the set of states of the pruned state-action pairs.

quite sensible.

For the domains shown in Figure 10 and 11 optimistic planning would return a universal plan with two state-action pairs: $(1, \mathit{dotted})$ and $(n-1, \mathit{solid})$. For both domains this is a universal plan with the shortest best case length. Compared to the strong cyclic solution the price in the first domain is that the plan may have an infinite length, while the price in the second domain is that a dead-end may be reached.

7 The UMOP Planner

UMOP is written in C and C++. It uses the programs `lex` and `yacc` for generating a scanner and parser for *NADL* descriptions and includes the BUDDY OBDD package (Lind-Nielsen, 1999) for handling OBDD operations. The UMOP program is actually comprised of three subprograms: a program for classical deterministic and non-deterministic planning (see Figure 13), a program for extracting sequential plans from universal plans (see Figure 14) and finally a program for analyzing universal plans and other OBDDs produced by UMOP (see Figure 15). A detailed description of these subprograms is given in the following two sections.

7.1 Planning

A flow diagram of a planning session is shown in Figure 13. A planning session begins with a parsing of the *NADL* description. The parsing is done by a parser generated by `lex` and `yacc` from the script files `mnp.l` and `mnp.y` (see Appendix D.2 and D.3). The domain is represented internally by the structure defined in `domain.h` (see Appendix D.4.5). The parser is written in C, because `lex` and `yacc` generates C files. The rest of the code is written in C++ partly to exploit the C++ interface of the BUDDY package. The corresponding C++ domain representation is defined in `domain.hpp` (see Appendix D.4.6). The domain representation is analyzed and translated to a domain definition structure defined in `fsm.hpp` (see Appendix D.4.7). The domain definition is a large structure containing all the domain information needed by any function. Its most important structures are:

1. A mapping of each domain variable and action to the set of OBDD variables representing it (e.g. used for translating formulas to OBDD's).

2. A 2D array of pointers to action descriptions. number).

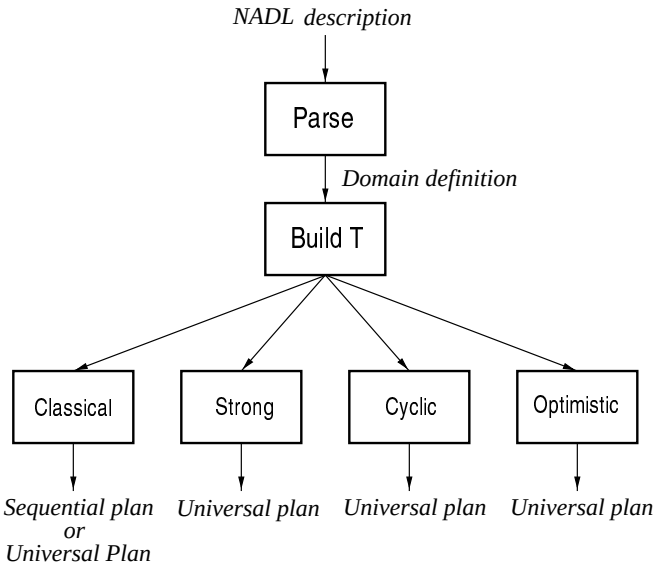

Figure 13: Flow diagram of a UMOP planning session.

3. A disjoint segmentation of the domain variables defining the largest number of transition relation partitions.

4. An OBDD representation of action variables and current and next state variables.

The domain definition is generated by `mkdomdef` in `fsm.cc` (see Appendix D.5.5).

Next, the transition relation is calculated. By default the largest number of partitions is chosen. In this version of UMOP, there is no automatic union of small partitions. This must be done by hand for example by joining some of the sets in the variable segmentation in the domain definition.

The partitioned transition relation is computed by the function `T` in `fsm.cc`. `T` uses the functions `A`, `IV` and `AC`. `A` computes the action relation (A) for a single action, `IV` computes the frame relation (F) for a partition and `AC` computes the interference relation (I). The translation of a formula to an OBDD is done by `formula2bdd`. It computes the OBDD by a recursive descent of the formula expression. Numerical values are represented by an array of OBDDs with a size corresponding to the number of digits in their binary representation. Thus, a domain variable represented by the OBDD variables $x_1, x_2, \cdots, x_n$ have the array representation $[\tilde{x}_1, \tilde{x}_2, \cdots, \tilde{x}_n]$, where

$\tilde{x}_1, \tilde{x}_2, \cdots, \tilde{x}_n$ denote the OBDD encoding of the variables.

Because arithmetic operations can be performed directly on the OBDD arrays, the arithmetic functions are implemented as the usual logical algorithms for addition, subtraction etc.. Due to the time limitations of the project the current version of UMOP only includes addition and subtraction. An explicit algorithm for subtraction has been avoided by rearranging the arithmetic expression (see `numberprop::minus2plus` in `domain.cc`, Appendix D.5.4).

The partitioned transition relation returned by `T` is used for generating plans by the planning algorithms. UMOP has four planning algorithms implemented:

1. Classical deterministic planning (described below).
2. Strong planning.
3. Strong cyclic planning.
4. Optimistic planning.

The algorithms are implemented in `plan.cc` (see Appendix D.5.7). Only the deterministic algorithm will be described here, as the non-deterministic algorithms already have been presented in previous sections.

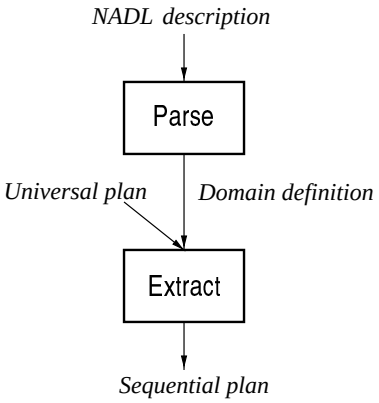

Figure 14: Flow diagram of a UMOP plan extraction session.

The backward search of the deterministic planning algorithm is similar to the optimistic planning algorithm. A sequential plan is generated from

the universal plan by choosing an initial state and iteratively adding an action from the universal plan until a goal state is reached. The output of the deterministic planning algorithm can either be the extracted sequential plan or the universal plan, it is based on. The universal plan can be used to extract other sequential plans. The flow diagram of such a plan extraction session is shown in Figure 14. As depicted in the figure, the *NADL* domain description must be read again to produce the domain definition structure. The deterministic planning algorithm has been implemented to verify the performance of UMOP compared to other classical planners. It has not been intended to be a fast OBDD based classical planning algorithm like the algorithms developed by Di Manzo et al. (1998).

7.2 Analyzing

Universal plans, and any other OBDDs representing an expression on some domain variables, can be analyzed in an interactive session. To analyze an OBDD, the domain definition, of the domain it origins from, is first computed. The domain definition is used to generate OBDDs representing formulas for constraining the input OBDD (see Figure 15). The analyzer is implemented in

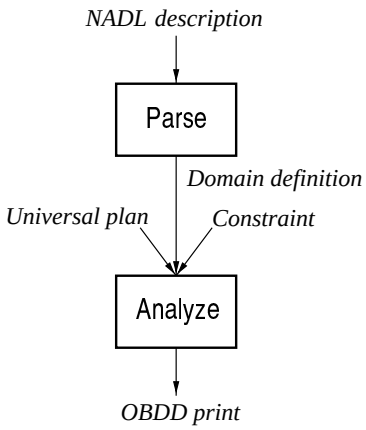

Figure 15: Flow diagram of a UMOP analyze session.

`analyse.cc` (see Appendix D.5.1). Commands can be executed for counting the number of nodes in the OBDD, writing the constrained OBDD to a file or constraining the OBDD by a formula that may be read from a file.

8 Results

In the following four subsections the results obtained with the UMOP planning system are presented in 10 different domains ranging from deterministic and single-agent with no environment actions to non-deterministic and multi-agent with complex environment actions[6]. For each domain, a domain example and the raw results of the experiments are shown in the appendix.

8.1 Deterministic Domains

A number of experiments have been carried out in deterministic domains from the AIPS'98 planning competition in order to compare UMOPs performance with some of the AIPS'98 planners. Subsequently, an obstacle domain is presented to demonstrate that a deterministic, universal plan can comprise a large number of classical sequential plans. Finally, a deterministic approach for handling non-determinism in the power plant domain is shown.

8.1.1 AIPS'98 Competition Domains

Five planning systems BLACKBOX, IPP, STAN, HSP and SGP participated in the competition. Only the first four of these planners competed in the three examined domains. BLACKBOX is based on SATPLAN (Kautz & Selman, 1996), while IPP and STAN are graphplan-based planners (Blum & Furst, 1995). HSP uses a heuristic search approach based on a preprocessing of the domain. The AIPS'98 planners were run on 233/400 MHz[7] Pentium PCs with 128 MB RAM equipped with Linux.

The Gripper Domain. The gripper domain (see Appendix C.1) consists of two rooms A and B, a robot with a left and right gripper and a number of balls that can be moved by the robot. The task is to move all the balls from room A to room B, with the robot initially in room A. The state variables of the *NADL* encoding of the domain are the position of the

[6] All experiments were carried out on a 350 MHz Pentium PC with 1 GB RAM running Red Hat Linux 4.2.

[7] Unfortunately no exact record has been kept on the machines and there is some disagreement about their clock frequency. According to Drew McDermott, who chaired the competition, they were 233 MHz Pentiums, but Derek Long (STAN) believes, they were at least 400 MHz Pentiums, as STAN performed worse on a 300 MHz Pentium than in the competition.

Problem	umop Part.			umop Mono.		stan		hsp		ipp		blackbox	
1	20	11	1	20	11	46	11	2007	13	50	15	113	11
2	150	17	1	130	17	1075	17	2150	21	380	23	7820	17
3	710	23	1	740	23	54693	23	2485	31	3270	31	-	-
4	1490	29	2	2230	29	3038381	29	3060	37	26680	39	-	-
5	3600	35	2	6040	35	-	-	3320	47	226460	47	-	-
6	7260	41	2	11840	41	-	-	3779	53	-	-	-	-
7	13750	47	2	24380	47	-	-	4797	63	-	-	-	-
8	23840	53	2	38400	53	-	-	5565	71	-	-	-	-
9	36220	59	3	68750	59	-	-	6675	79	-	-	-	-
10	56200	65	3	95140	65	-	-	7583	85	-	-	-	-
11	84930	71	3	145770	71	-	-	9060	93	-	-	-	-
12	127870	77	3	216110	77	-	-	10617	101	-	-	-	-
13	197170	83	3	315150	83	-	-	12499	109	-	-	-	-
14	290620	89	4	474560	89	-	-	15050	119	-	-	-	-
15	411720	95	4	668920	95	-	-	16886	125	-	-	-	-
16	549610	101	4	976690	101	-	-	20084	135	-	-	-	-
17	746920	107	4	-	-	-	-	23613	143	-	-	-	-
18	971420	113	4	-	-	-	-	26973	151	-	-	-	-
19	1361580	119	5	-	-	-	-	29851	157	-	-	-	-
20	1838110	125	5	-	-	-	-	33210	165	-	-	-	-

Table 1: Gripper domain results. Column one and two show the execution time in milliseconds and the plan length. UMOP Part. and UMOP Mono. show the execution time for UMOP using a partitioned and a monolithic transition relation respectively. For UMOP with partitioned transition relation the third column shows the number of partitions. (- -) means the planner failed. Only results for executions using less than 128 MB are shown for UMOP.

robot and the position of the balls. The position of the robot is either 0 (room A) or 1 (room B), while the position of a ball can be 0 (room A), 1 (room B), 2 (in left gripper) or 3 (in right gripper). For the AIPS'98 gripper problems the number of plan steps in an optimal plan grows linear with the problem number. Problem 1 contains 4 balls, and the number of balls grow with two for each problem. The result of the experiment is shown in Table 1 together with the results of the planners in the AIPS'98 competition. A graphical representation of the execution time in the table is shown in Figure 16. UMOP generates shortest plans due to its parallel breadth first search algorithm. As depicted in Figure 16, it avoids the exponential growth of the execution time that characterizes all of the competition planners except HSP. When using a partitioned transition relation UMOP is the only planner capable of generating optimal plans for all the problems. For this domain the transition relation of an *NADL* description can be divided into $n + 1$ basic partitions, where n is the number of balls. As discussed in Section 5, the optimal number of partitions is not necessarily the largest number of partitions. For the results in Table 1 each partition equaled a conjunction of 10 basic partitions. Compared to the monolithic transition relation representation the results obtained with the partitioned transition

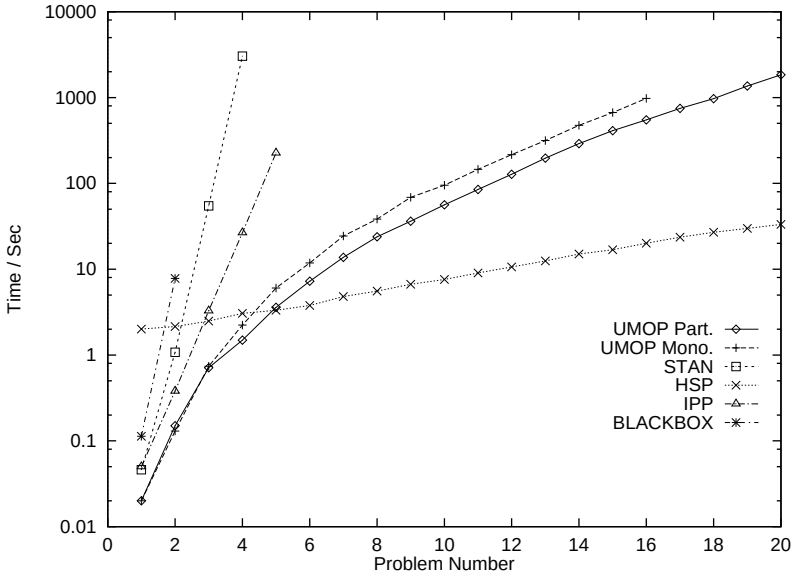

Figure 16: Execution time for UMOP and the AIPS'98 competition planners for the gripper domain problems. UMOP Part. and UMOP Mono. show the execution time for UMOP using a partitioned and a monolithic transition relation respectively.

relation was significantly better on the larger problems. The memory usage for problem 20 with a partitioned transition relation was 87 MB, while it, for the monolithic transition relation, exceeded the limit of 128 MB at problem 17.

The Movie Domain. In the movie domain (see Appendix C.2) the task is to get chips, dip, pop, cheese and crackers, rewind a movie and set the counter to zero. The only interference between the subgoals is that the movie must be rewound, before the counter can be set to zero. The problems in the movie domain only differs by the number of objects of each type of food. The number of objects increases linear from 5 for problem 1 to 34 for problem 30.

The *NADL* description of the movie domain represents each type of food as a numerical state variable with a range equal to the number of objects of that type of food. Table 2 shows the execution time for UMOP and the competition planners for the movie domain problems. In this experiment

Problem	UMOP		STAN		HSP		IPP		BLACKBOX	
1	14	7	19	7	2121	7	10	7	11	7
2	12	7	18	7	2104	7	10	7	12	7
3	14	7	19	7	2144	7	10	7	14	7
4	4	7	20	7	2188	7	10	7	16	7
5	14	7	21	7	2208	7	10	7	18	7
6	16	7	22	7	2617	7	10	7	20	7
7	14	7	22	7	2316	7	20	7	22	7
8	12	7	23	7	2315	7	20	7	24	7
9	14	7	25	7	2357	7	-	-	26	7
10	14	7	26	7	2511	7	10	7	29	7
11	14	7	27	7	2427	7	30	7	30	7
12	4	7	28	7	2456	7	30	7	32	7
13	16	7	29	7	3070	7	20	7	36	7
14	14	7	31	7	2573	7	30	7	35	7
15	16	7	32	7	2577	7	30	7	38	7
16	14	7	34	7	2699	7	10	7	39	7
17	16	7	35	7	2645	7	30	7	41	7
18	14	7	37	7	2686	7	10	7	43	7
19	16	7	39	7	2727	7	30	7	45	7
20	12	7	40	7	2787	7	20	7	47	7
21	16	7	42	7	2834	7	20	7	49	7
22	14	7	45	7	2834	7	20	7	51	7
23	16	7	48	7	2866	7	20	7	53	7
24	14	7	50	7	3341	7	20	7	55	7
25	16	7	52	7	2997	7	30	7	57	7
26	16	7	54	7	3013	7	40	7	58	7
27	16	7	57	7	3253	7	50	7	60	7
28	4	7	62	7	3049	7	40	7	63	7
29	18	7	64	7	3384	7	50	7	64	7
30	16	7	67	7	3127	7	40	7	66	7

Table 2: Movie domain results. For each planner column one and two show the run time in milliseconds and the plan length. (- -) means the planner failed. UMOP used far less than 128 MB for any problem in this domain.

and the remaining experiments UMOP used its default partitioning of the transition relation. For every problem all the planners find the optimal plan. Like the competition planners UMOP has a low computation time, but it is the only planner not showing any increase in computation time even though, the size of the state space of its encoding increases from 2^{24} to 2^{39}.

The Logistics Domain. The logistics domain (see Appendix C.3) consists of cities, trucks, airplanes and packages. The task is to move packages to specific locations. Problems differ by the number of packages, cities, airplanes and trucks. The logistics domain is hard and only problem 1,2,5,7 and 11 of the 30 problems were solved by any planner in the AIPS'98 competition (see Table 3). The *NADL* description of the logistics domain uses numerical state variables to represent locations of packages, where trucks and airplanes are treated as special locations. Even though, the state space for the small problems is moderate, UMOP fails to solve any of the problems in the domain. It succeeds to generate the transition relation but fails to finish the preimage calculations. The reason for this might be a bad representation or variable ordering. It might also be that no compact OBDD representation exists for this domain in the same way, that no compact OBDD representation exists for the integer multiplier (Bryant, 1986). More research is needed to decide this. Appendix C.3 shows results from a number smaller logistics domains, where UMOP succeeds to find a solution.

Problem	STAN		HSP		IPP		BLACKBOX	
1	767	27	79682	43	900	26	2062	27
2	4319	32	97114	44	-	-	6436	32
5	364932	29	144413	26	2400	24	-	-
7	-	-	788914	112	-	-	-	-
11	12806	34	86195	30	6940	33	6544	32

Table 3: Logistics domain results. For each planner column one and two show the run time in milliseconds and the plan length. (- -) means the planner was unable to find a solution.

8.1.2 The Obstacle Domain

The obstacle domain (see Appendix C.4) has been constructed to demonstrate the generality of universal plans. It consists of a 8×4 grid world, n obstacles and a robot agent. The position of the obstacles are not defined. The goal position of the robot is the upper right corner of the grid, and the

task for the robot is to move from any position in the grid, different from the goal position, to the goal position. Because the initial location of obstacles is unknown, the universal plan must take any possible position of obstacles into account, which gives $2^{5(n+1)} - 2^{5n}$ initial states. For a specific initial state a sequential plan can be generated from the universal plan. Thus, $2^{5(n+1)} - 2^{5n}$ sequential plans are comprised in one universal plan. Note that a universal plan with n obstacles includes any universal plan with 1 to n obstacles, as obstacles can be placed at the same location. Note moreover, that the universal plans never covers all initial states, because obstacles can be placed at the goal position, and obstacles can block the way for the agent.

A universal plan for an obstacle domain with 5 obstacles was generated with UMOP in 420 seconds and contained 488296 OBDD nodes (13.3 MB). Sequential plans were extracted from the universal plan for a specific position of the obstacles, for which 16 step plans existed. Figure 17 shows

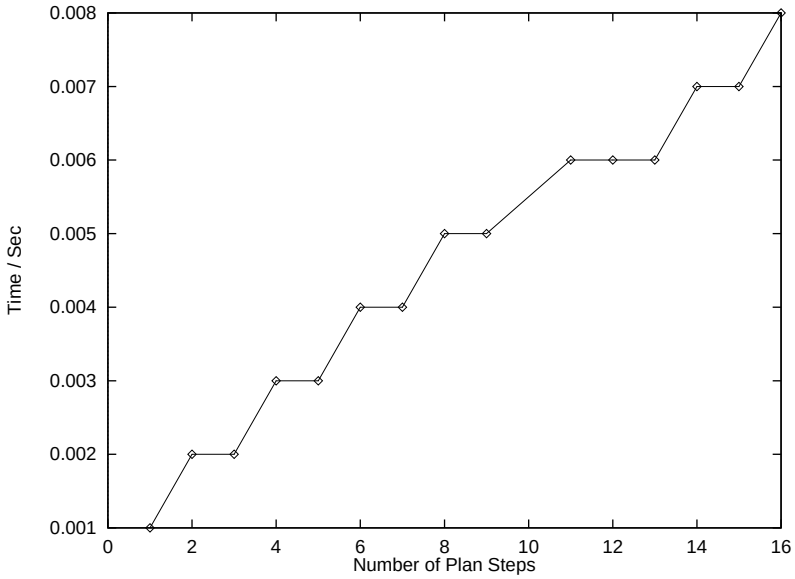

Figure 17: Time for extracting sequential plans from a universal plan for the obstacle domain with 5 obstacles.

the extraction time of sequential plans for an increasing number of steps in the plan. Even though the OBDD representing the universal plan is large, the extraction is very fast and only grows linear with the plan length. The set of actions associated with a state s in a universal plan p is extracted by computing the conjunction of the OBDD representation of s and p. As

described in Section 3, this operation has an upper bound complexity of $O(|s||p|)$. For the universal plan in the obstacle domain with five obstacles this computation was fast (less than one millisecond) and would allow an executing agent to meet low reaction time constraints, but in the general case, it depends on the structure of the universal plan and might be more time consuming.

8.1.3 Deterministic Power Plant Domain

The deterministic power plant domain shows how a deterministic approach can be used to handle non-determinism caused by the environment. The power plant domain consists of reactors, heat exchangers, turbines and valves. The task is to execute the right control actions in order to bring the plant from some bad state, where the plant is unsafe or not working properly, to some good state, where the plant is safe and working optimally. An agent is associated to each controllable unit such that any adjustment can be made at each time step. The actions of the environment consist of breaking units (like heat exchangers, valves and turbines). In the deterministic approach the actions of the environment are ignored, such that control actions always have their expected outcome.

Consider a universal plan covering as many bad states as possible. The plan is obviously sensible, if we assume, no environment actions can happen. But for the power plant domain, the plan is also sensible, even if the environment executes actions simultaneously with the control actions. To realize this, consider the plant to be in some bad state. Because the actions of the environment are unpredictable, the only sensible thing to do is to execute a control action that brings the plant nearer to a good state. If some environment actions do happen, the rationale still applies to the resulting bad state (or no rationale applies if a failed state is reached). Thus, the effect of simultaneous environment actions can be ignored.

The studied power plant domain is shown in figure 18. The *NADL* description is shown in Appendix C.5. The domain consists of one reactor R, four heat exchangers H1, H2, H3 and H4 and four turbines T1, T2, T3 and T4. The heat exchangers can fail and leak radio active water from the internal water loop to the external water loop. If this happens the blocking valve ($a1$, $a2$, $a3$ or $a4$) of the heat exchanger must be closed. Unfortunately, these valves can fail too in which case, the valves $m2$, $m3$ or $m1$ are used. Similarly, if turbines break, they must be shut down by closing one of the valves $b1$, $b2$, $b3$ or $b4$, or $m4$, $m5$ and $m1$, when the turbine valves are failed. These safety requirements are expressed in the goal condition by:

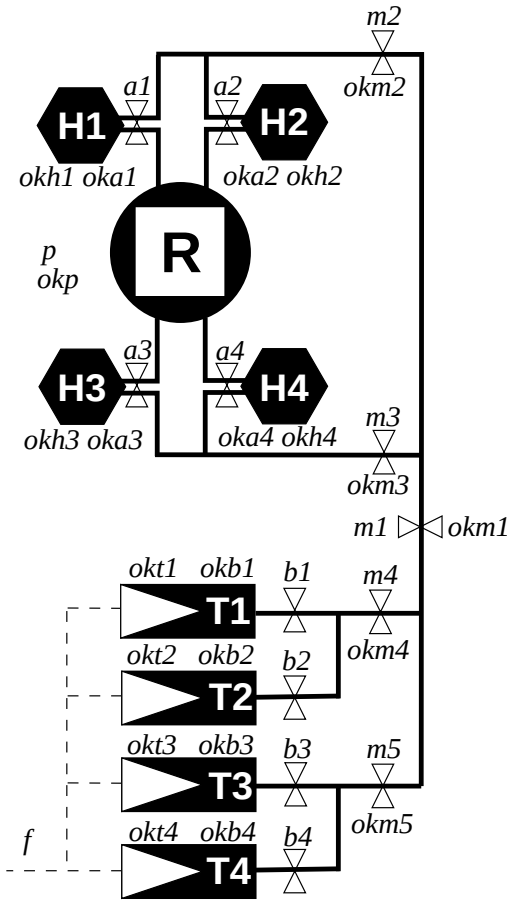

Figure 18: The deterministic power plant domain. The reactor R is surrounded by the four heat exchangers H1, H2, H3 and H4. The heat exchangers produce high pressure damp to the four electricity generating turbines T1, T2, T3 and T4. The heat exchangers can fail and leak radio active water from the internal water loop to the external water loop. If this happens, the blocking valve ($a1$, $a2$, $a3$ or $a4$) of the heat exchanger must be closed. Unfortunately, these valves can fail too in which case, the valves $m2$, $m3$ or $m1$ are used. Similarly, if turbines break, they must be shut down by closing one of the valves $b1$, $b2$, $b3$ or $b4$, or $m4$, $m5$ and $m1$, when the turbine valves are failed. The energy production of the plant p can either be 0,1,2,3 or 4 energy units pr. time unit. The production must be adjusted to fit the demand f. A heat exchanger can only transfer energy to one turbine, and one turbine can only produce one energy unit of electricity.

```
% environment security
(~okh1 => (~a1 \/  a1/\~oka1/\~m2  \/
                   a1/\~oka1/\m2/\~okm2/\~m1)) /\
(~okh2 => (~a2 \/  a2/\~oka2/\~m2  \/
                   a2/\~oka2/\m2/\~okm2/\~m1)) /\
(~okh3 => (~a3 \/  a3/\~oka3/\~m3  \/
                   a3/\~oka3/\m3/\~okm3/\~m1)) /\
(~okh4 => (~a4 \/  a4/\~oka4/\~m3  \/
                   a4/\~oka4/\m3/\~okm3/\~m1)) /\

% turbine security
(~okt1 => (~b1 \/  b1/\~okb1/\~m4  \/
    b1/\~okb1/\m4/\~okm4/\~m1 )) /\
(~okt2 => (~b2 \/  b2/\~okb2/\~m4  \/
    b2/\~okb2/\m4/\~okm4/\~m1 )) /\
(~okt3 => (~b3 \/  b3/\~okb3/\~m5  \/
    b3/\~okb3/\m5/\~okm5/\~m1 )) /\
(~okt4 => (~b4 \/  b4/\~okb4/\~m5  \/
    b4/\~okb4/\m5/\~okm5/\~m1 ))
```

The energy production can be controlled by p to be either 0,1,2,3 or 4 energy units per time unit. If p fails the energy production stays at its last value. An energy production of n requires that at least n heat exchangers and turbines are working and that the necessary valves are open. Moreover, at least a demand of f should be present. The production requirements are stated in the goal condition by:

```
% produce 0 units if system blocked or no demand
(p = 0 => f = 0 \/ ~m1 \/ ~m2/\~m3 \/ ~m2/\~a3/\~a4 \/
          ~m3/\~a1/\~a2 \/ ~m4/\~m5 \/ ~m4/\~b1/\~b2 \/
           ~m5/\~b3/\~b4) /\

% produce 1 unit if:
% A: can get 1 energy unit through and have more than 0 unit demand
(p = 1 => f > 0 /\ m1 /\
          (m2/\(a1\/a2) \/ m3/\(a3\/a4)) /\
          (m4/\(b1\/b2) \/ m5/\(b3\/b4))) /\

% B: haven't got an ok and more than 1 unit demand
(p = 1 => ~(f > 1 /\ m1 /\
        ( m2/\a1/\a2 \/ m3/\a3/\a4 \/
          m2/\(a1\/a2)/\m3/\(a3\/a4) ) /\
```

```
        ( m4/\b1/\b2 \/ m5/\b3/\b4 \/
          m4/\(b1\/b2)/\m5/\(b3\/b4)))) /\

...

% produce 4 energy units if:
% can get 4 energy units through and have a 4 unit demand
( p = 4 => f = 4/\a1/\a2/\m2/\a3/\a4/\m3/\m1/\
                 b1/\b2/\m4/\b3/\b4/\m5)
```

Finally valves of working subsystems should be open. This is expressed by:

```
(okh1 => a1) /\
(okh2 => a2) /\
(okh3 => a3) /\
(okh4 => a4) /\
(okh1 /\ okh2 => m2) /\
(okh3 /\ okh4 => m3) /\

(okt1 => b1) /\
(okt2 => b2) /\
(okt3 => b3) /\
(okt4 => b4) /\
(okt1 /\ okt2 => m4) /\
(okt3 /\ okt4 => m5) /\

( ~( (~okh1/\a1 \/ ~okh2/\a2)/\m2 \/
     (~okh3/\a3 \/ ~okh4/\a4)/\m3 \/
     (~okt1/\b1 \/ ~okt2/\b2)/\m4 \/
     (~okt3/\b3 \/ ~okt4/\b4)/\m5) => m1)
```

The initial states of the planning problem is simply defined to be states, that are not goal states.

The domain shown in Appendix C.5 was solved by UMOP in 14.0 seconds. The universal plan does not cover the initial state, as some states are unrecoverably failed (e.g. p is failed at level 4 and T1 is failed).

13 other versions of the domain with 1 to 13 agents were defined to examine the complexity of the multi-agent decomposition. The results are shown in the graph in Figure 19. As depicted the execution time is lower for domains with a large number of agents. One reason for this could be, that fewer preimage calculations are necessary for domains with many agents.

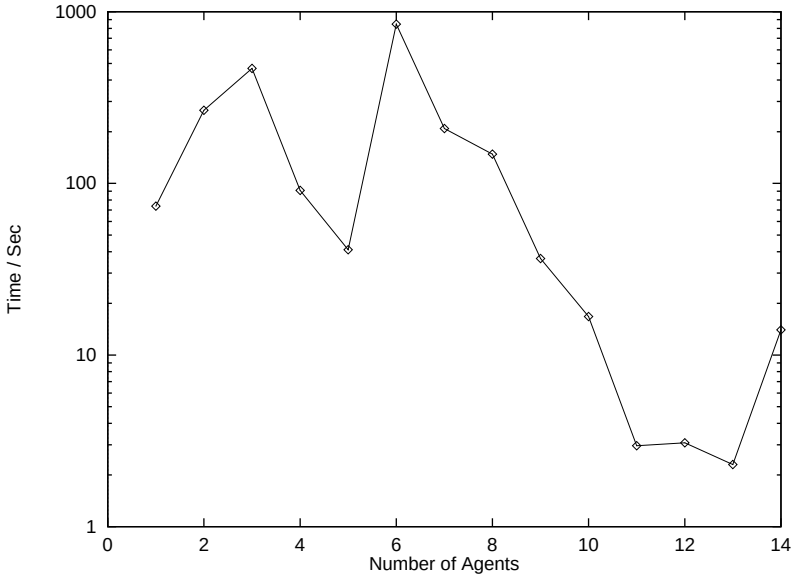

Figure 19: Execution time of UMOP for power plant domains with 1 to 14 agents.

But the result is not trivial, as the number of OBDD variables, used to represent the joint actions, grows linear with the number of agents. In the soccer domain, presented in Section 8.2.3, it turns out that a version of the doamin with only a single system and environment agent is executed much faster than the multi-agent version.

8.2 Non-Deterministic Domains

In this section UMOP's performance is first tested for some of the non-deterministic domains solved by MBP (Cimatti et al., 1998a, 1998b). Next, the power plant domain briefly described in Section 6.2 is presented and finally, results from a multi-agent soccer domain are shown.

8.2.1 Domains Tested by MBP

One of the domains introduced by Cimatti et al. (1998a, 1998b) is a non-deterministic transportation domain. The domain consists of a set of locations and a set of actions like drive-truck, drive-train and fly to move between the locations. Non-determinism is caused by non-deterministic actions (e.g.,

a truck may use the last fuel) and environmental changes (e.g., fog at airports). The two domain examples from Cimatti et al. (1998a, 1998b) (see Appendix C.6 and C.7) were defined for strong and strong cyclic planning in *NADL* and executed by UMOP using strong and strong cyclic planning. Both examples were solved in less than 0.05 seconds. Similar results were obtained with MBP. Cimatti et al. (1998a) also study a general version of the hunter and prey domain (Koenig & Simmons, 1995) and a beam walk domain. Their generalization of the hunter and prey domain is not described in detail. Thus, an *NADL* implementation of this domain has not been possible.

The problem in the beam walk domain is for an agent to walk from one end of a beam to the other without falling down. If the agent falls, it has to walk back to the end of the beam and try again. The finite state machine of the domain is shown in Figure 20. The edges denotes the outcome of a walk action. When the agent is on the beam, the walk action can either move it one step further on the beam or make it fall to a location under the beam. A generator program for *NADL* descriptions of beam walk domains

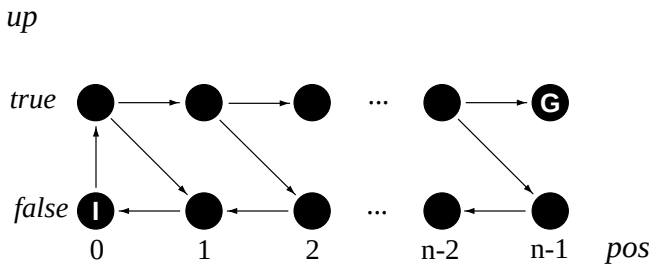

Figure 20: The beam walk domain. The *NADL* encoding of the beam walk domain has one propositional state variable *up*, which is true if the agent is on the beam and a numerical state variable *pos*, which denotes the position of the agent either on the beam or on the ground. "I" and "G" are the intial state and goal state repspectively.

were implemented and used to produce domains with 4 to 4096 positions (see Appendix C.8). Because the domain only contains two state variables, UMOP cannot exploit a partitioned transition relation for this domain, but have to use a monolithic representation. As shown in Figure 21 the execution time of UMOP was a little smaller than MBP. When discounting that a 75% faster machine was used for UMOP, MBP performs better in this domain. This is probably not due to an inefficient representation, as UMOP exploits

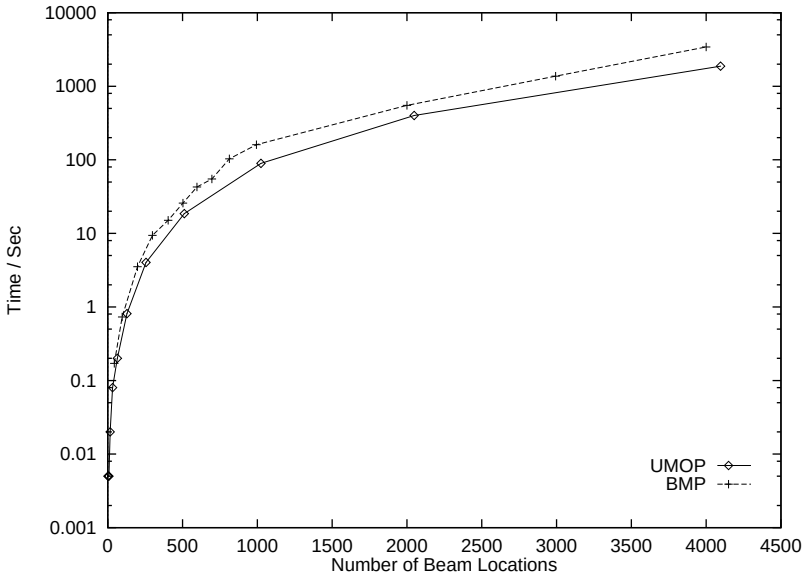

Figure 21: Execution time of UMOP and MBP in the beam walk domain. The MBP data has been extracted with some loss of accuracy from (Cimatti et al., 1998a)

the regularity of the domain in the same way as MBP. A more reasonable explanation is that UMOP uses a less efficient implementation of the strong cyclic planning algorithm.

A detailed comparison of UMOP and MBP is very interesting, as the two systems represent planning problems in a quite different way. Currently MBP is unable to use a partitioned transition relation representation, but it is still an open question if UMOP is able to solve larger problems than MBP due to this feature.

8.2.2 The Non-deterministic Power Plant Domain

The non-deterministic power plant domain (see Appendix C.9) demonstrates a multi-agent domain with an environment model and further exemplifies optimistic planning. It consists of reactors, heat exchangers, turbines and valves. A domain example is shown in Figure 22. In the power plant domain

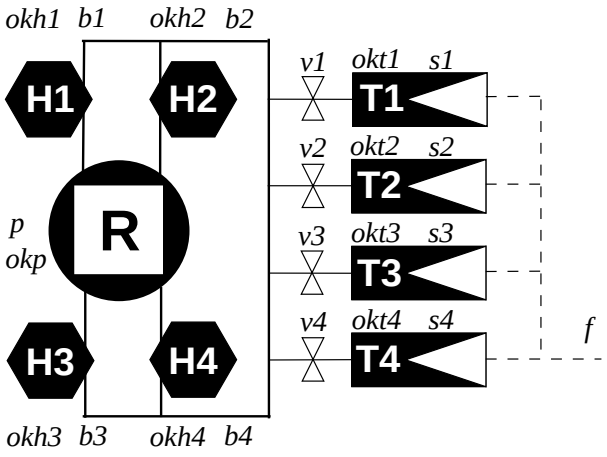

Figure 22: A power plant domain example. The reactor R is surrounded by the four heat exchangers H1, H2, H3 and H4. The heat exchangers produces high pressure damp to the four electricity generating turbines T1, T2, T3 and T4. A failed heat exchanger must be closed by a block action b. For a failed turbine the stop action s must be carried out. The energy production of the reactor is p and can be controlled to fit the demand f. Each turbine can be closed of by a valve v.

each controllable unit is associated with an agent such that all control actions can be executed simultaneously. The environment consists of a single agent

that at any time can fail a number of heat exchanges and turbines and ensures that already failed units remain failed. A failed heat exchanger leaks radioactive water from the internal to the external water loop and must be closed by a block action b. For a failed turbine the stop action s must be carried out. The energy production from the reactor can be controlled by p to fit the demand f, but the reactor will always produce two energy units. To transport the energy from the reactor away from the plant at least one heat exchanger and one turbine must be working. Otherwise the plant is in an unrecoverable failed state, where the reactor will overheat.

The state space of the power plant can be divided into three disjoint sets: good, bad and failed states. In the good states the power plant satisfies its safety and activity requirements. In the example the safety requirements ensures that energy can be transported away from the plant and failed units are shut down:

```
% energy can be transported away from the plant
(okh1 \/ okh2 \/ okh3 \/ okh4) /\
(okt1 \/ okt2 \/ okt3 \/ okt4) /\

% heat exchangers blocked if failed
(~okh1 => b1) /\
(~okh2 => b2) /\
(~okh3 => b3) /\
(~okh4 => b4) /\

% turbines stopped if failed
(~okt1 => s1) /\
(~okt2 => s2) /\
(~okt3 => s3) /\
(~okt4 => s4)
```

The activity requirements state that the energy production equals the demand and that all valves to working turbines are open:

```
% power production equals demand
p = f /\

% turbine valve is open if turbine is ok
(okt1 => v1) /\
```

```
(okt2 => v2) /\
(okt3 => v3) /\
(okt4 => v4)
```

In a bad state the plant does not satisfy the safety and activity requirements, but on the other hand is not unrecoverably failed. Finally, in a failed state all heat exchangers or turbines are failed.

The universal planning task is to generate a universal plan to get from any bad state to some good state without ending in a failed state. Assuming that no units fail during execution, it is obvious that only one joint action is needed. Unfortunately, the environment can fail any number of units during execution, thus, as described in Section 6.2, for any bad state the resulting joint action may loop back to a bad state or cause the plant to end in a failed state. (see Figure 9). For this reason no strong or strong cyclic solution exists to the problem.

An optimistic solution simply ignores that joint actions can loop back to a bad state or lead to a failed state and finds a solution to the problem after one preimage calculation. Intuitively, the optimistic plan assumes that no units will fail during execution and always chooses joint actions that lead directly from a bad state to a good state. The optimistic plan is an optimal control strategy, because it always chooses the shortest way to a good state and no other strategy exists that can avoid looping back to a bad state or end in a failed state.

The size of the state space of the above power plant domain is 2^{24}. An optimistic solution was generated by UMOP in 0.92 seconds and contained 37619 OBDD nodes. As an example, a joint action was extracted from the plan for a bad state where H3 and H4 were failed and energy demand f was 2 energy units, while the energy production p was only 1 unit. The extraction time was 0.013 seconds and as expected the set of joint actions included a single joint action changing $b3$ and $b4$ to true and setting p to 2.

8.2.3 The Soccer Domain

The purpose of the soccer domain (see Appendix C.10) is to demonstrate a multi-agent domain with a more elaborate environment model than the power plant domain. It consists of two teams of players that can move in a grid world and pass a ball to each other. At each time step a player either moves in one of the four major directions or passes the ball to another team

player. The task is to generate a universal plan for one of the teams that can be applied, whenever the team possesses the ball in order to score a goal.

A simple *NADL* description of the soccer domain models the team possessing the ball as system agents that can move and pass the ball independent of each other. Thus, a player possessing the ball can always pass to any other team player. The opponent team is modelled as a set of environment agents that can move in the four major directions but have no actions for handling the ball. The goal of the universal plan is to have a player with the ball in front of the opponent goal without having any opponents in the goal area.

Clearly, it is impossible to generate a strong plan that covers all possible initial states. But a strong plan covering as many initial states as possible is useful, because it defines all the "scoring" states of the game and further provides a plan for scoring the goal no matter what actions, the opponent players choose.

An *NADL* generator was implemented for soccer domains with different field sizes and numbers of agents. The Multi-Agent graph in Figure 23 shows UMOP's execution time using the strong planning algorithm in soccer domains with 64 locations and one to six players on each team. The execution time seems to grow exponential with the number of players. This is not surprising as not only the state space but also the number of joint actions grow exponential with the number of agents. To investigate the complexity introduced by joint actions, a version of the soccer domain with only a single system and environment agent was constructed. The Single-Agent graph in Figure 23 shows a dramatic decrease in computation time. Its is not obvious though, that a parallelization of domain actions increases the computational load as this normally also reduces the number of preimage calculations, because a larger number of states is reached in each iteration. Indeed, in the deterministic version of the power plant domain the execution time was found to decrease, when more agents were added (see Figure 19). Again the time for extracting a joint action from the generated universal plan was measured. For the multi-agent version of the five player soccer domain the two joint actions achieving the goal shown in Figure 24 were extracted from the universal plan in less than 0.001 seconds.

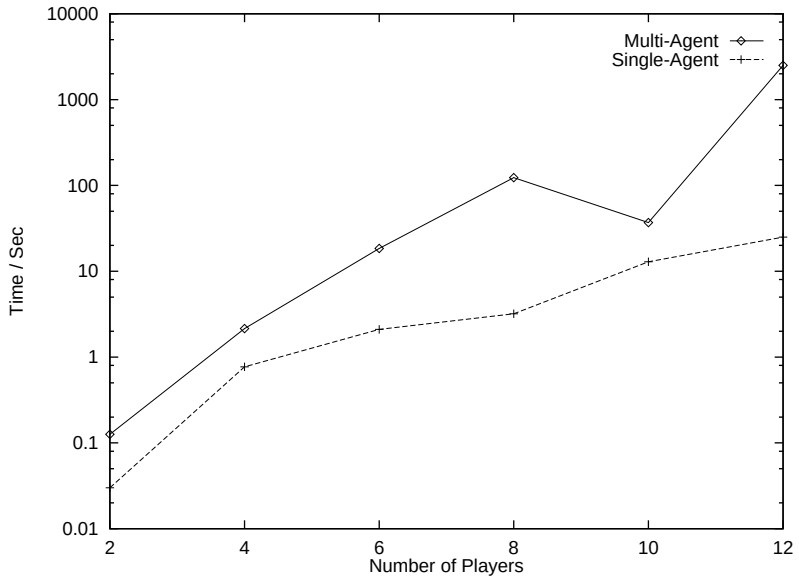

Figure 23: Execution time of UMOP for generating strong universal plans in soccer domains with one to six players on each team. For the multi-agent experiment each player was associated with an agent, while only a single system and environment agent was used in the single-agent experiment.

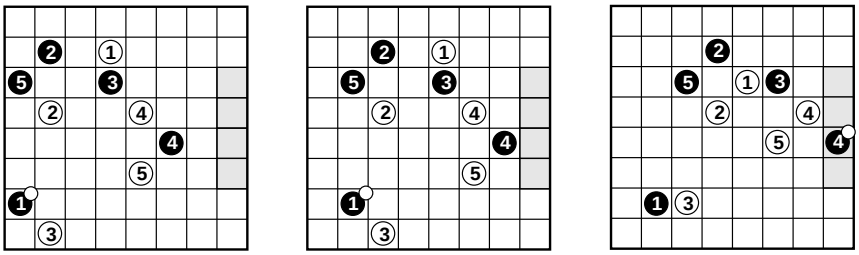

Figure 24: Plan execution sequence. The three states show a hypothetical attack based on a universal plan. The state (a) is a "scoring" state, because the attackers (black) can extract a nonempty set of joint actions from the universal plan. Choosing some joint actions from the plan the attackers can enter the goal area (shaded) with the ball within two time steps (state (b) and (c)) no matter what actions, the opponent players choose.

9 Previous Work

Recurrent approaches performing planning in parallel with execution have been widely used in non-deterministic robotic domains (e.g., Georgeff & Lansky, 1987; Gat, 1992; Wilkins et al., 1994; Haigh & Veloso, 1998). A group of planners suitable for recurrent planning is action selectors based on heuristic search (Koenig & Simmons, 1995; Bonet et al., 1997). The min-max LRTA* algorithm (Koenig & Simmons, 1995) can generate suboptimal plans in non-deterministic domains through a search and execution iteration. The search is based on a heuristic goal distance function, which must be provided for a specific problem. The ASP algorithm (Bonet et al., 1997) uses a similar approach and further defines a heuristic function for STRIPS-like (Fikes & Nilsson, 1971) action representations. In contrast to min-max LRTA*, ASP does not assume a non-deterministic environment, but is robust to non-determinism caused by action perturbations (i.e., that another action than the planned action is chosen with some probability).

In general recurrent approaches are incomplete, because acting on an incomplete plan can make the goal unachievable. Precursor approaches perform all decision making prior to execution and thus may be able to generate complete plans by taking all possible non-deterministic changes of the environment into account.

The precursor approaches include conditional (Etzioni et al., 1992; Peot & Smith, 1992), probabilistic (Drummond & Bresina, 1990; Dean et al., 1995) and universal planning (Schoppers, 1987; Cimatti et al., 1998a, 1998b; Kabanza et al., 1997). The CNLP (Peot & Smith, 1992) partial ordered, conditional planner handles non-determinism by constructing a conditional plan that accounts for each possible situation or contingency that could arise. At execution time it is determined which part of the plan to execute by performing sensing actions that are included in the plan to test for the appropriate conditions.

Probabilistic planners try to maximize the probability of goal satisfaction, given conditional actions with probabilistic effects. Drummond and Bresina (1990) represent plans as a set of Situated Control Rules (SCRs) (Drummond, 1989) mapping situations to actions. The planning algorithm begins by adding SCRs corresponding to the most probable execution path that achieves the goal. It then continues adding SCRs for less probable paths, and may end with a complete plan taking all possible paths into account.

Universal plans differs from conditional and probabilistic plans by spec-

ifying appropriate actions for every possible state of the domain[8]. Like conditional and probabilistic plans universal plans require the world to be accessible in order to execute the universal plan.

Universal planning was introduced by Schoppers (1987) who used decision trees to represent plans. Recent approaches include Kabanza et al. (1997) and Cimatti et al. (1998a, 1998b). Kabanza et al. (1997) represents universal plans as a set of Situated Control Rules (Drummond, 1989). Their algorithm incrementally adds SCRs to a final plan in a way similar to Drummond and Bresina (1990). The goal is a formula in temporal logic that must hold on any valid sequence of actions.

Reinforcement Learning (RL) (Kaebling et al., 1996) can also be regarded as a kind of universal planning. In RL the goal is represented by a reward function in a Markov Decision Process (MDP) model of the domain. In the precursor version of RL the MDP is assumed to be known and a control policy maximizing the expected reward is found prior to execution. The policy can either be represented explicitly in a table or implicitly by a function (e.g., a neural network). Because RL is a probabilistic approach, its domain representation is more complex than the domain representation used by a non-deterministic planner. Thus, we may expect non-deterministic planners to be able to handle domains with a larger state space than RL. On the other hand, RL may produce policies with a higher quality, than a universal plan generated by a non-deterministic planner.

All previous approaches to universal planning, except Cimatti et al. (1998a, 1998b), use an explicit representation of the universal plan (e.g., SCRs). Thus, in the general case exponential growth of the plan size with the number of propositions defining a domain state must be expected, as argued by Ginsberg (1989).

The compact and implicit representation of universal plans obtained with OBDDs do not necessarily grow exponentially for regular domains as shown by Cimatti et al. (1998a). Further, the OBDD-based representation of the NFA of a non-deterministic domain enables the application of efficient search algorithms from model checking, capable of handling very large state spaces.

[8]The plan solving an *NADL* problem will only be universal if the initial states equals all the states in the domain.

10 Conclusion and Future Work

In this thesis a new OBDD-based planning system called UMOP for planning in non-deterministic, multi-agent domains has been presented. An expressive domain description language called *NADL* has been developed and an efficient OBDD representation of its NFA semantics has been described. Previous planning algorithms for OBDD-based planning have been analyzed and the understanding of when these planning algorithms are appropriate has been deepened. Finally, a planning algorithm called optimistic planning for finding sensible solutions in some domains where no strong or strong cyclic solution exists has been proposed. The results obtained with UMOP are encouraging, as UMOP has a good performance compared to some of the fastest classical planners known today.

The work has raised a number of questions that would be interesting to address in the future. The most exciting of these is how well UMOP's encoding of planning problems scales compared to the encoding used by MBP. Currently MBP's encoding does not support a partitioned representation of the transition relation, but the encoding may have other properties that, despite the monolithic representation, makes it a better choice than UMOP's encoding. On the other hand, the two systems may also have an equal performance when both are using a monolithic representation (as in the beam walk example), which should give UMOP an advantage in domains where a partitioning of the transition relation can be defined. A joint work with Marco Roveri has been planned in the Fall of 1999 to address these issues.

Another interesting question is to investigate which kind of planning domains are suitable for OBDD-based planning. It was surprising that the logistics domain turned out to be so hard for UMOP. A thorough study of this domain may be the key for defining new approaches and might bring important new knowledge about the strengths and limitations of OBDD-based planning.

The current definition of *NADL* is powerful but should be extended to enable modelling of constructive synergetic effects as described in Section 4. Also, more experiments comparing multi-agent and single-agent domains should be carried out to investigate the complexity of *NADL*'s representation of concurrent actions.

As argued by Bacchus and Kabanza (1996), domain knowledge must be used by a planning system in order to scale up to real world problems. They show how the search tree of a forward chaining planner can be efficiently

pruned by stating the goal as formula in temporal logic on the sequence of actions leading to the goal. In this way the goal can include knowledge about the domain (e.g., that towers in the blocks world must be built from bottom to top). A similar approach for reducing the complexity of OBDD-based planning is obvious, especially because techniques for testing temporal formulas already have been developed in model checking.

Other future challenges includes introducing abstraction in OBDD-based planning and defining specialized planning algorithms for multi-agent domains (e.g., algorithms using the least number of agents for solving a problem).

References

Bacchus, F., & Kabanza, F. (1996). Using temporal logic to control search in a forward chaining planner. In Ghallab, M., & Milani, A. (Eds.), *New directions in AI planning*, pp. 141–153. ISO Press.

Baral, C., & Gelfond, M. (1997). Reasoning about effects of concurrent actions. *The Journal of Logic Programming*, 85–117.

Blum, A., & Furst, M. L. (1995). Fast planning through planning graph analysis. In *Proceedings of the 14'th International Conference on Artificial Intelligence (IJCAI-95)*, pp. 1636–1642. Morgan Kaufmann.

Bonet, B., Loerincs, G., & Geffner, H. (1997). A robust and fast action selection mechanism for planning. In *Proceedings of the 14'th National Conference on Artificial Intelligence (AAAI'97)*, pp. 714–719. AAAI Press / The MIT Press.

Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, *8*, 677–691.

Burch, J., Clarke, E., & Long, D. (1991). Symbolic model checking with partitioned transition relations. In *International Conference on Very Large Scale Integration*, pp. 49–58. North-Holland.

Cimatti, A., Giunchiglia, E., Giunchiglia, F., & Traverso, P. (1997). Planning via model checking: A decision procedure for $\mathcal{AR}$. In *Proceedings of the 4'th European Conference on Planning (ECP'97)*, Lecture Notes in Artificial Intelligence, pp. 130–142. Springer-Verlag.

Cimatti, A., Roveri, M., & Traverso, P. (1998a). Automatic OBDD-based generation of universal plans in non-deterministic domains. In *Proceedings of the 15'th National Conference on Artificial Intelligence (AAAI'98)*, pp. 875–881. AAAI Press/The MIT Press.

Cimatti, A., Roveri, M., & Traverso, P. (1998b). Strong planning in non-deterministic domains via model checking. In *Proceedings of the 4'th International Conference on Artificial Intelligence Planning System (AIPS'98)*, pp. 36–43. AAAI Press.

Clarke, E. M., Emerson, E. A., & Sistla, A. P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifi-

cations. *ACM transactions on Programming Languages and Systems*, *8*(2), 244–263.

Currie, K., & Tate, A. (1991). O-plan: the open planning architecture. *Artificial Intelligence*, *52*, 49–86.

Dean, T., Kaelbling, L. P., Kirman, J., & Nicholson, A. (1995). Planning under time constraints in stochastic domains. *Artificial Intelligence*, *76*, 35–74.

Di Manzo, M., Giunchiglia, E., & Ruffino, S. (1998). Planning via model checking in deterministic domains: Preliminary report. In *Proceedings of the 8'th International Conference on Artificial Intelligence: Methodology, Systems and Applications (AIMSA'98)*, pp. 221–229. Springer-Verlag.

Drummond, M. (1989). Situated control rules. In *Proceedings of the 1'st International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, pp. 103–113. Morgan Kaufmann.

Drummond, M., & Bresina, J. (1990). Anytime synthetic projection: Maximizing the probability of goal satisfaction. In *Proceedings of the 8'th Conference on Artificial Intelligence*, pp. 138–144. AAAI Press / The MIT Press.

Etzioni, O., Hanks, S., Weld, D., Draper, D., Lesh, N., & Williamson, M. (1992). An approach for planning with incomplete information. In *Proceedings of the 3'rd International Conference on Principles of Knowledge Representation and Reasoning*.

Fikes, R. E., & Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, *2*, 189–208.

Gat, E. (1992). Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the 10'th National Conference on Artificial Intelligence (AAAI'92)*, pp. 809–815. MIT Press.

Gelfond, M., & Liftschitz, V. (1993). Representing action and change by logic programs. *The Journal of Logic Programming*, *17*, 301–322.

Georgeff, M. P., & Lansky, A. L. (1987). Reactive reasoning and planning. In *Proceedings of the 6'th National Conference on Artificial Intelligence (AAAI'87)*, pp. 677–682.

Ginsberg, M. L. (1989). Universal planning: An (almost) universal bad idea. *AI Magazine*, *10*(4), 40–44.

Giunchiglia, E., Kartha, G. N., & Lifschitz, Y. (1997). Representing action: Indeterminacy and ramifications. *Aritificial Intelligence*, *95*, 409–438.

Giunchiglia, E., & Lifschitz, V. (1998). An action language based on causal explanation: Preliminary report. In *Proceedings of the 15'th National Conference on Artificial Intelligence (AAAI'98)*, pp. 623–630. AAAI Press/The MIT Press.

Haigh, K. Z., & Veloso, M. M. (1998). Planning, execution and learning in a robotic agent. In *Proceedings of the 4'th International Conference on Artificial Intelligence Planning Systems (AIPS'98)*, pp. 120–127. AAAI Press.

Kabanza, F., Barbeau, M., & St-Denis, R. (1997). Planning control rules for reactive agents. *Artificial Intelligence*, *95*, 67–113.

Kaebling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, *4*, 237–285.

Kautz, H., & Selman, B. (1996). Pushing the envelope: Planning, propositional logic and stochastic search. In *Proceedings of the 13'th National Conference on Artificial Intelligence (AAAI'96)*, Vol. 2, pp. 1194–1201. AAAI Press/MIT Press.

Koenig, S., & Simmons, R. G. (1995). Real-time search in non-deterministic domains. In *Proceedings of the 14'th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pp. 1660–1667. Morgan Kaufmann.

Lever, J., & Richards, B. (1994). *Parc*plan: a planning architecture with parallel actions and constraints. In *Lecture Notes in Artificial Intelligence*, pp. 213–222. ISMIS'94, Springer-Verlag.

Lind-Nielsen, J. (1999). BuDDy - A Binary Decision Diagram Package. Tech. rep. IT-TR: 1999-028, Institute of Information Technology, Technical University of Denmark. `http://cs.it.dtu.dk/buddy`.

Lingard, A. R., & Richards, E. B. (1998). Planning parallel actions. *Artificial Intelligence*, *99*, 261–324.

McMillan, K. L. (1993). *Symbolic Model Checking*. Kluwer Academic Publ.

Penberthy, J. S., & Weld, D. S. (1992). UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the 3'rd International Conference on Principles of Knowledge Representation and Reasoning*, pp. 103–114. Morgan Kaufmann.

Peot, M., & Smith, D. (1992). Conditional nonlinear planning. In *Proceedings of the 1'st International Conference on Artificial Intelligence Planning Systems (AIPS'92)*, pp. 189–197. Morgan Kaufmann.

Ranjan, R. K., Aziz, A., Brayton, R. K., Plessier, B., & Pixley, C. (1995). Efficient BDD algorithms for FSM synthesis and verification. In *IEEE/ACM Proceedings International Workshop on Logic Synthesis*.

Schoppers, M. J. (1987). Universal plans for reactive robots in unpredictable environments. In *Proceedings of the 10'th International Joint Conference on Artificial Intelligence (IJCAI-87)*, pp. 1039–1046. Morgan Kaufmann.

Sutton, R. S., & G., B. A. (1998). *Reinforcement Learning: An Introduction*. MIT Press.

Veloso, M., Carbonell, J., Pérez, A., Borrajo, D., Fink, E., & Blythe, J. (1995). Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, *7*(1).

Weld, D. (1999). Recent advances in AI planning. *Artificial Intelligence Magazine*. (in press).

Wilkins, D. E., Myers, K. L., Lowrance, J. D., & Wesley, L. P. (1994). Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical Artificial Intelligence*, *6*, 197–227.

Appendix

Note that the source code and the result files in this appendix refer to UMOP as MNP. Note further, that the syntax of *NADL* used by the test domains is a little different from the definition in Appendix A. Thus, the list of constrained variables is prefixed with `var:` and not `con:` and numerical state variables are defined by `scalar` and not `nat`. The ASCII symbols used for operators are: `~` ($\neg$), `/\` ($\wedge$), `\/` ($\vee$), `=>` ($\Rightarrow$), `<=>` ($\Leftrightarrow$) and `->` ($\rightarrow$).

A BNF Definition of *NADL*

⟨*NADLDesc*⟩ ::= **variables** ⟨*VarDecls*⟩
system ⟨*AgtDecls*⟩
environment ⟨*AgtDecls*⟩
initially ⟨*Formula*⟩
goal ⟨*Formula*⟩

⟨*VarDecls*⟩ ::= ϵ
| ⟨*VarLst*⟩ ⟨*VarDecls*⟩

⟨*VarLst*⟩ ::= ⟨*VarType*⟩ ⟨*IdLst*⟩

⟨*VarType*⟩ ::= **bool**
| **nat(** ⟨*Number*⟩ **)**

⟨*IdLst*⟩ ::= ϵ
| ⟨*IdLst1*⟩

⟨*IdLst1*⟩ ::= ⟨*Id*⟩
| ⟨*Id*⟩ **,** ⟨*IdLst1*⟩

⟨*AgtDecls*⟩ ::= ϵ
| **agt:** ⟨*Id*⟩ ⟨*ActionDecls*⟩ ⟨*Agtdecls*⟩

⟨*ActionDecls*⟩ ::= ⟨*ActionDecl*⟩
| ⟨*ActionDecl*⟩ ⟨*ActionDecls*⟩

⟨*ActionDecl*⟩ ::= ⟨*Id*⟩
con: ⟨*IdLst*⟩
pre: ⟨*Formula*⟩
eff: ⟨*Formula*⟩

B *NADL* Includes the $\mathcal{AR}$ Family

Theorem 1 *If A is a domain description for some $\mathcal{AR}$ language A, then there exists a domain description D in the NADL language with the same semantics as A*

Proof: let $M_a = (Q, \Sigma, \delta)$ denote the NFA (see Definition 1) equal to the semantics of A as defined by Giunchiglia et al. (1997). An *NADL* domain description D with semantics equal to M_a can obviously be constructed in the following way: let D be a single-agent domain, where all fluents are encoded as numerical variables and there is an action for each element in the alphabet Σ of M_a. Consider the action a associated to input $i \in \Sigma$. Let the set of constrained state variables of a equal all the state variables in D. The precondition of a is an expression that defines the set of states having an outgoing transition for input i. The effect condition of a is a conjunction of conditional effects $P_s \Rightarrow N_s$. There is one conditional effect for each state that has an outgoing transition for input i. P_s in the conditional effect associated with state s is the characteristic expression for s and N_s is a characteristic expression for the set of next states $\delta(s, i)$. □

C UMOP Planning Domains

C.1 Gripper

C.1.1 Generator Script

```
/*****************************************************
 * File  : gripper.cc
 * Desc. : Generator program for mnp n-gripper worlds
 *         domain used in AIPS'98
 * Author: Rune M. Jensen CS, CMU, (IAU, DTU)
 * Date  : 4/5/99
 *****************************************************/

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>


int main(int argc, char **argv) {

  int ballnum,i,j;

  // check input
  if (argc != 3)
    {
      cout << "Usage: gripper <n> <domain>.mnp\n";
      exit(1);
    }

  ofstream  out(argv[2],ios::out);
  ballnum = atoi(argv[1]);

  // write head
  out << "\% File:   " << argv[2] << "\n";
  out << "\% Desc:   AIPS'98 Competition Gripper problem with" << ballnum << " balls\n";
  out << "\% Date:   99\n";
  out << "\% Author: Rune M. Jensen CS, CMU, (IAU, DTU)\n";

  // write variablesVARIABLES
  out << "\nVARIABLES\n";
  out << "  scalar(1) pr \% position of robot: 0:rooma, 1:roomb\n";
  out << "               \% position of balls: 0:rooma, 1:roomb 2:leftgripper 3:rightgripper\n";
  out << "  scalar(2) ";
  out << "p0";
  for (i=1; i<ballnum; i++)
    out << ",p" << i;
  out << "\n";
```

```
  // write actions
  out << "\nSYSTEM\n";
  out << "\n  agt: Action\n\n";

  out << "   MoveRobotA2B\n";
  out << "     var: pr\n";
  out << "     pre: pr = 0\n";
  out << "     eff: pr' = 1\n\n";

  out << "   MoveRobotB2A\n";
  out << "     var: pr\n";
  out << "     pre: pr = 1\n";
  out << "     eff: pr' = 0\n\n";

  for (i=0; i<ballnum; i++)
    {
      out << "   PickLeft" << i << "\n";
      out << "     var: p" << i << "\n";
      out << "     pre: p" << i << " = pr\n";
      for (j=0; j<ballnum; j++)
if (j != i)
  out << "          /\\ p" << j << " <> 2\n";
      out << "     eff: p" << i << "' = 2\n\n";
    }

  for (i=0; i<ballnum; i++)
    {
      out << "   PickRight" << i << "\n";
      out << "     var: p" << i << "\n";
      out << "     pre: p" << i << " = pr\n";
      for (j=0; j<ballnum; j++)
if (j != i)
  out << "          /\\ p" << j << " <> 3\n";
      out << "     eff: p" << i << "' = 3\n\n";
    }

  for (i=0; i<ballnum; i++)
    {
      out << "   DropLeft" << i << "\n";
      out << "     var: p" << i << "\n";
      out << "     pre: p" << i << " = 2\n";
      out << "     eff: p" << i << "' = pr\n\n";
    }


  for (i=0; i<ballnum; i++)
```

```
    {
      out << "   DropRight" << i << "\n";
      out << "     var: p" << i << "\n";
      out << "     pre: p" << i << " = 3\n";
      out << "     eff: p" << i << "' = pr\n\n";
    }


  out << "ENVIRONMENT\n\n";

  out << "INITIALLY\n";
  out << "  pr = 0";
  for (i=0; i<ballnum; i++)
    out << " /\\ p" << i << " = 0";
  out << "\n\n";

  out << "GOAL\n";
  out << "  p0 = 1";
  for (i=1; i<ballnum; i++)
    out << " /\\ p" << i << " = 1";
  out << "\n";
}
```

C.1.2 Domain Example

```
% File:   prob01.mnp
% Desc:   AIPS'98 Competition Gripper problem with4 balls
% Date:   99
% Author: Rune M. Jensen CS, CMU, (IAU, DTU)

VARIABLES
  scalar(1) pr % position of robot: 0:rooma, 1:roomb
               % position of balls: 0:rooma, 1:roomb 2:leftgripper 3:rightgripper
  scalar(2) p0,p1,p2,p3

SYSTEM

  agt: Action

   MoveRobotA2B
     var: pr
     pre: pr = 0
     eff: pr' = 1

   MoveRobotB2A
     var: pr
     pre: pr = 1
     eff: pr' = 0

   PickLeft0
     var: p0
     pre: p0 = pr
          /\ p1 <> 2
          /\ p2 <> 2
          /\ p3 <> 2
     eff: p0' = 2

   PickLeft1
     var: p1
     pre: p1 = pr
          /\ p0 <> 2
          /\ p2 <> 2
          /\ p3 <> 2
     eff: p1' = 2

   PickLeft2
     var: p2
     pre: p2 = pr
          /\ p0 <> 2
          /\ p1 <> 2
          /\ p3 <> 2
```

```
  eff: p2' = 2

PickLeft3
  var: p3
  pre: p3 = pr
       /\ p0 <> 2
       /\ p1 <> 2
       /\ p2 <> 2
  eff: p3' = 2

PickRight0
  var: p0
  pre: p0 = pr
       /\ p1 <> 3
       /\ p2 <> 3
       /\ p3 <> 3
  eff: p0' = 3

PickRight1
  var: p1
  pre: p1 = pr
       /\ p0 <> 3
       /\ p2 <> 3
       /\ p3 <> 3
  eff: p1' = 3

PickRight2
  var: p2
  pre: p2 = pr
       /\ p0 <> 3
       /\ p1 <> 3
       /\ p3 <> 3
  eff: p2' = 3

PickRight3
  var: p3
  pre: p3 = pr
       /\ p0 <> 3
       /\ p1 <> 3
       /\ p2 <> 3
  eff: p3' = 3

DropLeft0
  var: p0
  pre: p0 = 2
  eff: p0' = pr
```

```
   DropLeft1
     var: p1
     pre: p1 = 2
     eff: p1' = pr

   DropLeft2
     var: p2
     pre: p2 = 2
     eff: p2' = pr

   DropLeft3
     var: p3
     pre: p3 = 2
     eff: p3' = pr

   DropRight0
     var: p0
     pre: p0 = 3
     eff: p0' = pr

   DropRight1
     var: p1
     pre: p1 = 3
     eff: p1' = pr

   DropRight2
     var: p2
     pre: p2 = 3
     eff: p2' = pr

   DropRight3
     var: p3
     pre: p3 = 3
     eff: p3' = pr

ENVIRONMENT

INITIALLY
  pr = 0 /\ p0 = 0 /\ p1 = 0 /\ p2 = 0 /\ p3 = 0

GOAL
  p0 = 1 /\ p1 = 1 /\ p2 = 1 /\ p3 = 1
```

C.1.3 Result File

```
Result of comparison between Monolithic and Partitioned
transition relation in the gripper2 domain

Rune M Jensen
5/16/99.

Machine:
Humuhumu, 350 MHz Pentium, 1GB base memory

OS:
Linux

Experiment1: Partitioned transition relation
--------------------------------------------

humuhumu was run with
/afs/cs/project/prodigy-3/runej/HUMMNP/domains/gripper2/probxx.mnp

mnp was called:
 time ../../src/mnp -plan1 XXXXX probXX.mnp probXX.sat

The initial nodecount was chosen such that no dynamic reordering
happend during planning.

The domain by default has the same number of partitions as number of balls
in the problem + 1. This number was reduced to get a more optimal size
of partitions by collapsing groups of 10 together.

DATA:

problem number, time in msec, intial nodes ,  plan length, number of partitions
1 20 20000 11 1
2 150 40000 17 1
3 710 50000 23 1
4 1490 100000 29 2
5 3600 150000 35 2
6 7260 250000 41 2
7 13750 350000 47 2
8 23840 500000 53 2
9 36220 700000 59 3
10 56200 1000000 65 3
11 84930 1200000 71 3
12 127870 1400000 77 3
13 197170 1600000 83 3
14 290620 1800000 89 4
15 411720 2200000 95 4
```

```
16 549610 2400000 101 4
17 746920 2900000 107 4
18 971420 3500000 113 4
19 1361580 3900000 119 5
20 1838110 4400000 125 5



Experiment1: Monolithic transition relation
-------------------------------------------

humuhumu was run with
/afs/cs/project/prodigy-3/runej/HUMMNP/domains/gripper2/probxx.mnp

mnp was called:
 time ../../src/mnp -plan1 XXXXX probXX.mnp probXX.sat

The initial nodecount was chosen such that no dynamic reordering
happend during planning.


DATA:

problem number, time in msec, intial nodes ,  plan length, number of partitions
1 20 20000 11 1
2 130 40000 17 1
3 740 50000 23 1
4 2230 100000 29 1
5 6040 150000 35 1
6 11840 350000 41 1
7 24380 450000 47 1
8 38400 750000 53 1
9 68750 950000 59 1
10 95140 1800000 65 1
11 145770 2200000 71 1
12 216110 2800000 77 1
13 315150 3200000 83 1
14 474560 3500000 89 1
15 668920 4300000 95 1
16 976690 5500000 101 1
17 -
18 -      RAM usage > 128 MB
19 -
20 -
```

C.2 Movie

C.2.1 Domain Example

```
% File: movie10.mnp
% Desc: MNP representation of AIPS'98 competition planning problem
%       domain: movie
%       problem: 10
% Date: 3/10/99
% Author: Rune M. Jensen CS, CMU, (IAU, DTU)
%




VARIABLES
  bool movie_rewound,counter_a_t_hours,counter_a_o_t_t_hours
  bool counter_at_zero,have_chips,have_dip
  bool have_pop,have_cheese,have_crackers
  scalar(4) chips,dip,pop,cheese,crackers

SYSTEM
  AGT: Action

   rewind_movie_2
     VAR: movie_rewound
     PRE: counter_a_t_hours
     EFF: movie_rewound'

   rewind_movie
     VAR: movie_rewound, counter_at_zero
     PRE: counter_a_o_t_t_hours
     EFF: movie_rewound' /\ ~counter_at_zero'

   reset_counter
     VAR: counter_at_zero
     PRE: true
     EFF: counter_at_zero'

   get_chips
     VAR: have_chips
     PRE: chips < 14
     EFF: have_chips'

   get_dip
     VAR: have_dip
     PRE: dip < 14
     EFF: have_dip'
```

```
   get_pop
     VAR: have_pop
     PRE: pop < 14
     EFF: have_pop'

   get_cheese
     VAR: have_cheese
     PRE: cheese < 14
     EFF: have_cheese'

   get_crackers
     VAR: have_crackers
     PRE: crackers < 14
     EFF: have_crackers'

ENVIRONMENT

INITIALLY
  chips < 14  /\ dip < 14 /\ pop < 14 /\
  cheese < 14 /\ crackers < 14 /\ counter_a_o_t_t_hours /\
  ~movie_rewound /\ ~counter_at_zero /\
  ~have_chips    /\ ~have_dip /\
  ~have_cheese   /\ ~have_pop /\
  ~have_crackers /\ ~counter_a_t_hours
GOAL
  movie_rewound /\ counter_at_zero /\
  have_chips    /\ have_dip /\
  have_cheese   /\ have_pop /\
  have_crackers
```

C.2.2 Result File

```
Result of movie domain

Rune M Jensen
5/18/99.

Machine:
Humuhumu, 350 MHz Pentium, 1GB base memory

OS:
Linux

Experiment
----------

humuhumu was run with
/afs/cs/project/prodigy-3/runej/HUMMNP/domains/movie/moviexx.mnp

mnp was called:
 time ../../src/mnp -plan1 XXXXX movieXX.mnp movieXX.sat

The initial nodecount was chosen such that no dynamic reordering
happend during planning.


DATA:

problem number, time in msec, plan length
1 14 7
2 12        (same for all)
3 14
4 4
5 14
6 16
7 14
8 12
9 14
10 14
11 14
12 4
13 16
14 14
15 16
16 14
17 16
18 14
19 16
```

```
20 12
21 16
22 14
23 16
24 14
25 16
26 16
27 16
28 4
29 18
30 16
```

C.3 Logistics

C.3.1 Generator Script

```
// File: gen1.cc
// Desc: File generating the log-1 domain
// Date: 3/18/99
// Author: Rune M. Jensen

#include <iostream.h>
#include <stream.h>
#include <fstream.h>

int main() {

int i,j;
ofstream out("gen1.mnp",ios::out);

// checked
 // load_truck(i,j)
 for (i=0; i<6; i++)   // package number
   for (j=0; j<6; j++) // trucknumber
     {
       out << "  load_truck_ppa" << i + 1 << "_ptr" << j + 1 << "\n";
       out << "    var: " << "ppa" << i + 1 << "\n";
       out << "    pre: " << "ppa" << i + 1 << " = ptr" << j + 1 << " + " << 2*j << "\n";
       out << "    eff: " << "ppa" << i + 1 << "' = " << 12 + j << "\n\n";
     }


// checked
 // load_airplane(i,j)
 for (i=0; i<6; i++)   // package number
   for (j=0; j<2; j++)  // airplane number
```

```
    {
      out << "  load_airplane_ppa" << i + 1 << "_pai" << j + 1 << "\n";
      out << "    var: " << "ppa" << i + 1 << "\n";
      out << "    pre: " << "ppa" << i + 1 << " = ";
      out << "pai" << j+1 << " + pai" << j+1 << " + 1\n";
      out << "    eff: " << "ppa" << i + 1 << "' = " << 18 + j << "\n\n";
    }


// checked
 // unload_truck(i,j)
 for (i=0; i<6; i++)   // package number
   for (j=0; j<6; j++) // truck number
     {
       out << "  unload_truck_ppa" << i + 1 << "_ptr" << j + 1 << "\n";
       out << "    var: " << "ppa" << i + 1 << "\n";
       out << "    pre: " << "ppa" << i + 1 << " = " << j + 12 << "\n";
       out << "    eff: " << "ppa" << i + 1 << "' = " << "ptr" << j+1;
       out << " + " << 2*j << "\n\n";
     }

// checked
 // unload_airplane(i,j)
 for (i=0; i<6; i++)   // package number
   for (j=0; j<2; j++)  // airplane number
     {
       out << "  unload_airplane_ppa" << i + 1 << "_pai" << j + 1 << "\n";
       out << "    var: " << "ppa" << i + 1 << "\n";
       out << "    pre: " << "ppa" << i + 1 << " = " << 18 + j << "\n";
       out << "    eff: " << "ppa" << i + 1 << "' = ";
       out << "pai" << j+1 << " + pai" << j+1 << " + 1\n\n";
     }


// checked
 // drive_truck(i,j)
 for (i=0; i<6; i++)   // truck number
   for (j=0; j<2; j++) // pos (0 - 1)
     {
       out << "  drive_truck_ptr" << i + 1 << "_to" << 2*i + j << "\n";
       out << "    var: " << "ptr" << i + 1 << "\n";
       out << "    pre: true\n";
       out << "    eff: " << "ptr" << i + 1 << "' = " << j << "\n\n";
     }

// checked
 // fly_airplane(i,j)
```

```
 for (i=0; i<2; i++)   // plane number
   for (j=0; j<6; j++) // pos (0 - 9)
     {
       out << "  fly_airplane_pai" << i + 1 << "_to" << 2*j + 1 << "\n";
       out << "    var: " << "pai" << i + 1 << "\n";
       out << "    pre: true\n";
       out << "    eff: " << "pai" << i + 1 << "' = " << j << "\n\n";
     }



}
```

C.3.2 Domain Example

```
% File: log-x-1.mnp
% Desc: MNP representation of AIPS'98 competition planning problem
%       domain: logistics
%       problem: 1
%       -This planning problem was solved by some planner in the competition
% Date: 3/18/99
% Author: Rune M. Jensen CS, CMU, (IAU, DTU)

VARIABLES
  scalar(1) ptr1,ptr2,ptr3,ptr4,ptr5,ptr6
  scalar(5) ppa1,ppa2,ppa3,ppa4,ppa5,ppa6
  scalar(3) pai1,pai2

SYSTEM
  agt: Action

  load_truck_ppa1_ptr1
    var: ppa1
    pre: ppa1 = ptr1 + 0
    eff: ppa1' = 12

  load_truck_ppa1_ptr2
    var: ppa1
    pre: ppa1 = ptr2 + 2
    eff: ppa1' = 13

...

  load_airplane_ppa1_pai1
    var: ppa1
    pre: ppa1 = pai1 + pai1 + 1
    eff: ppa1' = 18

  load_airplane_ppa1_pai2
    var: ppa1
    pre: ppa1 = pai2 + pai2 + 1
    eff: ppa1' = 19

...

  unload_truck_ppa1_ptr1
    var: ppa1
    pre: ppa1 = 12
    eff: ppa1' = ptr1 + 0

  unload_truck_ppa1_ptr2
```

```
    var: ppa1
    pre: ppa1 = 13
    eff: ppa1' = ptr2 + 2

...

  unload_airplane_ppa1_pai1
    var: ppa1
    pre: ppa1 = 18
    eff: ppa1' = pai1 + pai1 + 1

  unload_airplane_ppa1_pai2
    var: ppa1
    pre: ppa1 = 19
    eff: ppa1' = pai2 + pai2 + 1

...

  drive_truck_ptr1_to0
    var: ptr1
    pre: true
    eff: ptr1' = 0

  drive_truck_ptr1_to1
    var: ptr1
    pre: true
    eff: ptr1' = 1

...

  fly_airplane_pai1_to1
    var: pai1
    pre: true
    eff: pai1' = 0

  fly_airplane_pai1_to3
    var: pai1
    pre: true
    eff: pai1' = 1

...


ENVIRONMENT

INITIALLY
  ppa1= 2  /\ ppa2= 1 /\ ppa3= 0  /\ ppa4= 0 /\ ppa5=7 /\ ppa6=4 /\
```

```
  ptr1= 0 /\ ptr2= 0 /\ ptr3= 0 /\ ptr4=0 /\ ptr5=0 /\ ptr6=0 /\

  pai1= 3 /\ pai2=3

GOAL
  ppa1= 2  /\ ppa2=11 /\ ppa3=10 /\ ppa4=5 /\ ppa5=11 /\ ppa6=1
```

C.3.3 Result File

```
Logistics experiments   5/27/99 Rune M. Jensen CMU
---------------------

machine : Humuhumu, 350 MHz 1 GB RAM, Linux 4.2

call : time ../../src/mnp -plan1 XXX logXXXX.mnp logXXXX.sat



The logistics domains from the AIPS competition

/afs/cs.cmu.edu/project/prodigy-3/runej/HUMMNP/domains/logistics2

log-x-1.mnp  run more than 2 hours allocating 365 MB without result
log-x-2.mnp
log-x-3.mnp

log4-irst.mnp: logistics domain 4 for the Medic planner probably
used in irst deterministic paper
2 trucks, 4 packages, 2 airplanes, 6 cities


log4-irst.mnp

time SA count nodes allocated planlength
2.620 46203 550000 9
```

C.4 Obstacle

C.4.1 Generator Script

```
/*****************************************************
 * File  : obs2.cc
 * Desc. : Generator program for obstacle worlds
 *         with different x, y dimensions
 * Author: Rune M. Jensen CS, CMU, (IAU, DTU)
 * Date  : 4/7/99
 *****************************************************/
#include <math.h>
#include <time.h>
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>


int main(int argc, char **argv) {

  int sizex, sizey,obsnum,agtnum;
  int i,j, sizeg;

  // check input
  if (argc != 6)
    {
      cout << "Usage: obstacles <sizex> <sizey> <obsnum> <agtnum> <domain>.mnp\n";
      exit(1);
    }

  ofstream  out(argv[5],ios::out);
  sizex =    atoi(argv[1]);
  sizey =    atoi(argv[2]);
  obsnum =  atoi(argv[3]);
  agtnum =  atoi(argv[4]);

  // WRITE DOMAIN

  // write head
  out << "\% File:   " << argv[5] << "\n";
  out << "\% Desc:   Obstacle world with size X:" << sizex << " Y:" << sizey << "\n";
  out << "\%         obstaclenum " << obsnum << ", agtnum " << agtnum << "\n";
  out << "\% Date:   99\n";
  out << "\% Author: Rune M. Jensen CS, CMU, (IAU, DTU)\n\n";

  // write variables
  out << "\nVARIABLES\n";
  out << "  \% agents\n";
```

```
  for (i=0; i<agtnum; i++)
    {
      out << "  scalar(" << sizex << ") ax" << i << "\n";
      out << "  scalar(" << sizey << ") ay" << i << "\n";
    }

  out << "  \% objects\n";
  for (i=0; i<obsnum; i++)
    {
      out << "  scalar(" << sizex << ") ox" << i << "\n";
      out << "  scalar(" << sizey << ") oy" << i << "\n";
    }

  // write system agent actions
  out << "\nSYSTEM\n";

  for (i=0; i<agtnum; i++)
    {
      out << "\n  agt: a" << i << "\n\n";

      out << "   MoveA" << i << "Up\n";
      out << "     var: ay" << i << "\n";
      out << "     pre: \n";
      for (j=0; j<obsnum; j++)
if (j != obsnum - 1)
      out << "      ~(ay" << i << " + 1 = oy" << j << " /\\ ax" << i << " = ox" << j << ") /\\\n";
else
      out << "      ~(ay" << i << " + 1 = oy" << j << " /\\ ax" << i << " = ox" << j << ")\n";
      out << "     eff: ay" << i << "' = ay" << i << " + 1\n\n";


      out << "   MoveA" << i << "Down\n";
      out << "     var: ay" << i << "\n";
      out << "     pre: \n";
      for (j=0; j<obsnum; j++)
if (j != obsnum - 1)
      out << "      ~(ay" << i << " - 1 = oy" << j << " /\\ ax" << i << " = ox" << j << ") /\\\n";
else
      out << "      ~(ay" << i << " - 1 = oy" << j << " /\\ ax" << i << " = ox" << j << ")\n";
      out << "     eff: ay" << i << "' = ay" << i << " - 1\n\n";


      out << "   MoveA" << i << "Right\n";
      out << "     var: ax" << i << "\n";
      out << "     pre: \n";
      for (j=0; j<obsnum; j++)
if (j != obsnum - 1)
```

```
      out << "      ~(ax" << i << " + 1 = ox" << j << " /\\ ay" << i << " = oy" << j << ") /\\\n";
else
      out << "      ~(ax" << i << " + 1 = ox" << j << " /\\ ay" << i << " = oy" << j << ")\n";
      out << "     eff: ax" << i << "' = ax" << i << " + 1\n\n";


      out << "   MoveA" << i << "Left\n";
      out << "     var: ax" << i << "\n";
      out << "     pre: \n";
      for (j=0; j<obsnum; j++)
if (j != obsnum - 1)
      out << "      ~(ax" << i << " - 1 = ox" << j << " /\\ ay" << i << " = oy" << j << ") /\\\n";
else
      out << "      ~(ax" << i << " - 1 = ox" << j << " /\\ ay" << i << " = oy" << j << ")\n";
      out << "     eff: ax" << i << "' = ax" << i << " - 1\n\n";
    }

  out << "ENVIRONMENT\n";

  out << "INITIALLY\n";

  out << "GOAL\n  ";
}
```

C.4.2 Domain Example

```
% File:   o1.mnp
% Desc:   Obstacle world with size X:8 Y:4
%         obstaclenum 1, agtnum 1
% Date:   99
% Author: Rune M. Jensen CS, CMU, (IAU, DTU)


VARIABLES
  % agents
  scalar(3) ax0
  scalar(2) ay0
  % objects
  scalar(3) ox0
  scalar(2) oy0

SYSTEM

  agt: a0

   MoveA0Up
     var: ay0
     pre:
      ~(ay0 + 1 = oy0 /\ ax0 = ox0)
     eff: ay0' = ay0 + 1

   MoveA0Down
     var: ay0
     pre:
      ~(ay0 - 1 = oy0 /\ ax0 = ox0)
     eff: ay0' = ay0 - 1

   MoveA0Right
     var: ax0
     pre:
      ~(ax0 + 1 = ox0 /\ ay0 = oy0)
     eff: ax0' = ax0 + 1

   MoveA0Left
     var: ax0
     pre:
      ~(ax0 - 1 = ox0 /\ ay0 = oy0)
     eff: ax0' = ax0 - 1

ENVIRONMENT
INITIALLY
  ~(ax0 = 7 /\ ay0 = 3)
```

```
GOAL
  ax0 = 7 /\ ay0 = 3
```

C.4.3 Result File

```
Result of obstacle domain experiments with
0,1,2,3,4,5 obstacles

Rune M Jensen
5/16/99.

Machine:
Humuhumu, 350 MHz Pentium, 1GB base memory

OS:
Linux

Exp. 1.
-------

generation of the universal plans in obstacle world with
1 agent, n obstacles, and a 8x4 grid. Goal pos 7,3
Init: anything else.

mnp was called e.g.:
time ../../src/mnp -plan1 150000 o3.mnp o3.bdd


#obstacle     mnpfile      result file      SA size time msec
0 o0.mnp o0.bdd 8  0
1 o1.mnp o1.bdd 196 10
2 o2.mnp o2.bdd 1353 40
3 o3.mnp o3.bdd 10607 990
4 o4.mnp o4.bdd 80797 18810
5 o5.mnp o5.bdd 488296 421080


Exp. 2
------

Extracting single plans.

problem files: o5p4eXX.mnp
resultfile     o5p4exx.sat

SA file: o5.bdd


plansteps time/ msec
1 1
2 2
```

```
3 2
4 3
5 3
6 4
7 4
8 5
9 5
11 6
12 6
13 6
14 7
15 7
16 8
```

C.5 Power Plant (Deterministic)

C.5.1 Domain Example

```
% File: power14.mnp
% Desc: Nuclear Power Plant test domain to demonstrate non-deterministic
%       multi-agent planning
% Author: Rune M. Jensen CS, CMU, (IAU, DTU)
% Date: 5/20/99


VARIABLES
  scalar(3) p, f
  bool sys,okp,a1,oka1,okh1,m2,okm2,a2,oka2,okh2,a3,oka3,okh3,m3,okm3,a4,oka4,okh4,
       m1,okm1,b1,okb1,okt1,m4,okm4,b2,okb2,okt2,b3,okb3,okt3,m5,okm5,b4,okb4,okt4


SYSTEM

% valves in reactor hall

  agt: a1

   change
     var: a1,sys
     pre: oka1 /\ sys
     eff: (a1 -> ~a1', a1') /\ ~sys'

   nop
     var: sys
     pre: true
     eff: sys -> ~sys',sys'

  agt: a2

   change
     var: a2
     pre: oka2
     eff: a2 -> ~a2', a2'

   nop
     var:
     pre: true
     eff: true

  agt: a3

   change
```

```
   var: a3
   pre: oka3
   eff: a3 -> ~a3', a3'

 nop
   var:
   pre: true
   eff: true

agt: a4

 change
   var: a4
   pre: oka4
   eff: a4 -> ~a4', a4'

 nop
   var:
   pre: true
   eff: true

agt: m2

 change
   var: m2
   pre: okm2
   eff: m2 -> ~m2', m2'

 nop
   var:
   pre: true
   eff: true

agt: m3

 change
   var: m3
   pre: okm3
   eff: m3 -> ~m3', m3'

 nop
   var:
   pre: true
   eff: true


agt: p
```

```
   p0
     var: p
     pre: p < 5 /\ okp
     eff: p' = 0

   p1
     var: p
     pre: p < 5 /\ okp
     eff: p' = 1

   p2
     var: p
     pre: p < 5 /\ okp
     eff: p' = 2

   p3
     var: p
     pre: p < 5 /\ okp
     eff: p' = 3

   nop
     var:
     pre: true
     eff: true


% main valve between reactor and turbine hall

  agt: m1

   change
     var: m1
     pre: okm1
     eff: m1 -> ~m1', m1'

   nop
     var:
     pre: true
     eff: true


% valves in turbine hall

  agt: b1

   change
```

```
   var: b1
   pre: okb1
   eff: b1 -> ~b1', b1'

 nop
   var:
   pre: true
   eff: true

agt: b2

 change
   var: b2
   pre: okb2
   eff: b2 -> ~b2', b2'

 nop
   var:
   pre: true
   eff: true

agt: b3

 change
   var: b3
   pre: okb3
   eff: b3 -> ~b3', b3'

 nop
   var:
   pre: true
   eff: true

agt: b4

 change
   var: b4
   pre: okb4
   eff: b4 -> ~b4', b4'

 nop
   var:
   pre: true
   eff: true


agt: m4
```

```
   change
     var: m4
     pre: okm4
     eff: m4 -> ~m4', m4'

   nop
     var:
     pre: true
     eff: true

  agt: m5

   change
     var: m5
     pre: okm5
     eff: m5 -> ~m5', m5'

   nop
     var:
     pre: true
     eff: true

ENVIRONMENT

INITIALLY
 ~(% turbine security
  (~okt1 => (~b1 \/  b1/\~okb1/\~m4  \/
      b1/\~okb1/\m4/\~okm4/\~m1 )) /\
  (~okt2 => (~b2 \/  b2/\~okb2/\~m4  \/
      b2/\~okb2/\m4/\~okm4/\~m1 )) /\
  (~okt3 => (~b3 \/  b3/\~okb3/\~m5  \/
      b3/\~okb3/\m5/\~okm5/\~m1 )) /\
  (~okt4 => (~b4 \/  b4/\~okb4/\~m5  \/
      b4/\~okb4/\m5/\~okm5/\~m1 )) /\

  % environment security
  (~okh1 => (~a1 \/  a1/\~oka1/\~m2  \/
                     a1/\~oka1/\m2/\~okm2/\~m1)) /\
  (~okh2 => (~a2 \/  a2/\~oka2/\~m2  \/
                     a2/\~oka2/\m2/\~okm2/\~m1)) /\
  (~okh3 => (~a3 \/  a3/\~oka3/\~m3  \/
                     a3/\~oka3/\m3/\~okm3/\~m1)) /\
  (~okh4 => (~a4 \/  a4/\~oka4/\~m3  \/
                     a4/\~oka4/\m3/\~okm3/\~m1)) /\
```

```
% Production requirement

% produce 0 units if system blocked or no demand
(p = 0 => f = 0 \/ ~m1 \/ ~m2/\~m3 \/ ~m2/\~a3/\~a4 \/
          ~m3/\~a1/\~a2 \/ ~m4/\~m5 \/ ~m4/\~b1/\~b2 \/
           ~m5/\~b3/\~b4) /\

% produce 1 unit if:
% A: can get 1 energy unit through and have more than 0 unit demand
(p = 1 => f > 0 /\ m1 /\
          (m2/\(a1\/a2) \/ m3/\(a3\/a4)) /\
          (m4/\(b1\/b2) \/ m5/\(b3\/b4))) /\

% B: haven't got an ok more than 1 unit demand
(p = 1 => ~(f > 1 /\ m1 /\
        ( m2/\a1/\a2 \/ m3/\a3/\a4 \/
          m2/\(a1\/a2)/\m3/\(a3\/a4) ) /\
        ( m4/\b1/\b2 \/ m5/\b3/\b4 \/
          m4/\(b1\/b2)/\m5/\(b3\/b4)))) /\

% produce 2 units if:
% A: can get 2 energy units through and have more than 1 unit demand
(p=2 => f > 1 /\ m1 /\
        ( m2/\a1/\a2 \/ m3/\a3/\a4 \/
          m2/\(a1\/a2)/\m3/\(a3\/a4) ) /\
        ( m4/\b1/\b2 \/ m5/\b3/\b4 \/
          m4/\(b1\/b2)/\m5/\(b3\/b4))) /\

% B: haven't got an ok more than 2 unit demand
(p=2 => ~(f > 2 /\ m1 /\
          m2/\m3/\(a1/\a2/\(a3\/a4) \/ a3/\a4/\(a1\/a2)) /\
          m4/\m5/\(b1/\b2/\(b3\/b4) \/ b3/\b4/\(b1\/b2)))) /\


% produce 3 units if:
% A: can get 3 energy units through and have more than 2 unit demand
(p = 3 => f > 2 /\ m1 /\
          m2/\m3/\(a1/\a2/\(a3\/a4) \/ a3/\a4/\(a1\/a2)) /\
          m4/\m5/\(b1/\b2/\(b3\/b4) \/ b3/\b4/\(b1\/b2))) /\

% B: haven't got an ok more than 3 unit demand
(p = 3 => ~(f = 4/\a1/\a2/\m2/\a3/\a4/\m3/\m1/\
                  b1/\b2/\m4/\b3/\b4/\m5)) /\


% produce 4 energy units if:
% can get 4 energy units through and have a 4 unit demand
```

```
  ( p = 4 => f = 4/\a1/\a2/\m2/\a3/\a4/\m3/\m1/\
                  b1/\b2/\m4/\b3/\b4/\m5) /\


  % setting requirements
  p < 5 /\ f < 5 /\

  % activity requirements
  % valves should be open if their subsystems are ok
  (okh1 => a1) /\
  (okh2 => a2) /\
  (okh3 => a3) /\
  (okh4 => a4) /\
  (okh1 /\ okh2 => m2) /\
  (okh3 /\ okh4 => m3) /\

  (okt1 => b1) /\
  (okt2 => b2) /\
  (okt3 => b3) /\
  (okt4 => b4) /\
  (okt1 /\ okt2 => m4) /\
  (okt3 /\ okt4 => m5) /\

  ( ~( (~okh1/\a1 \/ ~okh2/\a2)/\m2 \/
       (~okh3/\a3 \/ ~okh4/\a4)/\m3 \/
       (~okt1/\b1 \/ ~okt2/\b2)/\m4 \/
       (~okt3/\b3 \/ ~okt4/\b4)/\m5) => m1))

GOAL

  % turbine security
  (~okt1 => (~b1 \/  b1/\~okb1/\~m4  \/
      b1/\~okb1/\m4/\~okm4/\~m1 )) /\
  (~okt2 => (~b2 \/  b2/\~okb2/\~m4  \/
      b2/\~okb2/\m4/\~okm4/\~m1 )) /\
  (~okt3 => (~b3 \/  b3/\~okb3/\~m5  \/
      b3/\~okb3/\m5/\~okm5/\~m1 )) /\
  (~okt4 => (~b4 \/  b4/\~okb4/\~m5  \/
      b4/\~okb4/\m5/\~okm5/\~m1 )) /\

  % environment security
  (~okh1 => (~a1 \/  a1/\~oka1/\~m2  \/
                     a1/\~oka1/\m2/\~okm2/\~m1)) /\
  (~okh2 => (~a2 \/  a2/\~oka2/\~m2  \/
                     a2/\~oka2/\m2/\~okm2/\~m1)) /\
  (~okh3 => (~a3 \/  a3/\~oka3/\~m3  \/
                     a3/\~oka3/\m3/\~okm3/\~m1)) /\
```

```
(~okh4 => (~a4 \/  a4/\~oka4/\~m3  \/
                   a4/\~oka4/\m3/\~okm3/\~m1)) /\


% Production requirement

% produce 0 units if system blocked or no demand
(p = 0 => f = 0 \/ ~m1 \/ ~m2/\~m3 \/ ~m2/\~a3/\~a4 \/
          ~m3/\~a1/\~a2 \/ ~m4/\~m5 \/ ~m4/\~b1/\~b2 \/
           ~m5/\~b3/\~b4) /\

% produce 1 unit if:
% A: can get 1 energy unit through and have more than 0 unit demand
(p = 1 => f > 0 /\ m1 /\
          (m2/\(a1\/a2) \/ m3/\(a3\/a4)) /\
          (m4/\(b1\/b2) \/ m5/\(b3\/b4))) /\

% B: haven't got an ok more than 1 unit demand
(p = 1 => ~(f > 1 /\ m1 /\
        ( m2/\a1/\a2 \/ m3/\a3/\a4 \/
          m2/\(a1\/a2)/\m3/\(a3\/a4) ) /\
        ( m4/\b1/\b2 \/ m5/\b3/\b4 \/
          m4/\(b1\/b2)/\m5/\(b3\/b4)))) /\

% produce 2 units if:
% A: can get 2 energy units through and have more than 1 unit demand
(p=2 => f > 1 /\ m1 /\
        ( m2/\a1/\a2 \/ m3/\a3/\a4 \/
          m2/\(a1\/a2)/\m3/\(a3\/a4) ) /\
        ( m4/\b1/\b2 \/ m5/\b3/\b4 \/
          m4/\(b1\/b2)/\m5/\(b3\/b4))) /\

% B: haven't got an ok more than 2 unit demand
(p=2 => ~(f > 2 /\ m1 /\
          m2/\m3/\(a1/\a2/\(a3\/a4) \/ a3/\a4/\(a1\/a2)) /\
          m4/\m5/\(b1/\b2/\(b3\/b4) \/ b3/\b4/\(b1\/b2)))) /\


% produce 3 units if:
% A: can get 3 energy units through and have more than 2 unit demand
(p = 3 => f > 2 /\ m1 /\
          m2/\m3/\(a1/\a2/\(a3\/a4) \/ a3/\a4/\(a1\/a2)) /\
          m4/\m5/\(b1/\b2/\(b3\/b4) \/ b3/\b4/\(b1\/b2))) /\

% B: haven't got an ok more than 3 unit demand
(p = 3 => ~(f = 4/\a1/\a2/\m2/\a3/\a4/\m3/\m1/\
                  b1/\b2/\m4/\b3/\b4/\m5)) /\
```

```
% produce 4 energy units if:
% can get 4 energy units through and have a 4 unit demand
( p = 4 => f = 4/\a1/\a2/\m2/\a3/\a4/\m3/\m1/\
                 b1/\b2/\m4/\b3/\b4/\m5) /\


% setting requirements
p < 5 /\ f < 5 /\

% activity requirements
% valves should be open if their subsystems are ok
(okh1 => a1) /\
(okh2 => a2) /\
(okh3 => a3) /\
(okh4 => a4) /\
(okh1 /\ okh2 => m2) /\
(okh3 /\ okh4 => m3) /\

(okt1 => b1) /\
(okt2 => b2) /\
(okt3 => b3) /\
(okt4 => b4) /\
(okt1 /\ okt2 => m4) /\
(okt3 /\ okt4 => m5) /\

( ~( (~okh1/\a1 \/ ~okh2/\a2)/\m2 \/
     (~okh3/\a3 \/ ~okh4/\a4)/\m3 \/
     (~okt1/\b1 \/ ~okt2/\b2)/\m4 \/
     (~okt3/\b3 \/ ~okt4/\b4)/\m5) => m1)
```

C.5.2 Result File

```
Power plant multiagent experiments Deterministic : No environment
-------------------------------------------------

Power plant domain: number of states: 2^42
doesn't change for the different problems.


Experiment 1:
-------------

Machine: humuhumu, 350 MHz, Pentium, 1 GB RAM,

Experiment purpose: examine the impact of using a different number of agents.


notes: dynamic reordering used in this experiment as the variable ordering
       turns out to be bad. ( all problems started with 5000 nodes such that
       variable reordering was necessary for all problems )
       ( in the other JAIR experiments the initial variable ordering was ok, thus
         dynamic reordering could be avoided)

files: /afs/cs/project/prodigy-3/runej/HUMMNP/domains/power2/powerXX.mnp ,
       powerXX.bdd

Optimistic planning used: does not matter: domain is deterministic.

call: time ../../src/mnp -plan4 5000 powerXX.mnp powerXX.bdd

The problem number indicate the number of agents used.

Problem# SA node count time/ msec #clusters
1   11871 73650 10
2 347982 266540
3 915766 467740
4  449624 91020
5  193824 41060
6 2423862 847090
7   759441 208810
8 558946 148000
9  192593 36460
10 67154 16750
11  31362 2960
12  23080 3080
13  10203 2300
14  31847 14020 10
```

```
Experiment 2:
-------------

Machine: humuhumu, Pentium PC, 350 Mhz, 1 GB RAM, Linux 4.2


A: Build an universal plan
--------------------------

dir: /afs/cs.cmu.edu/project/prodigy-3/runej/HUMMNP/domains/power2

in: power14.mnp
out: power14.bdd (covering as many initstates as possible)
call: time ../../src/mnp -plan4 5000 power14.mnp power14.bdd

10 partitions.

state space size: 2^42
number of joint actions: 5*2^14
number of agents: 15

Used: Dynamic reordering.

time / msec SA node count
14060 31847


B: Extract a single joint action
--------------------------------

dir: /afs/cs.cmu.edu/project/prodigy-3/runej/HUMMNP/domains/power2
in:  power15.bdd (covering as many init states as possible)
     constraint file: power.con

% constraint


% constraint

sys /\ a1 /\ a2 /\ a3 /\ a4 /\
~oka1 /\ oka2 /\ oka3 /\ oka4 /\
~okh1 /\ okh2 /\ okh3 /\ okh4 /\
m1 /\ m2 /\ m3 /\ m4 /\ m5 /\
okm1 /\ okm2 /\ okm3 /\ okm4 /\ okm5 /\

b1 /\ b2 /\ b3 /\ b4 /\
```

```
okt1 /\ okt2 /\ okt3 /\ okt4 /\
okb1 /\ okb2 /\ okb3 /\ okb4 /\
f = 1 /\ p = 4 /\ okp


call:

time ../../src/mnp -analyse 1500000 power14.mnp power14.bdd power14.sat

using constraint file: power.con

extaction time: 0 msec

extracted joint action:

power14.sat:

ACTION
a1: nop
a2: nop
a3: nop
a4: nop
m2: change
m3: nop
p: p1
m1: nop
b1: nop
b2: nop
b3: nop
b4: nop
m4: nop
m5: nop
VARIABLES
okt4: 1 *
okb4: 1 *
b4: 1 *
o
...
```

C.6 Transport (Strong Planning)

C.6.1 Domain Example

```
% File:   irst1.mnp
% Desc:   Irst transport domain in AIPS'98 paper
% Date:   5/19/99
% Author: Rune M. Jensen CS, CMU, (IAU, DTU)
% used as result in JAIR paper 1999

VARIABLES

  % {train_station = 0, Victoria_station = 1, Gatwick=2, Luton=3, air_station=4,
  %  truck_station=5, city_center=6 }
  scalar(3) pos

  % {green=1, red=0}
  scalar(1) light

  bool      fuel,fog

SYSTEM
  AGT: trans

    drive_train
      VAR: pos
      PRE: pos = 0 \/  (pos = 1 /\ light = 1)
      EFF: (pos = 0 => pos' = 1) /\
           (pos = 1 /\ light = 1 => pos' = 2)

    wait_at_light
      VAR:
      PRE: true
      EFF: (light = 1 =>  light' = 0) /\
           (light = 0 =>  light' = 1)

    drive_truck
      VAR: pos,fuel
      PRE: (pos = 5 \/ pos = 6) /\ fuel
      EFF: (pos = 5 => pos' = 6) /\
           (pos = 6 => pos' = 2)

    make_fuel
      VAR: fuel
      PRE: ~fuel
      EFF: fuel'

    fly
```

```
      VAR: pos
      PRE: pos = 4
      EFF: (~fog => pos' = 2) /\
           ( fog => pos' = 3)

    air_truck_transit
      VAR: pos
      PRE: pos = 4
      EFF: pos' = 5


ENVIRONMENT

  AGT: lftagt

    lftact
      VAR: fog,light
      PRE: true
      EFF: true

INITIALLY
  (pos = 0 \/ pos = 4) \/ ( pos = 5 /\ fuel )
GOAL
  pos = 2
```

C.6.2 Result File

```
IRST AIPS'98 train domain:


Machine humuhumu, 350 MHz Pentium, 128 MB RAM,

call: time ../../../src/mnp -plan2 5000 train.mnp train.bdd

result:
0.000u 0.020s 0:03.75 0.5% 0+0k 0+0io 109pf+0w
0.000u 0.010s 0:01.35 0.7% 0+0k 0+0io 109pf+0w
0.000u 0.010s 0:01.23 0.8% 0+0k 0+0io 109pf+0w
0.010u 0.000s 0:01.57 0.6% 0+0k 0+0io 109pf+0w
```

C.7 Transport (Strong Cyclic Planning)

C.7.1 Domain Example

```
% File: irst.mnp
% Desc: IRST test domain in AAAI 98 paper
% Author: Rune M. Jensen CS, CMU, (IAU, DTU)
% Date: 3/26/99

VARIABLES
scalar(2) pos   % 1: at_station, 2:at_light, 3:at_airport
scalar(1) light % 0: red, 1: Green

SYSTEM
  agt: sys

    drive_train
      var: pos
      pre: pos = 1 \/ (pos = 2 /\ light = 1)
      eff: (pos = 1 => pos' = 2) /\
           (pos = 2 /\ light = 1 => pos' = 3)

    wait_at_light
      var:
      pre: pos = 2
      eff: true

ENVIRONMENT
  agt: env

    light
      var: light
      pre: true
      eff: true

INITIALLY
  pos = 1

GOAL
  pos = 3
```

C.7.2 Result File

```
IRST AIPS'98 train domain:
Rune M. Jensen 5/19/99, JAIR results

Machine humuhumu, 350 MHz Pentium, 128 MB RAM,

call: time ../../../src/mnp -plan3 5000 train.mnp train.bdd

result:
0.000u 0.020s 0:01.59 1.2% 0+0k 0+0io 109pf+0w
0.000u 0.020s 0:01.24 1.6% 0+0k 0+0io 109pf+0w
0.000u 0.010s 0:01.30 0.7% 0+0k 0+0io 109pf+0w
0.000u 0.010s 0:01.35 0.7% 0+0k 0+0io 109pf+0w
0.010u 0.000s 0:01.60 0.6% 0+0k 0+0io 109pf+0w
0.000u 0.010s 0:03.66 0.2% 0+0k 0+0io 109pf+0w
```

C.8 Beam Walk

C.8.1 Generator Script

```
/*****************************************************
 * File  : beam2.cc
 * Desc. : Generator program for beam walks
 * Author: Rune M. Jensen CS, CMU
 * Date  : 5/19/99
 *****************************************************/
#include <math.h>
#include <time.h>
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>


int main(int argc, char **argv) {

  int logsize, size;

  // check input
  if (argc != 3)
    {
      cout << "Usage: beam <logsize> <domain>.mnp\n";
      exit(1);
    }

  ofstream  out(argv[2],ios::out);
  logsize =    atoi(argv[1]);
  size = int(pow(2,logsize));


  // WRITE DOMAIN

  // write head
  out << "\% File:   " << argv[2] << "\n";
  out << "\% Desc:   Beam2 walk with size " << 2*size << "\n";
  out << "\% Date:   99\n";
  out << "\% Author: Rune M. Jensen CS, CMU \n\n";

  // write variables
  out << "\nVARIABLES\n";
  out << "  \% obstacles and agents\n";
  out << "  scalar(" << logsize << ") pos";
  out << "  bool up\n";
```

```
  // write system agent actions
  out << "\nSYSTEM\n\n";
  out << "  agt:sys\n\n";

      out << "   Walk\n";
      out << "     var: pos, up\n";
      out << "     pre: ~(up /\\ pos = " << size - 1 << ")\n";
      out << "     eff: (pos = 0 /\\ ~up -> up' /\\ pos' = 0,\n";
      out << "          (up -> pos' = pos + 1,\n";
      out << "          (pos' = pos - 1 /\\ ~up')))\n\n";

  out << "ENVIRONMENT\n";

  out << "INITIALLY\n";
  out << "  pos = 0 /\\ ~up\n\n";
  out << "GOAL\n  ";
  out << "  pos = " << size - 1  << " /\\ up\n";
}
```

C.8.2 Domain Example

```
% File:   2beam4.mnp
% Desc:   Beam2 walk with size 4
% Date:   99
% Author: Rune M. Jensen CS, CMU


VARIABLES
  % obstacles and agents
  scalar(1) pos  bool up

SYSTEM

  agt:sys

   Walk
     var: pos, up
     pre: ~(up /\ pos = 1)
     eff: (pos = 0 /\ ~up -> up' /\ pos' = 0,
          (up -> pos' = pos + 1,
          (pos' = pos - 1 /\ ~up')))

ENVIRONMENT
INITIALLY
  pos = 0 /\ ~up

GOAL
    pos = 1 /\ up
```

C.8.3 Result File

```
IRST AIPS'98 beam walk domain:
Rune M. Jensen 5/19/99, JAIR results

Machine humuhumu, 350 MHz Pentium, 128 MB RAM,

Using strong cyclic plannning:
call: time ../../../src/mnp -plan3 5000 train.mnp train.bdd

Experiment: 1 beamXX.mnp files
# locations SA size time/ msec
4 3 5
8 4 5
16 5 5
32 6 20
64 7 50
128 8 240
256 9 990
512 10 5440
1024 11 23290
2048 12 119160
4096 - -


Using strong cyclic plannning:
call: time ../../../src/mnp -plan3 5000 train.mnp train.bdd

Experiment: 2   2beamXX.mnp files
# locations SA size time/ msec
4 3 0
8 4 0
16 5 0
32 6 20
64 7 80
128 8 200
256 9 810
512 10 4020
1024 11 18590
2048 12 89690
4096 13 399800
8192            14              1874650

IRST only count every second location in their AAAI'98 paper
so they would represent our results:

# locations SA size time/ msec
2 3 0
```

```
4 4 0
8 5 0
16 6 20
32 7 80
64 8 200
128 9 810
256 10 4020
512      11 18590
1024      12 89690
2048      13 399800
4096            14              1874650
```

C.9 Power Plant (Non-Deterministic)

C.9.1 Domain Example

```
% File: power14.mnp
% Desc: Nuclear Power Plant test domain to demonstrate non-deterministic
%       multi-agent planning
% Author: Rune M. Jensen CS, CMU, (IAU, DTU)
% Date: 5/20/99
% 14 agents (13 sys)

VARIABLES
  scalar(2) p, f
  bool okh1,b1,okh2,b2,okh3,b3,okh4,b4,
       v1,v2,v3,v4,okt1,s1,okt2,s2,okt3,s3,okt4,s4

SYSTEM

% block agents

  agt: b1

   block1
     var: b1
     pre: ~okh1
     eff: b1'

   nop
     var:
     pre: true
     eff: true

  agt: b2

   block2
     var: b2
     pre: ~okh2
     eff: b2'

   nop
     var:
     pre: true
     eff: true

  agt: b3

   block3
     var: b3
```

```
    pre: ~okh3
    eff: b3'

  nop
    var:
    pre: true
    eff: true

 agt: b4

  block4
    var: b4
    pre: ~okh4
    eff: b4'

  nop
    var:
    pre: true
    eff: true

% valve agents

 agt: v1

  change1
    var: v1
    pre: true
    eff: v1 -> ~v1', v1'

  nop
    var:
    pre: true
    eff: true

 agt: v2

  change2
    var: v2
    pre: true
    eff: v2 -> ~v2', v2'

  nop
    var:
    pre: true
    eff: true

 agt: v3
```

```
   change3
     var: v3
     pre: true
     eff: v3 -> ~v3', v3'

   nop
     var:
     pre: true
     eff: true

  agt: v4

   change4
     var: v4
     pre: true
     eff: v4 -> ~v4', v4'

   nop
     var:
     pre: true
     eff: true


% turbine stop agents

  agt: s1

   stop1
     var: s1
     pre: ~okt1
     eff: s1'

   nop
     var:
     pre: true
     eff: true


  agt: s2

   stop2
     var: s2
     pre: ~okt2
     eff: s2'

   nop
```

```
    var:
    pre: true
    eff: true


  agt: s3

   stop3
    var: s3
    pre: ~okt3
    eff: s3'

   nop
    var:
    pre: true
    eff: true


  agt: s4

   stop4
    var: s4
    pre: ~okt4
    eff: s4'

   nop
    var:
    pre: true
    eff: true

% reactor control agent

  agt: p

   p0
    var: p
    pre: true
    eff: p' = 0

   p1
    var: p
    pre: true
    eff: p' = 1

   p2
    var: p
    pre: true
```

```
    eff: p' = 2

  p3
    var: p
    pre: true
    eff: p' = 3



ENVIRONMENT

  agt: env

  fail
    var: okh1,okh2,okh3,okt1,okt2,okt3,okt4
    pre: true
    eff: (~okh1 => ~okh1') /\
         (~okh2 => ~okh2') /\
         (~okh3 => ~okh3') /\
         (~okh4 => ~okh4') /\
         (~okt1 => ~okt1') /\
         (~okt2 => ~okt2') /\
         (~okt3 => ~okt3') /\
         (~okt4 => ~okt4')

INITIALLY
  % not irreversibly failed
  (okh1 \/ okh2 \/ okh3 \/ okh4) /\
  (okt1 \/ okt2 \/ okt3 \/ okt4)

GOAL

  % not irreversibly failed
  (okh1 \/ okh2 \/ okh3 \/ okh4) /\
  (okt1 \/ okt2 \/ okt3 \/ okt4) /\

  % safety requirements

  % heat exchangers blocked if failed
  (~okh1 => b1) /\
  (~okh2 => b2) /\
  (~okh3 => b3) /\
  (~okh4 => b4) /\

  % turbines stopped if failed
  (~okt1 => s1) /\
  (~okt2 => s2) /\
```

```
(~okt3 => s3) /\
(~okt4 => s4) /\

% activity requirements

% power production equals demand
p = f /\

% turbine valves are open if turbine is ok
(okt1 => v1) /\
(okt2 => v2) /\
(okt3 => v3) /\
(okt4 => v4)
```

C.9.2 Result File

```
Power plant multiagent experiments for JAIR paper : power domain 3
-------------------------------------------------


Power plant domain: number of states: 2^24


Experiment 1: Generate a universal plan with optimistic planning
-------------

Machine: humuhumu, 350 MHz, Pentium, 1 GB RAM,


files: /afs/cs/project/prodigy-3/runej/HUMMNP/domains/power3/power14.mnp

call: time ../../src/mnp -plan4 50000 power14.mnp power14.bdd


RESULT:

[runej@humuhumu]$ time ../../src/mnp -plan4 50000 power14.mnp power14.bdd
Number of clusters 16
going into preimagef
replaced old2new
Finished partition 0 node count 96
Garbage collection #1: 50000 nodes / 33361 free / 0.0s / 0.0s total
Garbage collection #2: 50000 nodes / 20021 free / 0.0s / 0.1s total
going into prunedpreimage
Garbage collection #3: 50000 nodes / 29708 free / 0.0s / 0.1s total
updating sa
SA node count: 37620
updating acc
Garbage collection #4: 50000 nodes / 6643 free / 0.0s / 0.1s total
Garbage collection #5: 100000 nodes / 56643 free / 0.1s / 0.2s total
Start reordering (98102 nodes)
Garbage collection #6: 100000 nodes / 57049 free / 0.1s / 0.2s total
End reordering (42951 nodes, 0.2 sec)
ACC node count: 8
Added preimage 1
SA covers Init
Final node count of SA: 37619
Write SA to output file "power14.bdd" (y/n): y
0.920u 0.070s 0:19.14 5.1% 0+0k 0+0io 113pf+0w
[runej@humuhumu]$


Experiment 2:
```

```
-------------

A) Extraction by analyse

Machine: humuhumu, 350 MHz, Pentium, 1 GB RAM,

Experiment purpose: extract a couple of example plans

extract file: /afs/cs/project/prodigy-3/runej/HUMMNP/domains/power2/power14.bdd
logical formula: powerexpX.con:

outputfile: powerX1.sat

EXP 1
-----


[runej@humuhumu]$ ../../src/mnp -analyse 50000 power14.mnp power14.bdd powerX1b.sat

Number of clusters 3

Analyse BDD Type "help" for command list
$ f
constraint file: powerexp1.con
Time elapsed: 13 msec
---------------------

$



powerX1.con
% constraint

~b1 /\ ~b2 /\ ~b3 /\ ~b4 /\
okh1 /\ okh2 /\ ~okh3 /\ ~okh4 /\

~s1 /\ ~s2 /\ ~s3 /\ ~s4 /\
okt1 /\ okt2 /\ okt3 /\ okt4 /\

v1 /\ v2 /\ v3 /\ v4 /\
f = 2 /\ p = 1

powerexp1.sat
ACTION
env: *
b1: nop
```

```
b2: nop
b3: block3
b4: block4
v1: nop
v2: nop
v3: nop
v4: nop
s1: nop
s2: nop
s3: nop
s4: nop
p: p2
VARIABLES
...
```

C.10 Soccer

C.10.1 Generator Script

```
/*****************************************************
 * File  : soccer.cc
 * Desc. : Generator program for JAIR paper soccer domains
 * Author: Rune M. Jensen CS, CMU
 * Date  : 5/20/99
 *****************************************************/
#include <math.h>
#include <time.h>
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>


int main(int argc, char **argv) {

  int logsizex, logsizey, sizex, sizey, ournum, oppnum, i,j,k;

  // check input
  if (argc != 6)
    {
      cout << "Usage: soccer <logsizex> <logsizey> <#our (<=8)> <#opp> <domain>.mnp\n";
      exit(1);
    }

  ofstream  out(argv[5],ios::out);
  logsizex =    atoi(argv[1]);
  sizex = int(pow(2,logsizex));
  logsizey =    atoi(argv[2]);
  sizey = int(pow(2,logsizey));
  ournum =      atoi(argv[3]);
  oppnum =      atoi(argv[4]);


  // WRITE DOMAIN

  // write head
  out << "\% File:   " << argv[5] << "\n";
  out << "\% Desc:   JAIR paper Soccer domain with logX:" << logsizex
      << " logY: " << logsizey << " #our:" << ournum << " #opp:" << oppnum << "\n";
  out << "\% Date:   99\n";
  out << "\% Author: Rune M. Jensen CS, CMU \n\n";

  // write variables
  out << "\nVARIABLES\n";
```

```
  out << "  \% ball \n";
  out << "  scalar(3) has_ball\n";

  out << "  \% our agents\n";
  for (i=0; i<ournum; i++)
    {
      out << "  scalar(" << logsizex << ") our" << i << "x\n" ;
      out << "  scalar(" << logsizey << ") our" << i << "y\n" ;
    }

  out << "  \% opp agents\n";
  for (i=0; i<oppnum; i++)
    {
      out << "  scalar(" << logsizex << ") opp" << i << "x\n" ;
      out << "  scalar(" << logsizey << ") opp" << i << "y\n" ;
    }


  // write system agent actions
  out << "\nSYSTEM\n\n";

  for (i=0; i<ournum; i++)
    {
      out << "  agt: our" << i << "\n\n";

      out << "     move_right" << i << "\n";
      out << "       var: our" << i << "x\n";
      out << "       pre: true\n";
      out << "       eff: our" << i << "x' = our" << i << "x + 1\n\n";

      out << "     move_left" << i <<"\n";
      out << "       var: our" << i << "x\n";
      out << "       pre: true\n";
      out << "       eff: our" << i << "x' = our" << i << "x - 1\n\n";

      out << "     move_up" << i << "\n";
      out << "       var: our" << i << "y\n";
      out << "       pre: true\n";
      out << "       eff: our" << i << "y' = our" << i << "y + 1\n\n";

      out << "     move_down" << i << "\n";
      out << "       var: our" << i << "y\n";
      out << "       pre: true\n";
      out << "       eff: our" << i << "y' = our" << i << "y - 1\n\n";

      for (j=0; j<ournum; j++)
if (i != j)
```

```
{
  out << "     pass" << i << "_" << j << "\n";
  out << "       var: has_ball\n";
    out << "       pre: has_ball  = " << i << "\n";
  out << "       eff: has_ball' = " << j << "\n\n";
}
  }

// write system agent actions
out << "\nENVIRONMENT\n\n";

for (i=0; i<oppnum; i++)
  {
    out << "  agt: opp" << i << "\n\n";

    out << "     move_right" << i << "\n";
    out << "       var: opp" << i << "x\n";
    out << "       pre: true\n";
    out << "       eff: opp" << i << "x' = opp" << i << "x + 1\n\n";

    out << "     move_left" << i <<"\n";
    out << "       var: opp" << i << "x\n";
    out << "       pre: true\n";
    out << "       eff: opp" << i << "x' = opp" << i << "x - 1\n\n";

    out << "     move_up" << i << "\n";
    out << "       var: opp" << i << "y\n";
    out << "       pre: true\n";
    out << "       eff: opp" << i << "y' = opp" << i << "y + 1\n\n";

    out << "     move_down" << i << "\n";
    out << "       var: opp" << i << "y\n";
    out << "       pre: true\n";
    out << "       eff: opp" << i << "y' = opp" << i << "y - 1\n\n";
  }

out << "\nINITIALLY\n";
out << "  true\n\n";
out << "GOAL\n\n";
out << "      ((our0y < " << int(0.75*sizex)  << " /\\ \n";
out << "        our0y > " << int(0.25*sizex)-1 << " /\\ \n";
out << "        our0x = " << sizey - 1 << " /\\ \n";
out << "        has_ball = 0) \n\n";


for (i = 1; i < ournum; i++)
  {
```

```
      out << " \\/ (our" << i << "y < " << int(0.75*sizex) << " /\\\n";
      out << "      our" << i << "y > " << int(0.25*sizex) << " /\\\n";
      out << "      our" << i << "x = " << sizey - 1 << " /\\\n";
      out << "      has_ball = " << i << ")\n\n";
    }

      out << "     )  /\\ \n\n";

  out << "     ~((opp0y < " << int(0.75*sizex)  << " /\\ \n";
  out << "        opp0y > " << int(0.25*sizex)-1 << " /\\ \n";
  out << "        opp0x = " << sizey - 1 << ") \n";

  for (i = 1; i < oppnum; i++)
    {
      out << " \\/ (opp" << i << "y < " << int(0.75*sizex) << " /\\\n";
      out << "      opp" << i << "y > " << int(0.25*sizex) << " /\\\n";
      out << "      opp" << i << "x = " << sizey - 1 << ") \n\n";
    }
      out << "     ) \n\n";

}
```

C.10.2 Domain Example

```
% File:   soccertest.mnp
% Desc:   JAIR paper Soccer domain with logX:3 logY: 3 #our:6 #opp:6
% Date:   99
% Author: Rune M. Jensen CS, CMU


VARIABLES
  % ball
  scalar(3) has_ball
  % our agents
  scalar(3) our0x
  scalar(3) our0y
...
  % opp agents
  scalar(3) opp0x
  scalar(3) opp0y
...
SYSTEM

  agt: our0

     move_right0
       var: our0x
       pre: true
       eff: our0x' = our0x + 1

     move_left0
       var: our0x
       pre: true
       eff: our0x' = our0x - 1

     move_up0
       var: our0y
       pre: true
       eff: our0y' = our0y + 1

     move_down0
       var: our0y
       pre: true
       eff: our0y' = our0y - 1

     pass0_1
       var: has_ball
       pre: has_ball  = 0
       eff: has_ball' = 1
```

```
    pass0_2
      var: has_ball
      pre: has_ball  = 0
      eff: has_ball' = 2

    pass0_3
      var: has_ball
      pre: has_ball  = 0
      eff: has_ball' = 3

    pass0_4
      var: has_ball
      pre: has_ball  = 0
      eff: has_ball' = 4

    pass0_5
      var: has_ball
      pre: has_ball  = 0
      eff: has_ball' = 5

  agt: our1

...

ENVIRONMENT

  agt: opp0

    move_right0
      var: opp0x
      pre: true
      eff: opp0x' = opp0x + 1

    move_left0
      var: opp0x
      pre: true
      eff: opp0x' = opp0x - 1

    move_up0
      var: opp0y
      pre: true
      eff: opp0y' = opp0y + 1

    move_down0
      var: opp0y
      pre: true
      eff: opp0y' = opp0y - 1
```

```
  agt: opp1

     move_right1
       var: opp1x
       pre: true
       eff: opp1x' = opp1x + 1

     move_left1
       var: opp1x
       pre: true
       eff: opp1x' = opp1x - 1

     move_up1
       var: opp1y
       pre: true
       eff: opp1y' = opp1y + 1

     move_down1
       var: opp1y
       pre: true
       eff: opp1y' = opp1y - 1

  agt: opp2

...

INITIALLY
  true

GOAL
   ((our0x < 6 /\
     our0x > 1 /\
     our0y = 7 /\
     has_ball = 0)

 \/ (our1x < 6 /\
      our1x > 2 /\
      our1y = 7 /\
      has_ball = 1)

...)  /\

  ~((opp0x < 6 /\
     opp0x > 1 /\
     opp0y = 7)
 \/ ... )
```

C.10.3 Result File

```
JAIR paper soccer domain 5/21/99
--------------------------------

Experiment 1: Planning time for different number of agents


Machine: humuhumu, 350 MHz, 1 GB RAM, Linux

files:
in   /afs/cs/project/prodigy-3/runej/HUMMNP/domains/soccer2/soccerX.mnp
out  /afs/cs/project/prodigy-3/runej/HUMMNP/domains/soccer2/soccerX.bdd

call:  time ../../src/mnp -plan2 50000 soccerX.mnp soccerX.bdd
(that is: strong planning was used)


Note: We used dynamic reordering, as the obdd sizes seemed to benefit from it.



epx# #our #opp SA size / nodes t/msec
1 1 1 126 50
2 2 2 2149 2040
3 3 3 18428 10920
4 4 4 123191 83250
5 5 5 36927 145360
6 6 6 2502209 5003030


Experiment 2a: Extracting a strong plan.
----------------------------------------

Base domain: Soccer doamin with 8 system agents, no environment agents

Machine: humuhumu, 350 MHz, 1 GB RAM, Linux

files:
in   /afs/cs/project/prodigy-3/runej/HUMMNP/domains/soccer2/socagt5.mnp
1   /afs/cs/project/prodigy-3/runej/HUMMNP/domains/soccer2/socer1.con
2   /afs/cs/project/prodigy-3/runej/HUMMNP/domains/soccer2/socer2.con




out:  /afs/cs/project/prodigy-3/runej/HUMMNP/domains/soccer2/soccer5_X.sat
```

```
call:  ../../src/mnp -analyse 10000000 soccer5.mnp soccer5.bdd soccer5_X.sat

$ f
constraint file: soccer1.con
Time elapsed: 1 msec
$

$ f
constraint file: soccer2.con
Time elapsed: 1 msec
$
```

Experiment 2b: Planning time for different decomposition number of agents.

Base domain: Soccer doamin with 8 system agents, no environment agents

Machine: humuhumu, 350 MHz, 1 GB RAM, Linux

files:
in /afs/cs/project/prodigy-3/runej/HUMMNP/domains/soccer2/socagtX.mnp
out /afs/cs/project/prodigy-3/runej/HUMMNP/domains/soccer2/socagtX.bdd

call: time ../../src/mnp -plan2 XXXXXX socagtX.mnp socagtX.bdd
(that is: strong planning was used)

Note: We used dynamic reordering, as the obdd sizes seemed to benefit from it.

epx# #our #opp SA size / nodes t/msec allocated nodes
1 1 0 94 25960 150000
2 2 0 82021 58880 450000
4 4 0 - - -
8 8 0 - - -

Experiment 3: Planning time for a single agent version. (1 sys and 1 env agent)

Base domain: Soccer doamin with 8 system agents, no environment agents

Machine: humuhumu, 350 MHz, 1 GB RAM, Linux

```
files:
in   /afs/cs/project/prodigy-3/runej/HUMMNP/domains/soccer2/soccerXS.mnp
out  /afs/cs/project/prodigy-3/runej/HUMMNP/domains/soccer2/socagtX.bdd

call:  time ../../src/mnp -plan2 XXXXXX  soccerXS.mnp soccerXS.bdd
(that is: strong planning was used)


epx# #our #opp SA size / nodes t/msec allocated nodes
1 1 1 126 30 50000
2 1 1 1103 770 50000
3 1 1 3528 2100 50000
4 1 1 8100 3200 50000
5 1 1 13593 12870 50000
6 1 1 23734 24950 50000
```

D UMOP Program Files

D.1 Makefile

```
# ============================================================
# Makefile for MNP files
# ============================================================

CFLAGS = -O3 -W -Wtraditional -Wmissing-prototypes -Wall

LIBDIR = /afs/cs/project/prodigy-3/runej/BUDDYHUM/lib

INCDIR = /afs/cs/project/prodigy-3/runej/BUDDYHUM/include


# --- full object list
OBJ = lex.yy.o y.tab.o domain.o fsm.o bddprint.o plan.o dissets.o analyse.o reorder.o time.o main.o
CFILES = lex.yy.c y.tab.c
CCFILES = domain.cc fsm.cc bddprint.cc plan.cc diss1ets.cc analyse.cc reorder.cc time.cc main.cc


# ------------------------------------------------------------
# Code generation
# ------------------------------------------------------------

.SUFFIXES: .cc .c

.cc.o:
g++ -I$(INCDIR) -c  $<

.c.o:
cc -c -I$(INCDIR)  $<



# ------------------------------------------------------------
# The primary targets.
# ------------------------------------------------------------

mnp: $(OBJ)
g++ $(CFLAGS) -o mnp $(OBJ) -L$(LIBDIR) -lbdd -lfl -lm
chmod u+x mnp

lex.yy.c:    mnp.l
lex mnp.l

y.tab.c y.tab.h: mnp.y
yacc -d -v mnp.y
```

```
clean:
rm -f *.o core *~
rm -f lex.yy.c y.tab.c y.tab.h
rm -f mnp

depend:
gcc -MM $(CFILES) -I$(INCDIR) > depend.inf
g++ -MM $(CCFILES) -I$(INCDIR) >> depend.inf

backup:
tar cvf ../../MNP.tar ../../HUMMNP/

###
include depend.inf
```

D.2 Lex File (mnp.l)

```
Number                  [0-9]+
Id                      [a-zA-Z_][0-9A-Za-z_]*'?

%{
/*****************************************************
 * File  : np.l
 * Desc. : lex file for NP Language
 * Author: Rune M. Jensen
 * Date  : 2/21/99, CS,CMU
 *****************************************************/

#include "y.tab.h"

int lineno = 1;

%}

%%
"%".*                   /* ignore comments, (rest of line from %) */ ;
[\t ]+                  /* ignore whitespace */ ;
\n                      { lineno++; }
"system"                { return(SYSTEM); }
"environment"           { return(ENVIRONMENT); }
"variables"             { return(VARIABLES); }
"initially"             { return(INITIALLY); }
"goal"                  { return(GOAL); }
"bool"                  { return(BOOL); }
"scalar"                { return(SCALAR); }
"agt:"                  { return(AGT); }
"pre:"                  { return(PRE); }
"eff:"                  { return(EFF); }
"var:"                  { return(VAR); }
"true"                  { return(TRUE); }
"false"                 { return(FALSE); }
"SYSTEM"                { return(SYSTEM); }
"ENVIRONMENT"           { return(ENVIRONMENT); }
"VARIABLES"             { return(VARIABLES); }
"INITIALLY"             { return(INITIALLY); }
"GOAL"                  { return(GOAL); }
"BOOL"                  { return(BOOL); }
"SCALAR"                { return(SCALAR); }
"AGT:"                  { return(AGT); }
"PRE:"                  { return(PRE); }
"EFF:"                  { return(EFF); }
"VAR:"                  { return(VAR); }
"TRUE"                  { return(TRUE); }
```

```
"FALSE"                 { return(FALSE); }
"<=>"                   { return(BIIMPL); }
"=>"                    { return(IMPL); }
"->"                    { return(ARROW); }
"/\\"                   { return(AND); }
"\\/"                   { return(OR); }
"<>"                    { return(NE); }
"~"                     { return('~'); }
"="                     { return('='); }
">"                     { return('>'); }
"<"                     { return('<'); }
"+"                     { return('+'); }
"-"                     { return('-'); }
","                     { return(','); }
"("                     { return('('); }
")"                     { return(')'); }
{Id}                    { return(ID); }
{Number}                { return(NUMBER); }
%%
```

D.3 Yacc File (mnp.y)

```
%token SYSTEM ENVIRONMENT INITIALLY GOAL
%token PRE EFF VAR AGT SCALAR ARROW TRUE FALSE
%token NE ID NUMBER VARIABLES BOOL
%token BIIMPL IMPL AND OR

%nonassoc ARROW
%left BIIMPL IMPL
%left OR
%left AND
%nonassoc '~'
%left '+' '-'

%start mnpproblem
%%
/*****************************************************
 * File  : mnp.y
 * Desc. : Yacc file for MNP language
 *         Abstract Syntax described in domain.h
 *         See also lex file mnp.l
 * Author: Rune M. Jensen, CS, CMU (IAU, DTU)
 * Date  : 3/2/99
 *****************************************************/

mnpproblem:     VARIABLES vardecls SYSTEM agtdecls ENVIRONMENT agtdecls INITIALLY formula GOAL formula
                    { mnpprob = (mnpproblem*) malloc(sizeof(mnpproblem));
                      mnpprob->vars = (varlst*) $2;
                      mnpprob->sys =  (agent*)  $4;
                      mnpprob->env =  (agent*)  $6;
                      mnpprob->init = (formula*) $8;
                      mnpprob->goal = (formula*) $10;
                     }
                ;

vardecls:
                    { $$ = (int) NULL; }
              | varlst vardecls
            { vars = (varlst*) $1;
                      vars->next = (varlst*) $2;
      $$ = (int) vars; }
                ;

varlst:         vartype idlst
            { vars = (varlst*) malloc(sizeof(varlst));
                      vars->type = (vartype*) $1;
                      vars->ids  = (idlst*) $2;
      $$ = (int) vars; }
```

```
                ;

vartype:        BOOL
             { vatype = (vartype*) malloc(sizeof(vartype));
                        vatype->type = vt_bool;
       $$ = (int) vatype; }
               | SCALAR '(' number ')'
             { vatype = (vartype*) malloc(sizeof(vartype));
                        vatype->type = vt_scalar;
       vatype->range = (int) $3;
       $$ = (int) vatype; }
                ;

idlst:
             { $$ = (int) NULL; }
               | idlst1
    { $$ = $1; }
                ;

idlst1:         id
                    { ids = (idlst*) malloc(sizeof(idlst));
                      ids->id = (char*) $1;
                      ids->next = NULL;
       $$ = (int) ids; }
               | id ',' idlst1
                    { ids = (idlst*) malloc(sizeof(idlst));
                      ids->id = (char*) $1;
                      ids->next = (idlst*) $3;
       $$ = (int) ids; }
                ;


agtdecls:
             { $$ = (int) NULL; }
               | AGT id actiondecls agtdecls
    { agt = (agent*) malloc(sizeof(agent));
                      agt->name = (char*) $2;
                      agt->actions = (action*) $3;
                      agt->next = (agent*) $4;
                      $$ = (int) agt; }
                ;

actiondecls:    actiondecl
                    { act = (action*) $1;
                      act->next = NULL;
                      $$ = (int) act; }
              | actiondecl actiondecls
```

```
    { act = (action*) $1;
                      act->next = (action*) $2;
                      $$ = (int) act; }
                ;

actiondecl:     id VAR idlst PRE formula EFF formula
                    { act = (action*) malloc(sizeof(action));
                      act->name = (char*) $1;
                      act->var = (idlst*) $3;
                      act->pre = (formula*) $5;
                      act->eff = (formula*) $7;
                      $$ = (int) act; }
                ;

formula:        formula1 ARROW formula1 ',' formula1
                    { f = (formula*) malloc(sizeof(formula));
      f->type = ft_ite;
      f->f1 = (formula*) $1;
                      f->f2 = (formula*) $3;
                      f->f3 = (formula*) $5;
                      $$ = (int) f; }
              | formula1
                    { $$ = $1; }
                ;

formula1:       formula1 IMPL formula1
                    { f = (formula*) malloc(sizeof(formula));
                      f->f1 = (formula*) $1;
                      f->type = ft_impl;
                      f->f2 = (formula*) $3;
                      $$ = (int) f; }
              | formula1 BIIMPL formula1
                    { f = (formula*) malloc(sizeof(formula));
                      f->f1 = (formula*) $1;
                      f->type = ft_biimpl;
                      f->f2 = (formula*) $3;
                      $$ = (int) f; }
              | formula1 OR formula1
                    { f = (formula*) malloc(sizeof(formula));
                      f->f1 = (formula*) $1;
                      f->type = ft_or;
                      f->f2 = (formula*) $3;
                      $$ = (int) f; }
              | formula1 AND formula1
                    { f = (formula*) malloc(sizeof(formula));
                      f->f1 = (formula*) $1;
                      f->type = ft_and;
```

```
                        f->f2 = (formula*) $3;
                        $$ = (int) f; }
                | '~' formula1
                     { f = (formula*) malloc(sizeof(formula));
                        f->type = ft_neg;
                        f->f1 = (formula*) $2;
                        $$ = (int) f; }
                | '(' formula ')'
                     { f = (formula*) malloc(sizeof(formula));
                        f->type = ft_paren;
                        f->f1 = (formula*) $2;
                        $$ = (int) f; }
                | TRUE
                     { f = (formula*) malloc(sizeof(formula));
                        f->type = ft_true;
                        $$ = (int) f; }
                | FALSE
                     { f = (formula*) malloc(sizeof(formula));
                        f->type = ft_false;
                        $$ = (int) f; }
                | atom
                     { f = (formula*) malloc(sizeof(formula));
                        f->type = ft_atom;
                        f->atomic = (atom*) $1;
                        $$ = (int) f; }
                ;

atom:           id
                     { a = (atom*) malloc(sizeof(atom));
                        a->type = at_boolvar;
                        a->var = (char*) $1;
                        $$ = (int) a; }
                | numprop
    { a = (atom*) malloc(sizeof(atom));
                        a->type = at_numprop;
                        a->prop = (numberprop*) $1;
                        $$ = (int) a; }
                ;

numprop:        numberexp relop numberexp
                     { np = (numberprop*) malloc(sizeof(numberprop));
                        np->left  = (numberexp*) $1;
                        np->op = (relop) $2;
                        np->right = (numberexp*) $3;
                        $$ = (int) np; }
                ;
```

```
relop:          '='
                    { $$ = (int) ro_eq; }
              | NE
                    { $$ = (int) ro_ne; }
              | '>'
                    { $$ = (int) ro_gt; }
              | '<'
                    { $$ = (int) ro_lt; }
                ;

numberexp:      id
                    { ne = (numberexp*) malloc(sizeof(numberexp));
                      ne->type = nt_var;
                      ne->var = (char*) $1;
                      $$ = (int) ne; }
              | number
                    { ne = (numberexp*) malloc(sizeof(numberexp));
                      ne->type = nt_number;
                      ne->number = (int) $1;
                      $$ = (int) ne; }
              | numberexp '+' numberexp
                    { ne = (numberexp*) malloc(sizeof(numberexp));
                      ne->type = nt_plus;
                      ne->left = (numberexp*) $1;
                      ne->right = (numberexp*) $3;
                      $$ = (int) ne; }
              | numberexp '-' numberexp
                    { ne = (numberexp*) malloc(sizeof(numberexp));
                      ne->type = nt_minus;
                      ne->left = (numberexp*) $1;
                      ne->right = (numberexp*) $3;
                      $$ = (int) ne; }
                ;

id:             ID
                    { s = (char*) malloc(MAXNAMELENGTH);
                      strcpy(s,yytext);
                      $$ = (int) s; }
                ;

number:         NUMBER
                    { $$ = (int) atoi(yytext);  }
                ;

%%
#include <stdio.h>
#include <stdlib.h>
```

```
#include "domain.h"

#define MAXNAMELENGTH 128

extern char *yytext; /* defined in mnp.l */
extern int lineno;    /* defined in mnp.l */

mnpproblem *mnpprob;  /* pointer to abstract syntax (only extern var)*/
agent      *agt;
formula    *f;
varlst     *vars;
vartype    *vatype;
idlst      *ids;
char       *s;
action     *act;
atom       *a;
numberprop *np;
numberexp  *ne;


yyerror(s)
char *s;
{
        fflush(stdout);
        printf("\n%s! lineno:%d at:\"%s\"\n", s, lineno, yytext);
}
```

D.4 Header Files

D.4.1 Analyse.hpp

```
/*****************************************************
 * File  : analyse.hpp
 * Desc. : header file for analyse.cc
 * Author: Rune M. Jensen CS, CMU (IAU,DTU)
 * Date  : 3/29/99
 *****************************************************/

#ifndef ANALYSEHPP
#define ANALYSEHPP

void bddanalyse(char *bbdinfile, char *satoutfile,int nodenum,domdef *def);
void dumdomain(char *formula);

#endif
```

D.4.2 Bddprint.hpp

```
/*****************************************************
 * File  : bddprint.hpp
 * Desc. : header for bddprint.cc
 * Author: Rune M. Jensen, CS, CMU, (IAU, DTU)
 * Date  : 2/26/99
 *****************************************************/

#ifndef BDDPRINTHPP
#define BDDPRINTHPP

#include <stdio.h>
#include <iostream.h>
#include <fstream.h>
#include <bdd.h>
#include "fsm.hpp"
#include "common.hpp"

struct bddprintabs {
  FILE       *setfile;
  domdef     *def;
  ofstream    outfile;
  char        outfilename[MAXNAMELENGTH];
  int        *actlen; // the length of the action output must be explicitly stored
  int         parse(int *,int);
  void        reset();
  void        bddprint(bdd b);
  void        bddprintv(bdd b);
  void        bddinfo(bdd b);
  void        bddprinthead();
  void        bddprintline(bdd b);
              bddprintabs(char *filename,domdef *d);
};

#endif
```

D.4.3 Common.hpp

```
/*****************************************************
 * File  : common.h
 * Desc. : Shared defines for MNP files
 * Author: Rune M. Jensen
 * Date  : 2/19/99
 *****************************************************/

#ifndef COMMONHPP
#define COMMONHPP

#define MAXNAMELENGTH 128
#define DEBUG         0
#define MAXPLANLENGTH 256

#endif
```

D.4.4 Dissets.hpp

```
/*****************************************************
 * File  : dissets.hpp
 * Desc. : Header for Disjoint set ADT
 * Author: Rune M. Jensen
 * Date  : 3/16/99
 *****************************************************/

#ifndef DISSETSHPP
#define DISSETSHPP

#include "domain.hpp"

struct disnode {
  struct disnode *parent;
  int size;
  int setnum;
};

struct idmap {
  char *id;
  disnode *node;
  struct idmap *next;
  disnode *lookup(char *id);
};


struct disset {
  idlst **partvars; // array of variables in each subset
  int   subsetnum;
  idmap *map;
  disset()
    { map = NULL; }
  void makeset(char *id);
  disnode *find(char *id);
  disnode *setunion(disnode *v, disnode *w);
  void enumerate();
  void mkpartvars();
};

#endif
```

D.4.5 Domain.h

```
/*****************************************************
 * File  : doamin.h
 * Desc. : Domain theory representation for MNP-domains,
 *         c compatible.
 * Author: Rune M. Jensen
 * Date  : 3/3/99
 *****************************************************/



/*****************************************************
 *                Number expressions                 *
 *****************************************************/

typedef enum _relop {ro_lt,ro_gt,ro_eq,ro_ne} relop;
typedef enum _exptype {nt_number, nt_plus, nt_minus, nt_var} exptype;


typedef struct _numberexp {
  exptype type;
  int number;
  struct _numberexp  *left;
  struct _numberexp  *right;
  char *var;
} numberexp;

typedef struct _numberprop {
  relop op;
  numberexp *left;
  numberexp *right;
} numberprop;



/*****************************************************
 *                      atoms                        *
 *****************************************************/

typedef enum _atomtype {at_numprop, at_boolvar} atomtype;

typedef struct _atom {
  atomtype type;
  numberprop *prop;
  char *var;
} atom;
```

```
/*************************************************
 *               Fomula representation           *
 *************************************************/


typedef enum _formulatype {ft_atom,ft_neg,ft_and,ft_or,ft_impl,
             ft_biimpl,ft_ite, ft_true, ft_false, ft_paren} formulatype;

typedef struct _formula {
  formulatype type;
  atom *atomic;
  struct _formula  *f1;
  struct _formula  *f2;
  struct _formula  *f3;
} formula;


/*************************************************
 *               Action representation           *
 *************************************************/

typedef struct _idlst {
  char *id;
  struct _idlst *next;
} idlst;


typedef struct _action {
  char    *name;
  idlst   *var;
  formula *pre;
  formula *eff;
  struct _action *next;
} action;


/*************************************************
 *            agent  representation              *
 *************************************************/

typedef struct _agent {
char   *name;
action *actions;
struct _agent *next;
} agent;
```

```
/*************************************************
 *       varlst, vartype representation          *
 *************************************************/

typedef enum _vtype {vt_bool, vt_scalar} vtype;

typedef struct _vartype {
  vtype type;
  int   range;
} vartype;



typedef struct _varlst {
  vartype *type;
  idlst *ids;
  struct _varlst *next; /* next varlst */
} varlst ;



/*************************************************
 *               Problem representation          *
 *************************************************/

typedef struct _mnpproblem {
  varlst     *vars;
  agent      *sys;
  agent      *env;
  formula    *init;
  formula    *goal;
} mnpproblem;
```

D.4.6 Domain.hpp

```
/*****************************************************
 * File  : doamin.hpp
 * Desc. : Domain theory representation for MNP-domains,
 *         c++ compatible.
 * Author: Rune M. Jensen
 * Date  : 3/3/99
 *****************************************************/

#ifndef DOMAINHPP
#define DOMAINHPP


/*************************************************
 *               Number expressions              *
 *************************************************/

enum relop {ro_lt,ro_gt,ro_eq,ro_ne};
enum exptype {nt_number, nt_plus, nt_minus, nt_var};


struct numberexp {
  exptype type;
  int number;
  struct numberexp  *left;
  struct numberexp  *right;
  char *var;
  void minus2plus();
  void print();
};

struct numberprop {
  relop op;
  numberexp *left;
  numberexp *right;
  void minus2plus();
  void print();
};



/*************************************************
 *                       atoms                   *
 *************************************************/

enum atomtype {at_numprop, at_boolvar};
```

```
struct atom {
  atomtype type;
  numberprop *prop;
  char *var;
  void print();
};


/*************************************************
 *               Fomula representation           *
 *************************************************/


enum formulatype {ft_atom,ft_neg,ft_and,ft_or,ft_impl,
              ft_biimpl,ft_ite, ft_true, ft_false, ft_paren};

struct formula {
  formulatype type;
  atom *atomic;
  struct formula  *f1;
  struct formula  *f2;
  struct formula  *f3;
  void print();
};


/*************************************************
 *               Action representation           *
 *************************************************/

struct idlst {
  char *id;
  struct idlst *next;
  void print();
  idlst(char *nm, struct idlst *nx)
    { id = nm; next = nx; }
  struct idlst *lookup(char *name);
};


struct action {
  char    *name;
  idlst   *var;
  formula *pre;
  formula *eff;
  struct action *next;
  void print();
```

```
};


/*************************************************
 *            agent  representation               *
 *************************************************/

struct agent {
  char   *name;
  action *actions;
  struct agent *next;
  void print();
};


/*************************************************
 *       varlst, vartype representation           *
 *************************************************/

enum vtype {vt_bool, vt_scalar};

struct vartype {
  vtype type;
  int   range;
  void print();
};



struct varlst {
  vartype *type;
  idlst *ids;
  struct varlst *next; /* next varlst */
  void print();
};



/*************************************************
 *               Problem representation           *
 *************************************************/

struct mnpproblem {
  varlst     *vars;
  agent      *sys;
  agent      *env;
  formula    *init;
```

```
  formula    *goal;
  void print();
};

#endif
```

D.4.7 Fsm.hpp

```
/*****************************************************
 * File  : fsm.hpp
 * Desc. : Structures for constructing the FSM
 *         transition relation for MNP problems.
 * Author: Rune M. Jensen, CS, CMU, (IAU, DTU)
 * Date  : 3/3/99
 *****************************************************/

#ifndef FSMHPP
#define FSMHPP

#include <bdd.h>
#include <fstream.h>
#include "domain.hpp"
#include "dissets.hpp"

/*****************************************************
 * structures for representing domain info          *
 *****************************************************/

//**************  vardef ******************************

struct actdef {
  char   *name;
  int     number;
  action *act;
  struct actdef *next;
  actdef(char *id,int n, action *a, struct actdef *nt)
    {name = id; number = n; act = a; next = nt;}
  void print(ofstream &out,int *actlen);
  void print();
};


//*************  agtdef    ***************************

enum sysclass {sc_env, sc_sys};

struct agtdef {
  char *name;
  int   number;
  int   actnum;
  sysclass sclass;
  actdef  *acts;
  struct agtdef *next;
  agtdef(char *nm, int n, int an, sysclass sc, actdef *ac, struct agtdef *nx)
```

```
    { name = nm; number = n; actnum = an; sclass = sc; acts = ac; next = nx; }
  struct agtdef *lookup(int i);
  void print(ofstream &out,int *actlen);
  void print();
};

//*************  vardef    ***************************


struct  vardef {
  char      *name;     // the action var has a name corresponding to their agent num.
  vtype      type;     // boolean, scalar
  int        length;   // no. of variables
  int        *var;     // mapping to bddvars
  struct vardef *next;
  vardef(char *n,vtype vt,int l, int *v, struct vardef *nt)
     { name = n; type = vt; length = l; var = v; next = nt; }
  void print();
  struct vardef *lookupnew(char *name);
  struct vardef *lookup(int i);
  struct vardef *lookup(char *name);
};


//       bdd variable location info
struct varrels {
  bddPair *old2new;
  bddPair *new2old;
  bdd     oldstate;
  bdd     *newstate; // an array corresponding to partition numbers
  bdd     envactions;   // (to enable early quantification)
  bdd     sysactions;
};


//*************  domdef     ************************

//  total bdd variable layout description
struct domdef {
  agtdef    *agt;
  int        envnum;
  int        agtnum;
  int       *actnum;
  action   ***act;
  vardef    *var;
  idlst     *varnames;
  int        bddvarnum;
```

```
  disset     *dset;
  varrels    *rels;
  domdef(agtdef *a, int en, int agnum, int *acnum, action ***ac, vardef *v,
 idlst *varn, int bvc, disset *ds )
    { agt = a;  envnum=en; agtnum = agnum; actnum = acnum; act = ac; var = v;
    varnames = varn; bddvarnum = bvc; dset = ds;}
  void print();
};



/*************************************************
 *          structure  for representing          *
 *            binary number expresions           *
 *************************************************/

struct binary {
  int  digitnum;
  bdd *digit;
  binary(int dn, bdd *d)
    { digitnum = dn; digit = d; }
};

/*************************************************
 *          structure  for representing          *
 *       a partitioned transition relation       *
 *************************************************/

struct bddtrel {
  bdd *T; // transition partitions
  bdd *IV; // corresponding invariant part
  bdd A;   // action value condition
  bddtrel(int size)
    { T = new bdd[size]; IV = new bdd[size]; }
};




// *********** function prototypes ********************
void mkbddblocks(domdef *def);
domdef *mkdomdef(mnpproblem *mnp);
varrels *mkvarrels(domdef *def);
void adjust2order(int *order, domdef *def);
```

```
bdd formula2bdd(formula *f, domdef *def) ;
bdd atom2bdd(atom *a, domdef *def) ;

bdd numprop2bdd(numberprop *prop,domdef *def);
binary *numexp2binary(numberexp *num,domdef *def);

bdd int2bdd(int *var,int length,int n)  ;

bdd I(int i, int j, domdef *def) ;
bdd F(char *v, int i, int j, domdef *def) ;
bdd ADEF(domdef *def);
bdd IND(int i1, int j1, int i2, int j2, domdef *def);
bdd AC(domdef *def);
bdd P(int i, int j, domdef *def) ;
bdd E(int i, int j, domdef *def) ;
bdd IV(int partno,domdef *def) ;
bdd A(int i,int j,domdef *def) ;

bddtrel *T(domdef *def) ;
#endif
```

D.4.8 Main.hpp

```
/*****************************************************
 * File  : main.hpp
 * Desc. : Header for mnp main file main.cc
 * Author: Rune M. Jensen CS, CMU, (IAU,DTU)
 * Date  : 4/3/99
 *****************************************************/

#ifndef MAINHPP
#define MAINHPP

//extern
extern mnpproblem *mnpprob; // parse tree structure
#endif
```

D.4.9 Plan.hpp

```
/*****************************************************
 * File  : plan.hpp
 * Desc. : Header for plan.cc
 * Author: Rune M. Jensen CS, CMU, (IAU, DTU)
 * Date  : 4/4/99
 *****************************************************/

#ifndef PLANHPP
#define PLANHPP

#include <bdd.h>
#include "fsm.hpp"
#include "bddprint.hpp"

// function prototypes

// aux. functions
bdd prunestates(bdd preimage, bdd acc);
bdd projact(bdd stateact_rules, domdef *def);
bdd image(bddtrel *tr, bdd s,domdef *def);
bdd successor(bdd action,domdef *def);
void  printplan(bddtrel *tr,bdd init,bdd goal,bdd sa,char *planfile,domdef *def);
bdd strongpreimage(bddtrel *tr, bdd acc,domdef *def);
bdd weakpreimage(bddtrel *tr, bdd acc,domdef *def);

// planning algorithm 1
bdd preimagef(bddtrel *tr, bdd acc,domdef *def);
int plan1(bddtrel *tr, bdd init, bdd goal, bdd *sa,domdef *def);

// nondet algorithms
// planning algorithm 2
int strongplan(bddtrel *tr, bdd init, bdd goal, bdd *sa,domdef *def);

// planning algorithm 3
int strongcyclicplan(bddtrel *tr, bdd init, bdd goal, bdd *sa,domdef *def);
bdd OSA(bddtrel *tr,bdd CSA,bdd s,domdef *def);
void faircycles(bddtrel *T, bdd *acc, bdd *sa,bdd *oldacc, domdef *def);

// planning algorithm 4
bdd preimagef2(bddtrel *tr, bdd acc,domdef *def);
int plan4(bddtrel *tr, bdd init, bdd goal, bdd *sa,domdef *def);
#endif
```

D.4.10 Reorder.hpp

```
/*****************************************************
 * File  : reorder.hpp
 * Desc. : header for reorder.cc
 * Rune M. Jensen CS, CMU, (IAU, DTU)
 * Date  : 4/4/99
 *****************************************************/

#ifndef REORDERHPP
#define REORDERHPP

// prototypes
void writeorder(char *filename,domdef *def);
int *readorder(char *filename,domdef *def);
#endif
```

D.4.11 Time.hpp

```
/*****************************************************
 * File  : time.hpp
 * Desc. : Header file for time.cc
 * Author: Rune M. Jensen
 * Date  : 4/8/99
 *****************************************************/

#ifndef TIMEHPP
#define TIMEHPP

void startwatch();
void stopwatch();
#endif
```

D.5 Source Files

D.5.1 Analyse.cc

```
/*****************************************************
 * File  : analyse.cc
 * Desc. : tool for analysing BDDs
 * Author: Rune M. Jensen CS, CMU, (IAU,DTU)
 * Date  : 3/29/99
 *****************************************************/

#include <bdd.h>
#include <string.h>
#include <stdio.h>
#include "bddprint.hpp"
#include "reorder.hpp"
#include "analyse.hpp"
#include "common.hpp"
#include "plan.hpp"
#include "time.hpp"


// parsetree structure
extern mnpproblem *mnpprob;

void bddanalyse(char *bddinfile, char *satoutfile, int nodenum,domdef *def) {

  bdd f,b,inbdd,init,goal;
  int loop,i,knownc;
  int *ordering;
  char command[32];
  char argument[512];
  extern FILE *yyin;
  bddprintabs *bp;
  char orderinfile[MAXNAMELENGTH];
  char cnames[18][16] = {"print","con","nodes","sat","logsat","help","quit","reset","file",
                         "p"    ,"c"  ,"n"    ,"s"  ,"l"     ,"h"   ,"q",   "r",    "f" };

  // read reordering information and adjust domdef
  sprintf(orderinfile,"%so",bddinfile);
  ordering = readorder(orderinfile,def);
  adjust2order(ordering,def);


  // initialize bdd package
  // NOTE: addref system breaks down if node number is
  // increased during load of file
  bdd_init(nodenum,10000);
```

```
  bdd_setvarnum(def->bddvarnum);


  // read bdd infile
  if (bdd_fnload(bddinfile,inbdd))
      {
cout << "mnp: Cannot load bdd from file \"" << bddinfile << "\"\nexiting\n";
exit(1);
      }


  // open sat outfile
  bp = new bddprintabs(satoutfile,def);

  // main command loop
  cout << "\nAnalyse BDD Type \"help\" for command list\n";
  argument[0] = '\0';
  b = inbdd;
  loop = 1;
  while (loop)
    {
      cout << "$ ";
      gets(command);

      knownc = 0;
      for (i=0; i < 18; i++)
if (!strcmp(command,cnames[i])) knownc = 1;
      if (!knownc && command[0] != 0)
  cout << "unknown command\n";
      else

switch (command[0]) {

case 'p' :
  cout << "vertical or horizontal (v/h): ";
  gets(argument);
  if (argument[0] == 'v')
    {
      bp->bddprintv(b);
      bp->outfile.flush();
    }
  else
    {
      bp->bddprint(b);
      bp->outfile.flush();
    }
  break;
```

```
case 'c' :
  cout << "formula: ";
  gets(argument);
  dumdomain(argument);
  yyin = fopen("deleteme.mnp","r");
  if (yyin == NULL)
    {
      printf("analyse.cc bddanalyse : cannot open deleteme.mnp\nexiting\n");
      exit(1);
    }

  if (!yyparse())
    {
      // parsing succeeded
      f = formula2bdd(mnpprob->init,def);
      b &= f;
    }
  fclose(yyin);
  break;

case 'f' :
  cout << "constraint file: ";
  gets(argument);
  yyin = fopen(argument,"r");
  if (yyin == NULL)
    printf("Cannot open file : \"%s\"\n",argument);
  else
    {
      if (!yyparse())
{
  // parsing succeeded
  f = formula2bdd(mnpprob->init,def);

  // start timing
  startwatch();
  b &= f;
  // stop timing
  stopwatch();
}
      fclose(yyin);
    }
  break;

case 'n' :
  cout << "BDD node count : " << bdd_nodecount(b) << "\n";
  break;
```

```
case 's' :
  cout << "BDD sat count : " << bdd_satcount(b) << "\n";
  break;
case 'l' :
  cout << "BDD log sat count : " << bdd_satcountln(b) << "\n";
  break;

case 'h' :
  cout << "(p)rint    : Print the bdd to file.\n";
  cout << "(c)on      : Constrain bdd by formula.\n";
  cout << "(f)ile     : Constrain bdd by file with formula.\n";
  cout << "(n)odes    : Print number of nodes in bdd.\n";
  cout << "(s)at      : Print sat count.\n";
  cout << "(l)ogsat   : Print logsat count.\n";
  cout << "(h)elp     : Print this message.\n";
  cout << "(r)eset    : Reset BDD to input BDD and reset outfile.\n";
  cout << "(q)uit     : End session.\n";
  break;

case 'r' :
  b = inbdd;
  bp->reset();
  break;


case 'q' :
  loop = 0;
  break;
}
    }


  // close bdd package
  bdd_done();

}




void dumdomain(char *formula) {
  FILE *out;

  // writes dummy mnp domain with formula stated
  // as init to lex input.
  out = fopen("deleteme.mnp","w+");
```

```
  if (out == NULL)
    {
      printf("analyse.cc dumdomain : cannot open deleteme.mnp\nexiting\n");
      exit(1);
    }
  fprintf(out,"variables bool dum\n");
  fprintf(out,"system agt: dum dum var: pre: true eff: true\n");
  fprintf(out,"environment initially %s goal true",formula);
  fclose(out);
}
```

D.5.2 Bddprint.cc

```
/*****************************************************
 * File  : bddprint.cc
 * Desc. : functions for nice prints of BDD trans. rel.
 *         of MNP domains.
 * Author: Rune M. Jensen, CS, CMU, (IAU, DTU)
 * Date  : 3/4/99
 *****************************************************/

#include <iostream.h>
#include <fstream.h>
#include <stream.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include "fsm.hpp"
#include "bddprint.hpp"
#include "common.hpp"


bddprintabs::bddprintabs(char *filename,domdef *d) {
  strcpy(outfilename,filename);
  outfile.open(filename,ios::out);
  def = d;
}

void bddprintabs::reset() {
  outfile.close();
  outfile.open(outfilename,ios::out);
}




// reads one variable assignment
// of the bdd_printset format
int bddprintabs::parse(int *as,int aslen) {
  int i, varno, ordno,value;
  char c;

  //check if file T or False
  c = getc(setfile);

  if ( c == 'F' || c==EOF )
    return 0;
  else
```

```
    if ( c == 'T' )
      {
// set all variables to -1 (any value)
for (i=0;i<aslen;i++)
  as[i] = -1;
return 1;
      }
    else
      if ( c == '<' )
{
  // there is an assignment to read

  // set all variables to -1 (any value)
  for (i=0;i<aslen;i++)
    as[i] = -1;

  // read assignment
  while(c != '>') {

    // pre: pointer is at first digit
    // in var. assignment
    if (fscanf(setfile,"%d/%d:%d",&varno,&ordno,&value) != 3)
      {
cout << "\nbddprint.cc printbddabs::parse: scan error";
cout << "\nexiting\n\n";
      }
    as[varno] = value;
    // post: pointer is just after last difit
    // in var. assignment
    c = getc(setfile);
  }
  return 1;
}
}




// aux. for printbdd
void mkspace(ofstream &out,int n) {
  int i;
  for (i=0;i<n;i++)
    out << " ";
}

int max(int a,int b) {
  return a > b ? a : b;
```

```
}

int bin2intstr(int *as,int *var,int length,char *buf) {
  int i,exp,res;

  res = 0;
  exp = length - 1;
  for (i= 0; i < length; i++)
    if (as[var[i]] < 0)
      return 0;
    else
      res += as[var[i]]*int(pow(2,exp--));
    sprintf(buf,"%d",res);
    return 1;
}


char *rawrep(int *as,int *var,int length,char *buf) {
  int i;

  strcpy(buf,"");
  for (i=0; i<length; i++)
    switch (as[var[i]]) {
    case 0 :
      strcat(buf,"0");
      break;
    case 1 :
      strcat(buf,"1");
      break;
    default:
      strcat(buf,"*");
      break;
    }
  return buf;
}




void bddprintabs::bddprint(bdd b) {

  agtdef     *agtd,*agt;
  idlst      *varnames;
  vardef     *vard,*var;
  int        *as;
  int         i,len,plen,aslen;
  char        buf[MAXNAMELENGTH];
```

```
// open outputfile for bdd_printset
if ( (setfile = fopen("deleteme.txt","w+")) == NULL)
  {
    cout << "bddprint.cc printbddabs::bddprintline: ";
    cout << "cannot open \"deleteme.txt\"\nexiting\n\n";
    exit(1);
  }

// make the set
bdd_fprintset(setfile,b);
fflush(setfile);
rewind(setfile);

// print action description (and fill the action length table)
agtd = def->agt;
actlen = new int[def->agtnum];
agtd->print(outfile,actlen);
outfile << "\n";


// write header
// agt names
for (i=0; i < def->agtnum; i++) {
  agt = agtd->lookup(i);
  outfile << agt->name;
  mkspace(outfile,actlen[agt->number] - strlen(agt->name));
}

// write header
// write all old state variables
  varnames = def->varnames;
  vard = def->var;

  while (varnames) {
    var = vard->lookup(varnames->id);
    len = max(strlen(var->name) + 2, (var->length + 1)*2);
    outfile << var->name;
    mkspace(outfile,len - strlen(var->name));
    varnames = varnames->next;
}
outfile << "\n";

// Init assignment table
aslen = def->bddvarnum;
as = new int[aslen];
```

```
  // go through all lines in the table
  while (parse(as,aslen))
    {
      // write action var info
      for (i=0; i<def->agtnum; i++)
{
  var = vard->lookup(i);
  if (bin2intstr(as,var->var,var->length,buf))
    {
      if ( atoi(buf) < agtd->lookup(i)->actnum )
{
  // action can be written by its name
  outfile << (def->act)[i][atoi(buf)]->name;
  mkspace(outfile,actlen[i] - strlen( (def->act)[i][atoi(buf)]->name));
}
      else
{
  // action doesn't exist write the number
  outfile << buf;
  mkspace(outfile,actlen[i] - strlen(buf));
}
    }
  else
    {
      // the line defines a set of actions
      // we have to write the number pattern
      outfile << rawrep(as,var->var,var->length,buf);
      mkspace(outfile,actlen[i] - strlen(buf));
    }
}

      // write individual variable info
      varnames = def->varnames;
      while (varnames)
{
  var = vard->lookup(varnames->id);
  len = max(strlen(var->name) + 2, (var->length + 1)*2);
  plen = 0;
       if (var->type == vt_bool)
    {
      // write old state var
      outfile << rawrep(as,var->var,var->length,buf) << " ";
      var = vard->lookupnew(varnames->id);
      // write new state var
      outfile << rawrep(as,var->var,var->length,buf);
      mkspace(outfile,len - 3);
```

```
    }
  else
    {
      // variable is scalar
      // write old state scalar variable
      if (bin2intstr(as,var->var,var->length,buf))
{
  // scalar can be represented as a number
  outfile << buf << " ";
  plen += strlen(buf) + 1;
}
      else
{
  // scalar must be represented by its number pattern
  outfile << rawrep(as,var->var,var->length,buf) << " ";
  plen += strlen(buf) + 1;
}

      var = vard->lookupnew(varnames->id);

      // write new state scalar variable
      if (bin2intstr(as,var->var,var->length,buf))
{
  // scalar can be represented as a number
  outfile << buf << " ";
  plen += strlen(buf) + 1;
}
      else
{
  // scalar must be represented by its number pattern
  outfile << rawrep(as,var->var,var->length,buf) << " ";
  plen += strlen(buf) + 1;
}
      mkspace(outfile,len-plen);
    }
  // next variable pair
  varnames = varnames->next;
}
      // next line
      outfile << "\n";
    }
  outfile.flush();
  fclose(setfile);
}


void bddprintabs::bddinfo(bdd b) {
```

```
  //write BDD size information
  cout << "\n\nBDD node count    = " << bdd_nodecount(b);
  cout << "\nlog(BDD satcount) = " << bdd_satcountln(b) << "\n\n";
}


void bddprintabs::bddprinthead(){

  agtdef     *agtd,*agt;
  idlst      *varnames;
  vardef     *vard,*var;
  int         len,i;

  // print action description (and fill the action length table)
  agtd = def->agt;
  actlen = new int[def->agtnum];
  agtd->print(outfile,actlen);
  outfile << "\n";

  // write header
  // agt names
  for (i=0; i < def->agtnum; i++) {
    agt = agtd->lookup(i);
    outfile << agt->name;
    mkspace(outfile,actlen[agt->number] - strlen(agt->name));
  }

  // write header
  // write all old state variables
    varnames = def->varnames;
    vard = def->var;

    while (varnames) {
      var = vard->lookup(varnames->id);
      len = max(strlen(var->name) + 2, (var->length + 1)*2);
      outfile << var->name;
      mkspace(outfile,len - strlen(var->name));
      varnames = varnames->next;
  }
  outfile << "\n";
}



void bddprintabs::bddprintline(bdd b) {

  agtdef     *agtd;
```

```
  idlst      *varnames;
  vardef     *vard,*var;
  int        *as;
  int         i,len,plen,aslen;
  char        buf[MAXNAMELENGTH];


  vard = def->var;
  agtd = def->agt;

  // open outputfile for bdd_printset
  if ( (setfile = fopen("deleteme.txt","w+")) == NULL)
    {
      cout << "bddprint.cc printbddabs::bddprintline: ";
      cout << "cannot open \"deleteme.txt\"\nexiting\n\n";
      exit(1);
    }

  // make the set
  bdd_fprintset(setfile,b);
  fflush(setfile);
  rewind(setfile);

  // Init assignment table
  aslen = def->bddvarnum;
  as = new int[aslen];

  // go through all lines in the table
  while (parse(as,aslen))
    {
      // write action var info
      for (i=0; i<def->agtnum; i++)
{
  var = vard->lookup(i);
  if (bin2intstr(as,var->var,var->length,buf))
    {
      if ( atoi(buf) < agtd->lookup(i)->actnum )
{
  // action can be written by its name
  outfile << (def->act)[i][atoi(buf)]->name;
  mkspace(outfile,actlen[i] - strlen( (def->act)[i][atoi(buf)]->name));
}
      else
{
  // action doesn't exist write the number
  outfile << buf;
  mkspace(outfile,actlen[i] - strlen(buf));
```

```
}
    }
  else
    {
      // the line defines a set of actions
      // we have to write the number pattern
      outfile << rawrep(as,var->var,var->length,buf);
      mkspace(outfile,actlen[i] - strlen(buf));
    }
}

      // write individual variable info
      varnames = def->varnames;
      while (varnames)
{
  var = vard->lookup(varnames->id);
  len = max(strlen(var->name) + 2, (var->length + 1)*2);
  plen = 0;
       if (var->type == vt_bool)
    {
      // write old state var
      outfile << rawrep(as,var->var,var->length,buf) << " ";
      var = vard->lookupnew(varnames->id);
      // write new state var
      outfile << rawrep(as,var->var,var->length,buf);
      mkspace(outfile,len - 3);
    }
  else
    {
      // variable is scalar
      // write old state scalar variable
      if (bin2intstr(as,var->var,var->length,buf))
{
  // scalar can be represented as a number
  outfile << buf << " ";
  plen += strlen(buf) + 1;
}
      else
{
  // scalar must be represented by its number pattern
  outfile << rawrep(as,var->var,var->length,buf) << " ";
  plen += strlen(buf) + 1;
}

      var = vard->lookupnew(varnames->id);

      // write new state scalar variable
```

```
      if (bin2intstr(as,var->var,var->length,buf))
{
  // scalar can be represented as a number
  outfile << buf << " ";
  plen += strlen(buf) + 1;
}
      else
{
  // scalar must be represented by its number pattern
  outfile << rawrep(as,var->var,var->length,buf) << " ";
  plen += strlen(buf) + 1;
}
      mkspace(outfile,len-plen);
    }
  // next variable pair
  varnames = varnames->next;
}
      // next line
      outfile << "\n";
    }
  fclose(setfile);
}




void bddprintabs::bddprintv(bdd b) {

  agtdef     *agtd,*agt;
  idlst      *varnames;
  vardef     *vard,*var;
  int        *as;
  int         i,aslen;
  char        buf[MAXNAMELENGTH];


  // open outputfile for bdd_printset
  if ( (setfile = fopen("deleteme.txt","w+")) == NULL)
    {
      cout << "bddprint.cc printbddabs::bddprintline: ";
      cout << "cannot open \"deleteme.txt\"\nexiting\n\n";
      exit(1);
    }

  // make the set
  bdd_fprintset(setfile,b);
```

```
  fflush(setfile);
  rewind(setfile);

  // print action description (and fill the action length table)
  agtd = def->agt;
  vard = def->var;
  actlen = new int[def->agtnum];
  agtd->print(outfile,actlen);
  outfile << "\n";

  // Init assignment table
  aslen = def->bddvarnum;
  as = new int[aslen];

  // go through all lines in the table
  while (parse(as,aslen))
    {
      // write action var info
      outfile << "ACTIONS\n";
      for (i=0; i<def->agtnum; i++)
{
  var = vard->lookup(i);
  outfile << agtd->lookup(i)->name << ": ";
  if (bin2intstr(as,var->var,var->length,buf))
    {
      if ( atoi(buf) < agtd->lookup(i)->actnum )
// action can be written by its name
outfile << (def->act)[i][atoi(buf)]->name << "\n";
      else
// action doesn't exist write the number
outfile << buf << "\n";
    }
  else
    {
      // the line defines a set of actions
      // we have to write the number pattern
      outfile << rawrep(as,var->var,var->length,buf) << "\n";
    }
}

      // write individual variable info
      outfile << "VARIABLES\n";
      varnames = def->varnames;
      while (varnames)
{
  outfile << varnames->id << ": ";
  var = vard->lookup(varnames->id);
```

```
       if (var->type == vt_bool)
    {
      // write old state var
      outfile << rawrep(as,var->var,var->length,buf) << " ";
      var = vard->lookupnew(varnames->id);
      // write new state var
      outfile << rawrep(as,var->var,var->length,buf) << "\n";
    }
  else
    {
      // variable is scalar
      // write old state scalar variable
      if (bin2intstr(as,var->var,var->length,buf))
// scalar can be represented as a number
  outfile << buf << " ";
      else
// scalar must be represented by its number pattern
outfile << rawrep(as,var->var,var->length,buf) << " ";

      var = vard->lookupnew(varnames->id);

      // write new state scalar variable
      if (bin2intstr(as,var->var,var->length,buf))
// scalar can be represented as a number
outfile << buf << "\n";
      else
// scalar must be represented by its number pattern
outfile << rawrep(as,var->var,var->length,buf) << "\n";
    }
  // next variable pair
  varnames = varnames->next;
}
      // next line
      outfile << "\n";
    }
  outfile.flush();
  fclose(setfile);
}
```

D.5.3 Dissets.cc

```
/*****************************************************
 * File  : dissets.cc
 * Desc. : Implementation of a disjoint set ADT,
 *         using both union by size and path compression.
 *         Note: id strings are looked up in a linked
 *               list for id# < 20, this is efficient.
 *               E.g. a splaytree should be used if id# > 20.
 * Author: Rune M. Jensen
 * Date  : 3/16/99
 *****************************************************/

#include <string.h>
#include <stream.h>
#include "common.hpp"
#include "dissets.hpp"

disnode *idmap::lookup(char *nm)  {
  if (this)
    if (!strcmp(nm,id))
      return node;
    else
      return next->lookup(nm);
  else
    return NULL;
}


void disset::makeset(char *id) {
  disnode *n;
  idmap *im;
  char *str;

  // add new disnode  node
  n = new disnode;
  n->parent = NULL;
  n->size = 1;

  str = new char[MAXNAMELENGTH];
  strcpy(str,id);

  // add node in idmap
  im = new idmap;
  im->id = str;
  im->node = n;
  im->next = map;
  map = im;
```

```
}

// IN
//  id : variable identifier
// OUT
//  disnode : identifier for id's set
disnode *disset::find(char *id) {
  disnode *x,*y,*z,*tmp;

  x = map->lookup(id);
  if (x == NULL)
    {
      cout << "dissets.cc disset::find : cannot find id \"" << id;
      cout << "\" in idmap \nexiting\n\n";
      exit(1);
    }

  y = x;
  // find set id.
  while (y->parent)
    y = y->parent;

  // compress the path
  z = x;
  while (z->parent)
    {
      tmp = z->parent;
      z->parent = y;
      z = tmp;
    }
  return y;
}


disnode *disset::setunion(disnode *v, disnode *w) {
  if (v->size < w->size)
    {
      v->parent = w;
      w->size += v->size;
      return w;
    }
  else
    {
      w->parent = v;
      v->size += w->size;
      return v;
    }
```

```
}

// OUT
//  number of subsets
// Identifies all sets by a number
// (changes size of top nodes to an identifier number)
void disset::enumerate() {
  idmap *m;
  int i;

  m = map;
  i = 0;

  while (m)
    {
      if (!m->node->parent)
m->node->setnum = i++;
      m = m->next;
    }
  subsetnum = i;
}



// pre: disset is enumerated
void disset::mkpartvars() {
  int i,setnum;
  idmap *m;
  idlst *n;

  partvars = new idlst*[subsetnum];

  for (i=0; i<subsetnum; i++)
    partvars[i] = NULL;

  m = map;
  while (m)
    {
      setnum = this->find(m->id)->setnum;
      n = new idlst(m->id,partvars[setnum]);
      partvars[setnum] = n;

      m = m->next;
    }
}
```

D.5.4 Domain.cc

```
/*****************************************************
 * File  : domain.cc
 * Desc. : NP domain member functions
 *         (functions on the np problem description)
 * Author: Rune M. Jensen
 * Date  : 2/22/99
 *****************************************************/

#include <stream.h>
#include "domain.hpp"



/*************************************************
 *          access functions                     *
 *************************************************/

idlst *idlst::lookup(char *name) {
  if (this)
    if (!strcmp(name,id))
      return this;
    else
      return next->lookup(name);
  else
    return NULL;
}


/*************************************************
 *          Manipulating functions               *
 *************************************************/


// IN
//  this : numberprop
// OUT
//  numberprop with all '-' expressions eliminated
void numberprop::minus2plus() {
  numberexp *l,*r,*a,*b,*c,*d;

  left->minus2plus();
  right->minus2plus();

    if (left->type == nt_number || left->type == nt_var || left->type == nt_plus)
      {
if (right->type == nt_number || right->type == nt_var || right->type == nt_plus)
```

```
  {
    // left: +exp,   right: +exp
    // already ok: nothing to do
  }
else
  {
    // org left: +exp,   right: -exp
    a = left;
    b = right->left;
    r = right;
    r->type = nt_plus;
    r->left = a;
    l = b;
    left = r;
    right = l;
  }
      }
    else
      {
if (right->type == nt_number || right->type == nt_var || right->type == nt_plus)
  {
    // left: -exp,   right: +exp
    a = left->left;
    c = right;
    l = left;
      l->type = nt_plus;
    l->left = c;
    r = a;
    left = r;
    right = l;
  }
else
  {
    // left: -exp,   right: -exp
    b = left->right;
    d = right->right;
    l = left;
    l->type = nt_plus;
    l->right = d;
    r = right;
    r->type = nt_plus;
    r->right = b;
  }
      }
}
```

```
// IN
//  this : numberexp
// OUT
//  numberexp with only one top '-' expression
void numberexp::minus2plus() {
  numberexp *l,*r,*a,*b,*c,*d;

  switch (type) {
  case nt_number:
  case nt_var:
    // already ok: do nothing
    break;
  case nt_plus:
    left->minus2plus();
    right->minus2plus();
    if (left->type == nt_number || left->type == nt_var || left->type == nt_plus)
      {
if (right->type == nt_number || right->type == nt_var || right->type == nt_plus)
  {
    // org: +exp, left: +exp,   right: +exp
    // already ok: do nothing
  }
else
  {
    // org +exp, left: +exp,   right: -exp
    a = left;
    c = right->right;
    l = c;
    r = right;
    r->right = a;
    r->type = nt_plus;
            type = nt_minus;
    left = r;
    right = l;
  }
      }
    else
      {
if (right->type == nt_number || right->type == nt_var || right->type == nt_plus)
  {
    // org: +exp, left: -exp,   right: +exp
    b = left->right;
    c = right;
    l = left;
    l->right = c;
```

```
    l->type = nt_plus;
    r = b;
    type = nt_minus;
    left = l;
    right = r;
  }
else
  {
    // org +exp, left: -exp,   right: -exp
    b = left->right;
    c = right->left;
    l = left;
    l->type = nt_plus;
    l->right = c;
    r = right;
    r->type = nt_plus;
    r->left = b;
    type = nt_minus;
  }
      }
    break;
  case nt_minus:
    left->minus2plus();
    right->minus2plus();
    if (left->type == nt_number || left->type == nt_var || left->type == nt_plus)
      {
if (right->type == nt_number || right->type == nt_var || right->type == nt_plus)
  {
    // org: -exp, left: +exp,   right: +exp
    // already ok : do nothing
  }
else
  {
    // org -exp, left: +exp,   right: -exp
    a = left;
    b = right->left;
    l = b;
    r = right;
    r->type = nt_plus;
    l->left = a;
    left = r;
    right = l;
  }
      }
    else
      {
if (right->type == nt_number || right->type == nt_var || right->type == nt_plus)
```

```
  {
    // org: -exp, left: -exp,   right: +exp
    a = left->left;
    c = right;
    l = left;
    l->type = nt_plus;
    l->left = c;
    r = a;
    left = r;
    right = l;
  }
else
  {
    // org -exp, left: -exp,   right: -exp
    b = left->right;
    d = right->right;
    l = left;
    l->type = nt_plus;
    l->right = d;
    r = right;
    r->type = nt_plus;
    r->right = b;
  }
      }
    break;
  }
}


/*************************************************
 *          print functions                      *
 *************************************************/

void numberexp::print() {
  switch (type) {
  case nt_number:
    cout << number;
    break;
  case nt_plus:
    cout << "(";
    left->print();
    cout << " + ";
    right->print();
    cout << ")";
    break;
  case nt_minus:
    cout << "(";
```

```
    left->print();
    cout << " - ";
    right->print();
    cout << ")";
    break;
  case nt_var:
    cout << var;
    break;
  }
}


void numberprop::print() {
  left->print();
  switch (op) {
  case ro_lt:
    cout << " < ";
    break;
  case ro_gt:
    cout << " > ";
    break;
  case ro_eq:
    cout << " = ";
    break;
  case ro_ne:
    cout << " <> ";
    break;
  }
  right->print();
}



void atom::print() {
  switch (type) {
  case at_numprop:
    cout << "[";
    prop->print();
    cout << "]";
    break;
  case at_boolvar:
    cout << var;
    break;
  }
}
```

```
void formula::print() {
  switch (type) {
  case ft_atom:
    atomic->print();
    break;
  case ft_neg:
    cout << "~";
    cout << "{";
    f1->print();
    cout << "}";
    break;
  case ft_and:
    cout << "{";
    f1->print();
    cout << " /\\ ";
    f2->print();
    cout << "}";
    break;
  case ft_or:
    cout << "{";
    f1->print();
    cout << " \\/ ";
    f2->print();
    cout << "}";
    break;
  case ft_impl:
    cout << "{";
    f1->print();
    cout << " => ";
    f2->print();
    cout << "}";
    break;
  case ft_biimpl:
    cout << "{";
    f1->print();
    cout << " <=> ";
    f2->print();
    cout << "}";
    break;
  case ft_ite:
    cout << "{";
    f1->print();
    cout << " -> ";
    f2->print();
    cout << " , ";
    f3->print();
```

```
    cout << "}";
    break;
  case ft_true:
    cout << "true";
    break;
  case ft_false:
    cout << "false";
    break;
  case ft_paren:
    cout << "(";
    f1->print();
    cout << ")";
    break;

  }
}

void idlst::print() {
  if (this)
    {
      cout << id;
      if (next) cout << ", ";
      next->print();
    }
}

void action::print() {

  if (this)
    {
      cout << "      " << name << "\n";
      cout << "        var: ";
      var->print();
      cout << "\n        pre: ";
      pre->print();
      cout << "\n        eff: ";
      eff->print();
      if (next)
{
  cout << "\n\n";
         next->print();
}
    }
}


void agent::print() {
```

```
  if (this)
    {
      cout << "    agt: " << name << "\n\n";
      actions->print();
      if (next)
{
  cout << "\n\n";
  next->print();
}
    }
}


void vartype::print() {
  if (type == vt_bool)
    cout << "    bool  ";
  else
    cout << "    scalar(1.." << range << ")  ";
}




void varlst::print() {
  if (this)
    {
      type->print();
      ids->print();
      cout << "\n";
      next->print();
    }
}



void mnpproblem::print() {
  cout << "\nvariables\n\n";
  vars->print();
  cout << "\nsystem\n\n";
  sys->print();
  cout << "\n\nenvironment\n\n";
  env->print();
  cout << "\n\ninit\n";
  init->print();
  cout << "\n\ngoal\n";
  goal->print();
  cout << "\n";
```

```
}
```

D.5.5 Fsm.cc

```
/*****************************************************
 * File  : fsm.cc
 * Desc. : misc. functions for generating the OBDD FSM
 *         transition relation.
 * Author: Rune M. Jensen
 * Date  : 3/4/99
 *****************************************************/

#include <stream.h>
#include <string.h>
#include <math.h>
#include <limits.h>
#include <bdd.h>
#include <fstream.h>
#include "common.hpp"
#include "fsm.hpp"
#include "bddprint.hpp"
#include "dissets.hpp"



/*****************************************************
 *              print functions                      *
 *****************************************************/
void agtdef::print() {
  if (this)
    {
      cout << name << "(" << number << ",";
      if (sclass == sc_env)
cout << "env): ";
      else
cout << "sys): ";
      acts->print();
      cout << "\n";
      next->print();
    }
}

// used by bddprint.cc
void agtdef::print(ofstream &out,int *actlen) {
  if (this)
    {
      out << name << "(" << number << ",";
      if (sclass == sc_env)
out << "env): ";
      else
```

```
out << "sys): ";
      actlen[number] = strlen(name) + 2;
      acts->print(out,&actlen[number]);
      out << "\n";
      next->print(out,actlen);
    }
}

void actdef::print() {
  if (this)
    {
      cout << name << "(" << number << ") ";
      next->print();
    }
}

//used by bddprint.cc
void actdef::print(ofstream &out,int *actlen) {
  if (this)
    {
      out << name << "(" << number << ") ";
      if (strlen(name) + 2 > *actlen) *actlen = strlen(name) + 2;
      next->print(out,actlen);
    }
}

void vardef::print() {
  int i;

  if (this)
    {
      cout << "varname: " << name << "\n";
      cout << "type: ";
      switch (type) {
        case vt_bool:
     cout << "bool\n";
  break;
        case vt_scalar:
  cout << "scalar\n";
  break;
      }
      cout << "length: " << length << "\nmapping: ";
      for (i=0; i<length; i++)
cout << i+1 << ":" << var[i] << " ";
      cout << "\n\n";
      next->print();
    }
```

```
}


void domdef::print() {
  if (this)
    {
      cout << "\n\nDomain Definition\n";
      cout << "Agtdef\n\n";
      agt->print();
      cout << "Vardef\n\n";
      var->print();
      cout << "\n\nagtnum: " << agtnum << "\n\n";
      cout << "\n varnames : ";
      varnames->print();
      cout << "\n\n";
    }
}


/*****************************************************
 * Access member functions                           *
 *****************************************************/

vardef *vardef::lookup(char *nm) {
  if (this)
    {
      if (!strcmp(name,nm))
return this;
      else
return next->lookup(nm);
    }
  else
    return NULL;
}

vardef *vardef::lookup(int i) {
  char name[MAXNAMELENGTH];

  sprintf(name,"%d",i);
  return this->lookup(name);
}

vardef *vardef::lookupnew(char *name) {
  char namem[MAXNAMELENGTH];

  sprintf(namem,"%s'",name);
  return this->lookup(namem);
```

```
}



agtdef *agtdef::lookup(int i) {
  if (this) {
    if (number == i)
      return this;
    else
      return next->lookup(i);
  }
  else
    return NULL;
}


/*****************************************************
 * Dynamic BDD var autoreorder definition             *
 *****************************************************/
// IN
//  def : domain definition
// OUT (side effect)
//  A BDD block definition for each action and variable.
//  action varibles and current and next state varibles
//  are assumed to be defined on a continuous block of
//  variables
void mkbddblocks(domdef *def) {
  vardef *vard,*act,*s,*sm;
  int    i,j,start,end;
  idlst  *id;

  vard = def->var;

  // clear some old block definition
  bdd_clrvarblocks();

  // add blocks for action variables
  for (i=0; i<def->agtnum; i++)
    {
      act = vard->lookup(i);

      // find the start and end of the action block
      start = def->bddvarnum;
      end = 0;
      for (j=0; j<act->length; j++)
{
  if (act->var[j] < start) start = act->var[j];
```

```
  if (act->var[j] > end)   end = act->var[j];
}

      // add variable block for dynamic bdd var reordering
      if (bdd_intaddvarblock(start,end,BDD_REORDER_FIXED) != 0)
{
  cout << "fsm.cc mkbddblocks : cannot make var. block ";
  cout << start << "-" << end << "\nexiting\n\n";
  exit(1);
}
    }


  // add blocks for variables
  id = def->varnames;
  while (id)
    {
      s = vard->lookup(id->id);
      sm = vard->lookupnew(id->id);

      // find the start and end of the variable block
      start = def->bddvarnum;
      end = 0;
      for (j=0; j<s->length; j++)
{
  if (s->var[j] < start) start = s->var[j];
  if (s->var[j] > end)   end = s->var[j];
}

      for (j=0; j<s->length; j++)
{
  if (sm->var[j] < start) start = sm->var[j];
  if (sm->var[j] > end)   end = sm->var[j];
}

      // add variable block for dynamic bdd var reordering
      if (bdd_intaddvarblock(start,end,BDD_REORDER_FIXED) != 0)
{
  cout << "fsm.cc mkbddblocks : cannot make var. block ";
  cout << start << "-" << end << "\nexiting\n\n";
  exit(1);
}

      id = id->next;
    }
}
```

```
/*************************************************
 *  Function building a domain definition          *
 *************************************************/

domdef *mkdomdef(mnpproblem *mnp) {

  agtdef     *nextagt, *agtd;
  agent      *agt;
  int         agtnum,maxacts,actnum,length,bvcount;
  int         i,j,envnum;
  int        *var;
  action     *act, ***acttbl;
  actdef     *nextact, *actd;
  int        *actnumtbl;
  vardef     *nextvar;
  varlst     *vlp;
  idlst      *ilp, *nextvarname;
  char       *str;

  // clustering variables
  disset *dset;
  disnode *set1,*set2;




  //************** build the agtdef  **************
  agtnum  = 0;
  envnum  = 0;
  maxacts = 0;
  agt = mnp->env;
  nextagt = NULL;


  // add environment agents
  while (agt)
    {
      nextact = NULL;
      actnum   = 0;
      act = agt->actions;
      while (act)
{
  nextact = new actdef(act->name,actnum++,act,nextact);
  act = act->next;
}
```

```
      // post: nextact is a pointer to all the actions of agt
      nextagt = new agtdef(agt->name,agtnum++,actnum,sc_env,nextact,nextagt);
      if (actnum > maxacts) maxacts = actnum;

      agt = agt->next;
    }

  envnum = agtnum;
  agt = mnp->sys;

  // add system agents
  while (agt)
    {
      nextact = NULL;
      actnum   = 0;
      act = agt->actions;
      while (act)
{
  nextact = new actdef(act->name,actnum++,act,nextact);
  act = act->next;
}
      // post: nextact is a pointer to all the actions of agt
      nextagt = new agtdef(agt->name,agtnum++,actnum,sc_sys,nextact,nextagt);
      if (actnum > maxacts) maxacts = actnum;

      agt = agt->next;
    }


  // post: nextagt is a pointer to the agt definition
  //       agtnum: the total number of agents
  //       maxacts: the maximal number of actions


  //**** build the act table and action number tabel ****
  acttbl = new action**[agtnum];
  for (i=0; i<agtnum; i++)
    acttbl[i] = new action*[maxacts];
  actnumtbl = new int[agtnum];
  agtd = nextagt;
  while (agtd)
    {
      actd = agtd->acts;
      actnumtbl[agtd->number] = agtd->actnum;
      while (actd)
{
  acttbl[agtd->number][actd->number] = actd->act;
```

```
  actd = actd->next;
}
      agtd = agtd->next;
    }



  //************** build the vardef ******************
  // Interleaved Ordering of S E and S' E':
  //
  // A1,A2,...,Am,S11,S11',...,S1nS'1n, S21,S21',... ,S2m,S2m' ...
  //
  //************************************************

  bvcount = 0;   // first bdd variable number
  nextvar = NULL;

  // start with the action identifier variables
  for (i=0;i < agtnum;i++)
    {
      if (actnumtbl[i] == 1)
length = 1; // redundant, but we want to keep all information in the BDD
      else
length = int(ceil(log(actnumtbl[i])/log(2)));

      // make mapping
      var = new int[length];
      for (j=0;j<length; j++)
var[j] = bvcount + j;

      // save description
      str = new char[MAXNAMELENGTH];
      sprintf(str,"%d",i);
      nextvar = new vardef(str,vt_scalar,length,
   var,nextvar);
      bvcount += length;
    }


  // continue with the variables S11,S11',...,S1nS'1n, S21,S21',... ,S2m,S2m'
  // (and save the var names in a list)
  nextvarname = NULL;
  vlp = mnp->vars;
  while (vlp)
    {
      // find the bdd variable length of variable type
      if (vlp->type->type == vt_bool)
```

```
length = 1;
      else
length = vlp->type->range;

      // assign bdd variable space for each variable
      ilp = vlp->ids;
      while (ilp)
{
  // S

  // make mapping to bddvars
  var = new int[length];
  for (i=0; i<length; i++)
    var[i] = bvcount + 2*i;

  // save desription
  nextvar = new vardef(ilp->id,vlp->type->type,length,
  var,nextvar);
  // add variable to variable list
         nextvarname = new idlst(ilp->id,nextvarname);

  // S'

  // make mapping to bddvars
  var = new int[length];
  for (i=0; i<length; i++)
    var[i] = bvcount + 2*i + 1;

  // save desription
  str = new char[MAXNAMELENGTH];
  sprintf(str,"%s'",ilp->id);
  nextvar = new vardef(str,vlp->type->type,length,
       var,nextvar);

  bvcount += 2*length;

  ilp = ilp->next;
}

      // look at next variable type
      vlp = vlp->next;
    }


  //************** find clusters    ******************

  // make initial disjoint set
```

```
  // (a singleton set for each variable)


   dset = new disset();
   ilp = nextvarname;
   while (ilp)
    {
      dset->makeset(ilp->id);
      ilp = ilp->next;
    }


  // Gripper2 Experiment compressing the init se
   ilp = nextvarname;
   i = 0;
   while (ilp)
    {
      if (i == 0)
set1 = dset->find(ilp->id);
      else
{
  set2 = dset->find(ilp->id);
  dset->setunion(set1,set2);
}
      i++;
      if (i == 5) i=0; // compress rate
      ilp = ilp->next;
    }



  // gripper2 Experiment Monolithic transition relation
   /*
  dset = new disset();
  ilp = nextvarname;
  dset->makeset(ilp->id);
  set1 = dset->find(ilp->id);
  ilp = ilp->next;
  while (ilp)
    {
      dset->makeset(ilp->id);
      set2 = dset->find(ilp->id);
      dset->setunion(set1,set2);
      ilp = ilp->next;
    }
    */
```

```
  // find disjoint set for variable clusters
  for (i=0; i < agtnum; i++)
    for (j=0; j < actnumtbl[i]; j++)
      {
        ilp = acttbl[i][j]->var;

// initial set1
        if (ilp)
  {
    set1 = dset->find(ilp->id);
    ilp = ilp->next;
  }


// union subsets of each variable in the variable list
while (ilp)
  {
    set2 = dset->find(ilp->id);
    if (set1 != set2)
      set1 = dset->setunion(set1,set2);
    ilp = ilp->next;
  }

      }

  // enumerate clusters
  dset->enumerate();

  // make partition table
  dset->mkpartvars();

  cout << "\nNumber of clusters " << dset->subsetnum << "\n";

  // build output
  return new domdef(nextagt,envnum,agtnum,actnumtbl,
    acttbl,nextvar,nextvarname,bvcount,dset);
}




/*****************************************************
 * Function defining bdd variable loactions          *
 *****************************************************/

// IN
//  def  : domain definition. (includes description
```

```
//         of bdd variable locations.
// OUT
//  instantiation of BuDDy variable location desriptions
//  and variable relations.
varrels *mkvarrels(domdef *def) {

  int     i,j,k,p,*varset;
  idlst   *vars;
  char    *id;
  vardef *oldv, *newv, *actv, *vard;
  varrels *rels;

  rels = new varrels;

  rels->old2new = bdd_newpair();
  rels->new2old = bdd_newpair();
  vard = def->var;

  // relate new/old and old/new state variables
  vars = def->varnames;
  while (vars)
    {
      oldv = vard->lookup(vars->id);
      newv = vard->lookupnew(vars->id);
      for (i=0; i < newv->length; i++)
{
    bdd_setpair(rels->old2new,oldv->var[i],newv->var[i]);
    bdd_setpair(rels->new2old,newv->var[i],oldv->var[i]);
}
      vars = vars->next;
    }

  varset = new int[def->bddvarnum];
  // define old state variables
  vars = def->varnames;
  j = 0;
  while (vars)
    {
      oldv = vard->lookup(vars->id);
      for (i=0; i < oldv->length; i++)
varset[j++] = oldv->var[i];
      vars = vars->next;
    }
  rels->oldstate = bdd_makeset(varset,j);

  rels->newstate = new bdd[def->dset->subsetnum];
  // define new state variables
```

```
  // split up on partitions
  for (p=0; p < def->dset->subsetnum; p++)
    {
      vars = def->dset->partvars[p];

      j = 0;
      while (vars)
{
  newv = vard->lookupnew(vars->id);
  for (i=0 ; i < newv->length; i++)
    varset[j++] = newv->var[i];
  vars = vars->next;
}
      rels->newstate[p] = bdd_makeset(varset,j);
    }


  // define environment action variables
  j = 0;
  for (i=0; i<def->envnum; i++)
    {
      actv = vard->lookup(i);
      for (k=0; k < actv->length; k++)
varset[j++] = actv->var[k];
    }
  rels->envactions = bdd_makeset(varset,j);

  // define system action variables
  j = 0;
  for (i=def->envnum; i<def->agtnum; i++)
    {
      actv = vard->lookup(i);
      for (k=0; k < actv->length; k++)
varset[j++] = actv->var[k];
    }
  rels->sysactions = bdd_makeset(varset,j);

return rels;
}



/*****************************************************
 * Function adjusting domdef to a variable ordering   *
 *****************************************************/

// IN
```

```
//  order : Array of variable levels in ordering
//  def   : domain definition
// OUT
//  sideeffect: adjusted domdef
void adjust2order(int *ordering, domdef *def) {
  vardef *vard;
  int i;

  vard = def->var;

  // go through all domain variables and map the
  // variable locations to the new ordering
  while (vard)
    {
      for (i=0; i<vard->length; i++)
vard->var[i] = ordering[vard->var[i]];
      vard = vard->next;
    }
}
```

```
/*************************************************
 * Functions for generating OBDD's from formulas      *
 *************************************************/

bdd formula2bdd(formula *f, domdef *def) {

bdd b,l,r,e;

  switch (f->type) {
  case ft_atom:
    b = atom2bdd(f->atomic, def);
    break;

  case ft_neg:
    b = formula2bdd(f->f1, def);
    b = !b;
    break;

  case ft_and:
    l = formula2bdd(f->f1,def);
    r = formula2bdd(f->f2,def);
```

```
    b = l & r;
    break;

  case ft_or:
    l = formula2bdd(f->f1,def);
    r = formula2bdd(f->f2,def);
    b = l | r;
    break;

  case ft_impl:
    l = formula2bdd(f->f1,def);
    r = formula2bdd(f->f2,def);
    b = l >> r;
    break;

  case ft_biimpl:
    l = formula2bdd(f->f1,def);
    r = formula2bdd(f->f2,def);
    b = (l >> r) & (l << r);
    break;

  case ft_ite:
    e = formula2bdd(f->f1,def);
    l = formula2bdd(f->f2,def);
    r = formula2bdd(f->f3,def);
    b = (e & l) | (!e & r);
    break;

  case ft_true:
    b = bddtrue;
    break;

  case ft_false:
    b = bddfalse;
    break;

  case ft_paren:
    b = formula2bdd(f->f1,def);
    break;
  }
return b;
}



bdd atom2bdd(atom *a, domdef *def) {
```

```
  bdd b;
  vardef *vd;

  switch (a->type) {
    case at_numprop:
      a->prop->minus2plus();
      b = numprop2bdd(a->prop,def);
      break;
    case at_boolvar:
      vd  = def->var->lookup(a->var);
      if (vd)
b = bdd_ithvar(vd->var[0]);
      else
{
  cout << "\nfsm.hpp, atom2bdd: variable " << a->var;
  cout << " is not defined\nexiting\n";
  exit(1);
}
      break;
  }
  return b;
}


//** Functions for generating BDD's from numprops  ***

bdd numprop2bdd(numberprop *prop,domdef *def) {
  bdd b,e,ld,rd;
  binary *l, *r;
  numberexp *c;
  int i,j,digitnum;

  switch (prop->op) {
  case ro_lt:
    c = prop->left;
    prop->left = prop->right;
    prop->right = c;
    prop->op = ro_gt;
    b = numprop2bdd(prop,def);
    break;
  case ro_ne:
    prop->op = ro_eq;
    b = !numprop2bdd(prop,def);
    break;
  case ro_eq:
    l = numexp2binary(prop->left,def);
    r = numexp2binary(prop->right,def);
```

```
    b = bddtrue;

    // test if l = r
    i = 0;
    while (i < l->digitnum || i < r->digitnum)
      {
if (i < l->digitnum && i < r->digitnum)
  {
    b &= l->digit[i] >> r->digit[i];
    b &= l->digit[i] << r->digit[i];
  }
else
  if ( i < l->digitnum )
    {
      // r->digit[i] = bddfalse
      b &= !l->digit[i];
    }
  else
    {
      // l->digit[i] = bddfalse
      b &= !r->digit[i];
    }
i++;
      }
    break;

  case ro_gt:
    l = numexp2binary(prop->left,def);
    r = numexp2binary(prop->right,def);

    digitnum = l->digitnum > r->digitnum ? l->digitnum : r->digitnum;

    b = bddfalse;
    for (i=0; i<digitnum; i++)
      {
// all digits before digit_i must be equal
e = bddtrue;
for (j = i+1; j<digitnum; j++)
  {
    ld = j >= l->digitnum ? bddfalse : l->digit[j];
    rd = j >= r->digitnum ? bddfalse : r->digit[j];
    e &= ld >> rd;
    e &= ld << rd;
  }

ld = i >= l->digitnum ? bddfalse : l->digit[i];
```

```
rd = i >= r->digitnum ? bddfalse : r->digit[i];

// if e is true then if ld > rd: l > r
b |= e & ld & !rd;
      }
    break;
  }
  return b;
}


binary *numexp2binary(numberexp *num,domdef *def) {
  binary *b,*l,*r;
  bdd    *digit,c;
  int     digitnum,i,number;
  vardef *vard;

  switch (num->type) {
  case nt_minus:
    cout << "\n\nfsm.cc numexp2binary: error type of numexp cannot be nt_minus\nexiting\n";
    exit(1);
    break;

  case nt_number:
    number = num->number;
    // the digitnum of 0 is 0, of 1 is 1, of 2,3 is 2, of 4,5,6,7 is 3 etc.
    digitnum = int(ceil(log(number+1)/log(2)));

    if (digitnum > 0)
      digit = new bdd[digitnum];
    else
      digit = NULL;

    // initialize digit array
    for (i=0; i<digitnum; i++)
      {
if (number % 2)
  digit[i] = bddtrue;
else
  digit[i] = bddfalse;
number /= 2;
      }

    b = new binary(digitnum,digit);
    break;

  case nt_var:
```

```
    vard = def->var->lookup(num->var);
    if (vard == NULL)
{
  cout << "\nfsm.hpp numexp2binary: variable " << num->var;
  cout << " is not defined\nexiting\n";
  exit(1);
}
    else
      {
digitnum = vard->length;

digit = new bdd[digitnum];

//intialize the digit array
for (i=0; i < digitnum; i++)
  digit[i] = bdd_ithvar(vard->var[digitnum - 1 - i]);

b = new binary(digitnum,digit);
      }
    break;

  case nt_plus:
    l = numexp2binary(num->left,def);
    r = numexp2binary(num->right,def);

    // define result binary
    digitnum = l->digitnum > r->digitnum ? l->digitnum + 1 : r->digitnum + 1;
    digit = new bdd[digitnum];

    // set initial carry
    c = bddfalse;
    i = 0;

    // add the binaries
    while ( i < l->digitnum || i < r->digitnum )
      {
if ( i < l->digitnum && i < r->digitnum )
  {
    digit[i] = l->digit[i] ^ r->digit[i] ^ c;
    c = l->digit[i] & r->digit[i] |
        l->digit[i] & c |
        r->digit[i] & c;
  }
else
  if ( i < l->digitnum )
    {
      //r->digit[i] = bddfalse
```

```
      digit[i] = l->digit[i] ^ c;
      c = l->digit[i] & c;
    }
  else
    {
      //l->digit[i] = bddfalse
      digit[i] = r->digit[i] ^ c;
      c = r->digit[i] & c;
    }
i++;
      }

    // post: i = last digit in result (corresponds to carry)
    digit[i] = c;

    b = new binary(digitnum,digit);
    break;
  }
  return b;
}




/*****************************************************
 * Functions for generating the transition relation   *
 *****************************************************/

// IN
//   v1: first variable in the binary encoding (most sig).
//   v2: last variable in the binary encoding.
//   n : integer to encode.
//   OUT
//   bdd : BDD with the binary encoding of n
//         starting with the most significant bit.
bdd int2bdd(int *var,int length,int n)  {
  bdd res;
  int j;

  res = bddtrue;
  for (j=length-1; j >= 0; j--)
    {
      if (n % 2)
res &= bdd_ithvar(var[j]);
      else
res &= bdd_nithvar(var[j]);
      n /= 2;
```

```
    }
return res;
}



// I(i,j,def)
// IN
//  i,j : Action j of agent i.
//  def : domain definition.
//
// OUT
//  the bdd identifier expression
//  for action(i,j).
bdd I(int i, int j, domdef *def) {
  vardef *vd;

  vd = def->var->lookup(i);
  return int2bdd(vd->var,vd->length,j);
}



// F(v,i,j,def)
// IN
//  v   : variable to check
//  i,j : Action j of agent i.
//  def : Domain definition
//
// OUT
//  bddtrue : var is unconstrained by action(i,j).
//  bddfalse: var is constrained by action(i,j)
//            (var is in the action's varlist.)
bdd F(char *v, int i, int j, domdef *def) {
if (def->act[i][j]->var->lookup(v))
  return I(i,j,def) >> bddfalse;
else
  return I(i,j,def) >> bddtrue;
}



//***** Functions for constraining the action tuple *********

// ADEF(def)
// IN
```

```
//  def : domain definition.
// OUT
//  bdd constraining possible actions to only defined actions
bdd ADEF(domdef *def) {
  bdd res;
  int i;
  formula *f;
  atom *a;
  numberprop *p;
  numberexp  *l,*r;
  char str[MAXNAMELENGTH];

  f = new formula;
  a = new atom;
  p = new numberprop;
  l = new numberexp;
  r = new numberexp;

  // define formula : "actid" < max_val+1
  f->type = ft_atom;
  f->atomic = a;
  a->type = at_numprop;
  a->prop = p;
  p->op = ro_lt;
  p->left = l;
  p->right = r;
  l->type = nt_var;
  r->type = nt_number;

  // constrain action tuple to only defined actions
  res = bddtrue;
  for (i=0; i<def->agtnum; i++)
    {
      sprintf(str,"%d",i);
      l->var = str;
      r->number = def->actnum[i];
      res &= formula2bdd(f,def);
    }

  return res;
}




// IND(i1,j1,i2,j2,def)
// IN
```

```
//  i1,j1 : action identifier 1
//  i2,j2 : action identifier 2
//  def : domain definition.
// OUT
//  bdd expressing that A1
//  is consistent with A2
//  (no overlap of constained variables)
bdd IND(int i1, int j1, int i2, int j2, domdef *def) {

  idlst *var1,*var2;

  var1 = def->act[i1][j1]->var;

  while (var1)
    {
      var2 = def->act[i2][j2]->var;
      while (var2)
{
  if (!strcmp(var1->id,var2->id))
    return bddfalse;
  var2 = var2->next;
}
      var1 = var1->next;
    }
  return bddtrue;
}


// AC(def)
// IN
//  def : domain definition.
// OUT
//  bdd expressing the total independency and definition constraint
//  on action tuples
bdd AC(domdef *def) {
  bdd res,l,r;
  int i1, i2, j1, j2;

  res = ADEF(def);

  // concurrent independency constraints

  // add internal independency condition
  // environment agents
  // go through all possible non-symmetric action pairs
  for (i1=0; i1 < def->envnum - 1; i1++)
    for (i2=i1+1; i2 < def->envnum; i2++)
```

```
      for (j1=0; j1 < def->actnum[i1]; j1++)
   for (j2=0; j2 < def->actnum[i2]; j2++)
  res &= ( I(i1,j1,def) & I(i2,j2,def) ) >>
    IND(i1,j1,i2,j2,def);

  // system agents
  // go through all possible non-symmetric action pairs
  for (i1=def->envnum; i1 < def->agtnum - 1; i1++)
    for (i2=i1+1; i2 < def->agtnum; i2++)
      for (j1=0; j1 < def->actnum[i1]; j1++)
   for (j2=0; j2 < def->actnum[i2]; j2++)
  res &= ( I(i1,j1,def) & I(i2,j2,def) ) >>
    IND(i1,j1,i2,j2,def);

  return res;
}



// P(i,j,def)
// IN
//  i,j : Action j of agent i.
//  def : domain definition.
//
// OUT
//  bdd expression for the precondition
//  of action(i,j).
bdd P(int i, int j, domdef *def) {
  return formula2bdd(def->act[i][j]->pre,def);
}

// E(i,j,def)
// IN
//  i,j : Action j of agent i.
//  def : domain definition.
//
// OUT
//  bdd expression for the effect
//  of action(i,j).
bdd E(int i, int j, domdef *def) {
  return formula2bdd(def->act[i][j]->eff,def);
}


// IV(partno,def)
// IN
```

```
//  def   : domain definition.
//  partno: partition no of IV
//
// OUT
//  bdd expression for the invariable expression
//  constraining only new vars in the partition
// IV(i) = /\v in part(i) ((/\i,j F(i,j,v)) => v' = v)
bdd IV(int partno,domdef *def) {
  int i,j;
  idlst *var;
  bdd    l,r,res;
  vardef *s,*sm;

  res = bddtrue;
  var = def->dset->partvars[partno];
  while (var)
    {
      // make left side
      l = bddtrue;
      for (i=0; i < def->agtnum; i++)
for (j=0; j < def->actnum[i]; j++)
  l &= F(var->id,i,j,def);

      // make right side
      s = def->var->lookup(var->id);
      sm = def->var->lookupnew(var->id);
      r = bddtrue;
      for (i=0; i < s->length; i++)
{
  r &= bdd_ithvar(s->var[i]) >> bdd_ithvar(sm->var[i]);
  r &= bdd_ithvar(s->var[i]) << bdd_ithvar(sm->var[i]);
}

      // add to result
      res &= l >> r;

      if (DEBUG)
cout << "Finished var: " << var->id << "\n";

      var = var->next;
    }
return res;
}


// IN
//  i : agentt number
```

```
//  j : action number
//  def : domain definition
// OUT
//  bdd representing the action
//  A =  I(i,j) => ~I1 /\ ... ~In /\ P(i,j) /\ E(i,j)
bdd A(int i,int j,domdef *def) {
  bdd l,r;

  l = bddtrue;
  r = bddtrue;

  // make left side of impl.
  l &= I(i,j,def);

  // add relation constraint
  r &= P(i,j,def) & E(i,j,def);

  return l >> r;
}


// T(def)
// IN
//  def : domain definition.
// OUT
//  bdd transition relation comprised of T*IV pairs
//  for each partition.
bddtrel *T(domdef *def) {
  int i,j,partno,subsetnum;
  bddtrel *tr;
  bdd env_vars;
  vardef *var;


  // init partition structure
  subsetnum = def->dset->subsetnum;
  tr = new bddtrel(subsetnum);
  for (i=0; i<subsetnum; i++)
    {
      tr->T[i] =  bddtrue;
      tr->IV[i] = bddtrue;
    }

  // instantiate T part of partition structure
  for (i=0; i < def->agtnum; i++)
    for (j=0; j< def->actnum[i]; j++)
      {
```

```
// find the partition number of the action
// lookup the first variable or else set partition number to 0
if ( def->act[i][j]->var )
  partno = def->dset->find(def->act[i][j]->var->id)->setnum;
else
  partno = 0;

      // add the action to the partition
      tr->T[partno] &= A(i,j,def);

      if (DEBUG)
  cout << "Finished action(" << i << "," << j << ")\n";
    }

  // instantiate IV part of partition structure
  for (i=0; i<subsetnum; i++)
    tr->IV[i] = IV(i,def);

  // add action value condition
  tr->A = AC(def);

  return tr;
}
```

D.5.6 Main.cc

```
/*****************************************************
 * File  : main.cc
 * Desc. : main file for the MNP planner
 * Author: Rune M. Jensen
 * Date  : 4/4/99
 *****************************************************/
#include <stream.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <bdd.h>
#include "common.hpp"
#include "domain.hpp"
#include "fsm.hpp"
#include "bddprint.hpp"
#include "plan.hpp"
#include "analyse.hpp"
#include "reorder.hpp"
#include "time.hpp"
#include "main.hpp"




int main(int argc,char **argv)  {
  extern FILE *yyin;
  domdef *def;
  int wellformed;

  // planning variables
  bddtrel *t;
  bdd tout;
  bdd sa,init,goal;
  int i;
  bddprintabs *tb;
  char buf[16];
  char orderoutfile[MAXNAMELENGTH];
  char orderinfile[MAXNAMELENGTH];
  int *ordering;



  // check input format
  wellformed = 1;
  if (!strcmp(argv[1],"-plan1") ||
```

```
      !strcmp(argv[1],"-plan2") ||
      !strcmp(argv[1],"-plan3") ||
      !strcmp(argv[1],"-plan4"))
    {
      if (argc != 5) wellformed = 0;
    }
  else
    if (!strcmp(argv[1],"-analyse") ||
!strcmp(argv[1],"-plan5") )
      {
if (argc != 6) wellformed = 0;
      }
    else
      wellformed = 0;

  if (!wellformed)
    {
      cout << "Usage: mnp -plan1   <initnodenum> <domain>.mnp [<plan>.sat <SA.bdd>] \n";
      cout << "       mnp -plan2   <initnodenum> <domain>.mnp <SA>.bdd\n";
      cout << "       mnp -plan3   <initnodenum> <domain>.mnp <SA>.bdd\n";
      cout << "       mnp -plan4   <initnodenum> <domain>.mnp <SA>.bdd\n";
      cout << "       mnp -plan5   <initnodenum> <domain>.mnp <SA>.bdd <plan>.sat\n";
      cout << "       mnp -analyse <initnodenum> <domain>.mnp <SA>.bdd <result>.sat\n\n";
      cout << "plan1: Deterministic algorithm, writes resulting plan to \
               <plan>.sat or SA to <SA>.bdd\n";
      cout << "plan2: Nondeterministic algorithm: IRST StrongPlan\n";
      cout << "plan3: Nondeterministic algorithm: IRST StrongCyclicPlan\n";
      cout << "plan4: Nondeterministic algorithm: Optimistic Plan\n";
      cout << "plan5: find one plan for problem in <domain>.mnp but based\n";
      cout << "       on <SA>.bdd write to <plan>.sat.\n";
      exit(1);
    }



  // parse mnp domain
  yyin = fopen(argv[3],"r");
  if (yyin == NULL)
    {
      cout << "mnp: Cannot open \"" << argv[3] << "\"\n";
      exit(1);
    }


  if (yyparse()) exit(1); // exit on syntax error
```

```
  // make domain info. struct
  def = mkdomdef(mnpprob);


  // Select mnp activity
  if ( !strcmp(argv[1],"-analyse"))
    // analyse existing SA
    bddanalyse(argv[4],argv[5],atoi(argv[2]),def);
  else
    if ( !strcmp(argv[1],"-plan1") ||
 !strcmp(argv[1],"-plan2") ||
 !strcmp(argv[1],"-plan3") ||
 !strcmp(argv[1],"-plan4") ||
 !strcmp(argv[1],"-plan5")  )
      {
// Some planning activity

// initialize bdd package
bdd_init(atoi(argv[2]),10000);
bdd_setvarnum(def->bddvarnum);

// add bdd structures defining bdd variable locations
def->rels = mkvarrels(def);

// enable automatic bdd variable reordering
mkbddblocks(def);
bdd_autoreorder(BDD_REORDER_WIN2ITE);

// make partitioned transition relation
t = T(def);

// This part is used  to force a monolithic transistion relation
//t->T[0] &= t->A;
//for (i=1; i<def->dset->subsetnum; i++)
//  {
//    t->T[0] &= t->T[i];
//    T[i] = bddtrue;
//    t->IV[0] &= t->IV[i];
       //    IV[i] = bddtrue;
//  }
//tout = bdd_exist(tout,def->rels->envactions);
//tb = new bddprintabs("t.sat",def);
//tb->bddprint(tout);


// debug info
if (DEBUG)
```

```
  cout << "\nfinished FSM\n";

// assign planning parameters
sa = bddfalse;
init = formula2bdd(mnpprob->init,def);
goal = formula2bdd(mnpprob->goal,def);


// ****** select planning algorithm *******
if (!strcmp(argv[1],"-plan1"))
  {

    // start timing
    startwatch();


    // make nondeterministic plan
    if (plan1(t,init,goal,&sa,def))
cout << "SA covers Init\n";
      else
cout << "SA do not cover Init\n";

    // stop timing
    stopwatch();



    // write bdd output
    cout << "Final node count of SA: " << bdd_nodecount(sa) << "\n";
    cout << "Write SA or one plan to output file \"" << argv[4]
                << "\" or nothing (s/p/n): ";
    gets(buf);
    //buf[0] = 'p';

    if (buf[0] == 's')
      {
if (bdd_fnsave(argv[4],sa))
  {
    cout << "mnp: Cannot save bdd in file \"" << argv[4] << "\"\nexiting\n";
    exit(1);
  }

// write order output
sprintf(orderoutfile,"%so",argv[4]);
writeorder(orderoutfile,def);
      }
    else
```

```
      if (buf[0] == 'p')
// write one plan to file
printplan(t,init,goal,sa,argv[4],def);

    // close bdd package
    bdd_done();
  }
else
  if (!strcmp(argv[1],"-plan2"))
    {
      if (strongplan(t,init,goal,&sa,def))
cout << "SA covers Init\n";
      else
cout << "SA do not cover Init\n";


      // write bdd output
      cout << "Final node count of SA: " << bdd_nodecount(sa) << "\n";
      cout << "Write SA to output file \"" << argv[4] << "\" (y/n): ";
      gets(buf);
      if (buf[0] == 'y')
if (bdd_fnsave(argv[4],sa))
  {
    cout << "mnp: Cannot save bdd in file \"" << argv[4] << "\"\nexiting\n";
      exit(1);
  }

      // write order output
      sprintf(orderoutfile,"%so",argv[4]);
writeorder(orderoutfile,def);

// close bdd package
bdd_done();
    }
  else
    if (!strcmp(argv[1],"-plan3"))
      {
if (strongcyclicplan(t,init,goal,&sa,def))
  cout << "SA covers Init\n";
else
  cout << "SA do not cover Init\n";


// write bdd output
cout << "Final node count of SA: " << bdd_nodecount(sa) << "\n";
cout << "Write SA to output file \"" << argv[4] << "\" (y/n): ";
gets(buf);
```

```
if (buf[0] == 'y')
  if (bdd_fnsave(argv[4],sa))
    {
      cout << "mnp: Cannot save bdd in file \"" << argv[4] << "\"\nexiting\n";
      exit(1);
    }

// write order output
sprintf(orderoutfile,"%so",argv[4]);
writeorder(orderoutfile,def);

// close bdd package
bdd_done();
      }
    else
      if (!strcmp(argv[1],"-plan4"))
{
  if (plan4(t,init,goal,&sa,def))
    cout << "SA covers Init\n";
  else
    cout << "SA do not cover Init\n";

  // write bdd output
  cout << "Final node count of SA: " << bdd_nodecount(sa) << "\n";
  cout << "Write SA to output file \"" << argv[4] << "\" (y/n): ";
  gets(buf);
  if (buf[0] == 'y')
    if (bdd_fnsave(argv[4],sa))
      {
cout << "mnp: Cannot save bdd in file \"" << argv[4] << "\"\nexiting\n";
exit(1);
      }

  // write order output
  sprintf(orderoutfile,"%so",argv[4]);
  writeorder(orderoutfile,def);

  // close bdd package
  bdd_done();
}
      else
if (!strcmp(argv[1],"-plan5"))
  {
    // read reordering information and adjust domdef
    sprintf(orderinfile,"%so",argv[4]);
    ordering = readorder(orderinfile,def);
    adjust2order(ordering,def);
```

```
    // read SA from bdd infile
    if (bdd_fnload(argv[4],sa))
      {
cout << "mnp: Cannot load SA from file \"" << argv[4] << "\"\nexiting\n";
exit(1);
      }

    // start timing
    startwatch();

    // write one plan to file
    printplan(t,init,goal,sa,argv[5],def);

    // stop timing
    stopwatch();

    // close bdd package
    bdd_done();
  }

      }


  return 1;

}
```

D.5.7 Plan.cc

```
/*****************************************************
 * File  : plan.cc
 * Desc. : Contains misc. planning algorithms
 * Author: Rune M. Jensen CS, CMU, (IAU, DTU)
 * Date  : 4/4/99
 *****************************************************/

#include <bdd.h>
#include "common.hpp"
#include "domain.hpp"
#include "fsm.hpp"
#include "plan.hpp"
#include "bddprint.hpp"



/*****************************************************
 *      Aux. functions
 *
 *****************************************************/



bdd prunestates(bdd preimage, bdd acc)
{
  return preimage & !acc;
}


bdd project(bdd stateact_rules,domdef *def)
{
  return bdd_exist(stateact_rules, def->rels->sysactions);
}

bdd image(bddtrel *tr, bdd s,domdef *def) {
  bdd res;
  int i,partnum;

  partnum = def->dset->subsetnum;

  // inner part of exist composition
  res = s & tr->A & tr->T[0] & tr->IV[0];
  // outer parts of exist composition
  for (i=1; i<partnum; i++)
      res &=  tr->T[i] & tr->IV[i];
```

```
  return res;
}


bdd successor(bdd action,domdef *def) {
  bdd c;
  varrels *rels;

  rels = def->rels;

  c = bdd_exist(action, rels->sysactions);
  c = bdd_exist(c, rels->oldstate);
  return bdd_replace(c,rels->new2old);
}


void  printplan(bddtrel *tr,bdd init,bdd goal,bdd sa,char *planfile,domdef *def) {

  bddprintabs *pb;
  int i;
  bdd s;   // state in planstep encoded in current state variables
  bdd a;   // planstep action encoded in actions and old/new state vars

  pb = new bddprintabs(planfile,def);

  //forward chain
  i = 0; // action count
  pb->bddprinthead();
  s = bdd_satone(init);
  while ((s >> goal) != bddtrue )
    {
      a = image(tr,s,def);
      a = bdd_satone(a & sa);
      pb->bddprintline(a);
      s = bdd_satone(successor(a,def));
      i++;
      if (DEBUG)
cout << "Adding action " << ++i << "\n";
    }
  cout << "steps in plan: " << i << "\n";
}


// IN
//  tr   : Partitioned transition relation
//  acc  : currently visited states (encoded in current state variables)
//  rels : bdd variable info structure
```

```
// OUT
//  strongpreimage of acc encoded in act * current state vars
//  the calculation combines a partitioned representation
bdd strongpreimage(bddtrel *tr, bdd acc,domdef *def)
{
  bdd p1,p2,c;
  int i,partnum;
  varrels *rels;

  partnum = def->dset->subsetnum;
  rels = def->rels;

  c = bdd_replace(acc,rels->old2new);

  // p1 part
  // inner part of exist composition
  p1 = c & tr->A & tr->T[0] & tr->IV[0];
  p1 = bdd_exist(p1, rels->newstate[0]);
  if (DEBUG)
    cout << "Strong Finished partition A0\n";

  // outer parts of exist composition
  for (i=1; i < partnum; i++)
    {
      p1 &=  tr->T[i] & tr->IV[i];
      p1 = bdd_exist(p1, rels->newstate[i]);
      if (DEBUG)
cout << "Strong Finished partition A" << i << " node count: "
     << bdd_nodecount(p1) << "\n";
    }
  // project env. actions
  p1 = bdd_exist(p1,rels->envactions);

  // p2 part
  // inner part of exist composition
  p2 = !c & tr->A & tr->T[0] & tr->IV[0];
  p2 = bdd_exist(p2, rels->newstate[0]);
  if (DEBUG)
    cout << "Strong Finished partition B0\n";

  // outer parts of exist composition
  for (i=1; i<partnum; i++)
    {
      p2 &=  tr->T[i] & tr->IV[i];
      p2 = bdd_exist(p2, rels->newstate[i]);
      if (DEBUG)
cout << "Strong Finished partition B" << i << " node count: "
```

```
     << bdd_nodecount(p2) << "\n";
    }
  // project env. actions
  p2 = bdd_exist(p2,rels->envactions);


  return p1 & !p2;
}


// IN
//  tr   : Partitioned transition relation
//  acc  : currently visited states (encoded in current state variables)
//  rels : bdd variable info structure
// OUT
//  strongpreimage of acc encoded in act * current state vars
//  the calculation combines a partitioned representation
bdd weakpreimage(bddtrel *tr, bdd acc,domdef *def)
{
  bdd res,c;
  int i,partnum;
  varrels *rels;

  partnum = def->dset->subsetnum;
  rels = def->rels;

  c = bdd_replace(acc,rels->old2new);
  if (DEBUG)
    cout << "replaced old2new\n";

  // p1 part
  // inner part of exist composition
  res = c & tr->A & tr->T[0] & tr->IV[0];
  res = bdd_exist(res, rels->newstate[0]);
  if (DEBUG)
  cout << "Weak Finished partition 0 node count: "
     << bdd_nodecount(res) << "\n";

  // outer parts of exist composition
  for (i=1; i<partnum; i++)
    {
      res &=  tr->T[i] & tr->IV[i];
      res = bdd_exist(res, rels->newstate[i]);
      if (DEBUG)
cout << "Weak Finished partition " << i << " node count: "
     << bdd_nodecount(res) << "\n";
    }
```

```
  // project env. actions
  res = bdd_exist(res,rels->envactions);

  return res;
}




/*****************************************************
 *      Deterministic planning (Plan1)
 *      simple backward chaining algorithm
 *****************************************************/

// IN
//  T    : Partitioned transition relation
//  acc  : currently visited states (encoded in current state variables)
//  rels : bdd variable info structure
// OUT
//  simple preimage of acc encoded in act * current state vars
//  that is all S->A pairs leading into acc are included
bdd preimagef(bddtrel *tr, bdd acc,domdef *def)
{
  bdd t,res,c;
  int i,partnum;
  varrels *rels;

  partnum = def->dset->subsetnum;
  rels = def->rels;

  c = bdd_replace(acc,rels->old2new);


  // inner part of exist composition
  res = c & tr->A & tr->T[0] & tr->IV[0];
  res = bdd_exist(res, rels->newstate[0]);

  // outer parts of exist composition
  for (i=1; i<partnum; i++)
    {
      res &=  tr->T[i] & tr->IV[i];
      res = bdd_exist(res, rels->newstate[i]);
      if (DEBUG)
cout << "Finished partition " << i << " node count " << bdd_nodecount(res) << "\n";
"\n";
    }
```

```
  return res;
}



// IN
//  T      : Partitioned transition relation
//  init   : Initial state (encoded in current state vars)
//  goal   : Goal state (encoded in current state vars)
//  def    : Domain definition (containing a description of the
//           location of variables in bdds.
//  outfile: Name of output file
// OUT
//  1:  plan in outfile (plan found)
//  0:  no plan found
int plan1(bddtrel *tr, bdd init, bdd goal, bdd *sa,domdef *def)
{
  bdd acc = goal;               // acc encoded using new state vars (visited states)
  bdd preimage, prunedpreimage;
  int i;


  // make universal plan by backward chaining from goal
  i = 0; // preimage count
  while ( (acc & init) != init )
    {
      preimage = preimagef(tr,acc,def); // act*current bdd
      prunedpreimage = prunestates(preimage,acc); // act*current bdd
      if (prunedpreimage != bddfalse)
{
  *sa |= prunedpreimage;
  acc |= project(prunedpreimage,def);
}
      else
  return 0;
    }
  return 1;
}




/*****************************************************
 *          Plan2
 *          IRST Strong Planning
 *****************************************************/
```

```
// IN
//  T      : Partitioned transition relation
//  init   : Initial state (encoded in current state vars)
//  goal   : Goal state (encoded in current state vars)
//  sa     : plan output handle
//  def    : Domain definition (containing a description of the
//           location of variables in bdds.
//  outfile: Name of output file
// OUT
//  1:  some plan found
//  0:  fix point found, but Init is not a subset
int strongplan(bddtrel *tr, bdd init, bdd goal, bdd *sa,domdef *def)
{
  bdd acc = goal;                // acc encoded using new state vars (visited states)
  bdd preimage, prunedpreimage;
  int i;

  // make universal plan by backward chaining from goal
  i = 0; // preimage count

  //main loop
  while ((acc & init) != init)
    {
      preimage = strongpreimage(tr,acc,def); // act*current bdd
      prunedpreimage = prunestates(preimage,acc); // act*current bdd

      if (prunedpreimage != bddfalse)
{
  *sa |= prunedpreimage;
  acc |= project(prunedpreimage,def);
  if (DEBUG)
    cout << "Added preimage " << i << "\n";
  i++;
}
      else
// fix point found, but init not included
return 0;
    }
  // init is included
  return 1;
}



/*****************************************************
 *          Plan3
 *          IRST Strong Cyclic Planning
```

```
 ****************************************************/

// IN
//  T      : Partitioned transition relation
//  init   : Initial state (encoded in current state vars)
//  goal   : Goal state (encoded in current state vars)
//  sa     : plan output handle
//  def    : Domain definition (containing a description of the
//           location of variables in bdds.
//  outfile: Name of output file
// OUT
//  1:  some plan found
//  0:  no plan found
int strongcyclicplan(bddtrel *tr, bdd init, bdd goal, bdd *sa,domdef *def)
{
  bdd acc = goal;               // acc encoded using new state vars (visited states)
  bdd oldacc = bddfalse;         // old acc
  bdd preimage, prunedpreimage;
  int i,nc;

  // make universal plan by backward chaining from goal
  i = 0; // preimage count

  //main loop
  while (acc != oldacc)
    {
      if ( (acc & init) == init )
return 1;
      preimage = strongpreimage(tr,acc,def); // act*current bdd
      prunedpreimage = prunestates(preimage,acc); // act*current bdd

      if (prunedpreimage != bddfalse)
{
  *sa |= prunedpreimage;
  if (DEBUG)
    {
      cout << "SA node count: " << bdd_nodecount(*sa)  << "\n";
      cout << "updating acc/oldacc\n";
    }
  oldacc = acc;
  acc |= project(prunedpreimage,def);
  if (DEBUG)
    {
      cout << "ACC node count: " << bdd_nodecount(acc) << "\n";
      cout << "Added preimage " << i << "\n";
    }
  i++;
```

```
}
      else
{
  cout << "going into faircycles\n";
  faircycles(tr,&acc,sa,&oldacc,def);
  i++;
}
    }
  return 0;
}



bdd OSA(bddtrel *tr,bdd CSA,bdd s,domdef *def) {
  bdd res,c;
  int i,partnum;
  varrels *rels;

  partnum = def->dset->subsetnum;
  rels = def->rels;

  c = bdd_replace(s,rels->old2new);

  if (DEBUG)
    cout << "replaced old2new\n";

  // p1 part
  // inner part of exist composition
  res = !c & tr->A & tr->T[0] & tr->IV[0];
  res = bdd_exist(res, rels->newstate[0]);
  if (DEBUG)
    cout << "OSA Finished partition 0\n";

  // outer parts of exist composition
  for (i=1; i<partnum; i++)
    {
      res &=  tr->T[i] & tr->IV[i];
      res = bdd_exist(res, rels->newstate[i]);
      if (DEBUG)
cout << "OSA Finished partition " << i << " node count: "
     << bdd_nodecount(res) << "\n";
    }
  // project env. actions
  res = bdd_exist(res,rels->envactions);
```

```
  return res & CSA;
}



// IN
//  T      : Transition relation (partitioned)
//  acc    : Visited states from goal
//  sa     : current s->a table
//  def    : Domain definition (containing a description of the
//           location of variables in bdds.)
// OUT
//  update of SA and acc
void faircycles(bddtrel *T, bdd *acc, bdd *sa,bdd *oldacc, domdef *def) {
  varrels *rels;
  bdd IWpi, Outgoing;

  bdd WpiAcc = bddfalse;
  bdd OldWpiAcc = bddtrue;
  bdd CSA = bddfalse;

  rels = def->rels;

  while (CSA == bddfalse && OldWpiAcc != WpiAcc)
    {
      IWpi = weakpreimage(T,project(WpiAcc,def) | *acc,def);
      CSA = prunestates(IWpi,*acc);
      Outgoing = CSA;
      while (CSA != bddfalse && Outgoing != bddfalse)
{
  Outgoing = OSA(T,CSA,project(CSA,def) | *acc,def);
  CSA &= !Outgoing;
}
      if (CSA != bddfalse)
{
  *sa |= CSA;
  *oldacc = *acc;
  *acc |= project(CSA,def);
  return;
}
      else
{
  OldWpiAcc = WpiAcc;
  WpiAcc |= IWpi;
}
    }
```

```
  *oldacc = *acc;
}


/*****************************************************
 *          Plan4
 *          Optimistic planning algorithm
 *****************************************************/

// IN
//  T    : Partitioned transition relation
//  acc  : currently visited states (encoded in current state variables)
//  rels : bdd variable info structure
// OUT
//  simple preimage of acc encoded in act * current state vars
//  that is all S->A pairs leading into acc are included
//  (similar to preimagef, but in addition projects env. actions
bdd preimagef2(bddtrel *tr, bdd acc,domdef *def)
{
  bdd t,res,c;
  int i,partnum;
  varrels *rels;

  partnum = def->dset->subsetnum;
  rels = def->rels;

  c = bdd_replace(acc,rels->old2new);
  cout << "replaced old2new\n";

  // inner part of exist composition
  res = c & tr->A & tr->T[0] & tr->IV[0];
  res = bdd_exist(res, rels->newstate[0]);
  cout << "Finished partition 0 node count " << bdd_nodecount(res) << "\n";;

  // outer parts of exist composition
  for (i=1; i<partnum; i++)
    {
      res &=  tr->T[i] & tr->IV[i];
      res = bdd_exist(res, rels->newstate[i]);
      if (DEBUG)
cout << "Finished partition " << i << " node count " << bdd_nodecount(res) << "\n";
"\n";
    }

  // project envactions
  res = bdd_exist(res,rels->envactions);
```

```
  return res;
}


// IN
//  T      : Partitioned transition relation
//  init   : Initial state (encoded in current state vars)
//  goal   : Goal state (encoded in current state vars)
//  def    : Domain definition (containing a description of the
//           location of variables in bdds.
//  outfile: Name of output file
// OUT
//  plan ignoring nondeterminism (uses the real preimage in back chaining)
//  1:  init subset of SA (plan found)
//  0:  init not subset of SA (no plan found)
int plan4(bddtrel *tr, bdd init, bdd goal, bdd *sa,domdef *def)
{
  bdd acc = goal;               // acc encoded using new state vars (visited states)
  bdd preimage, prunedpreimage;
  int i;


  // make universal plan by backward chaining from goal
  i = 0; // preimage count
  while ( (acc & init) != init )
    {
      cout << "going into preimagef\n";
      preimage = preimagef2(tr,acc,def); // act*current bdd
      cout << "going into prunedpreimage\n";
      prunedpreimage = prunestates(preimage,acc); // act*current bdd
      if (prunedpreimage != bddfalse)
{
  cout << "updating sa\n";
  *sa |= prunedpreimage;
  cout << "SA node count: " << bdd_nodecount(*sa) << "\n";
  cout << "updating acc\n";
  acc |= project(prunedpreimage,def);
  cout << "ACC node count: " << bdd_nodecount(acc) << "\n";
  cout << "Added preimage " << ++i << "\n";
}
      else
// fix point found, but init not subset of acc
return 0;
    }
  // init subset of acc
  return 1;
}
```

D.5.8 Reorder.cc

```
/*****************************************************
 * File  : reorder.cc
 * Desc. : functions for reading reorder information
 *         when loading a bdd from file.
 * Author: Rune M. Jensen CS, CMU, (IAU, DTU)
 * Date  : 4/4/99
 *****************************************************/

#include <stdio.h>
#include <bdd.h>
#include "fsm.hpp"

void writeorder(char *filename,domdef *def) {
  FILE *orderf;
  int i,out;

  // open outputfile for wrtiting
  if ( (orderf = fopen(filename,"w")) == NULL)
    {
      cout << "reorder.cc writeorder : ";
      cout << "cannot open \"" << filename << "\"\nexiting\n\n";
      exit(1);
    }

  // write order
  for (i=0; i<def->bddvarnum; i++)
    {
      out = bdd_varlevel(i);
      fwrite(&out, sizeof(int), 1, orderf);
    }

}


int *readorder(char *filename,domdef *def) {

  FILE *orderf;
  int *ordering;

  // open inputfile for reading
  if ( (orderf = fopen(filename,"r")) == NULL)
    {
      cout << "reorder.cc readorder: ";
      cout << "cannot open \"" << filename << "\"\nexiting\n\n";
      exit(1);
    }
```

```
  // read order
  ordering = new int[def->bddvarnum];
  fread(ordering, sizeof(int), def->bddvarnum, orderf);

  return ordering;
}
```

D.5.9 Time.cc

```
/*****************************************************
 * File  : time.cc
 * Desc. : Simple functions for measuring run times
 * Author: Rune M. Jensen
 * Date  : 4/8/99
 *****************************************************/

#include <sys/time.h>
#include  <stream.h>

struct timeval tp1,tp2;
struct timezone tzp; // dummy


void startwatch() {
   gettimeofday(&tp1, &tzp);
}

void stopwatch() {

  gettimeofday(&tp2, &tzp);

  if (tp2.tv_usec <  tp1.tv_usec)
    {
      tp2.tv_usec += 1000000;
      tp2.tv_sec--;
    }

  cout << "Time elapsed: " << (tp2.tv_sec - tp1.tv_sec)*1000 +
    (tp2.tv_usec - tp1.tv_usec) / 1000 << " msec \n";
}
```