Limits and Possibilities of PDDL for Model Checking Software

Stefan Edelkamp
Fachbereich Informatik
Universität Dortmund
D-44221 Dortmund
stefan.edelkamp@cs.uni-dortmund.de

Abstract

Automated validation of software systems with model checking technology either certifies that a given designs contain no specification error (like a deadlock or a failed assertion), or falsifies the desired property in form of a counterexample trace from the initial configuration to the error.

Since counterexamples can be seen as goal establishing plans, this paper studies the appropriateness of problem domain description languages (PDDL) to specify software validation problems. The selected software application domain are communication protocols, for which a structured translation from a static subset of the protocol modeling language PROMELA to PDDL2.1 is devised. It exploits the representation of protocols as communicating finite state machines.

First experimental results with metric heuristic search action planners are promising. Since the inferred problem and domain structures are restricted to currently accepted PDDL standards, we expect the protocol domain to be included as a benchmark in the next international planning competition.

Nevertheless, the modeling process makes some limits of PDDL explicit. Subsequently, possible extensions to PDDL are discussed that would ease specification of general software systems and that could widen the applicability of planning technology to software verification.

Introduction

Model checking (Clarke, Grumberg, & Peled 1999) is a formal method for the verification of synchronous and asynchronous systems. It allows a push-button validation of specified properties given in some temporal logic. Since all model-checking techniques rely on state-space enumeration, the major limitations to model checking are large state space sizes. Note that many model checking problems refer to safety properties only. Assertions, global invariances and deadlock-freedom are important classes of safety properties. Roughly speaking, safety properties state that nothing bad will happen. Safety properties can be falsified through displaying one error state.

In action planning (Allen, Hendler, & Tate 1990), domains and problem instances are specified in a problem domain description language, PDDL for short (McDermott 2000). PDDL planning problems usually come in two parts: the problem domain file and the instance specific file. In the first file, predicates and actions are chosen, while in the second file the domain objects, the initial and the goal states are specified. Only the instance specific file refers to grounded predicates, while actions and predicates are specified with object parameters. Recent problem domain description languages developed for action planning, like PDDL2.1 (Fox & Long 2001) are capable to deal with numerical quantities, action duration and plan objective functions (metrics).

Action planning and model checking problems are closely related (Giunchiglia & Traverso 1999). Both approaches explore state spaces of propositionally labeled states, possibly extended with numerical state information. For the case of deterministic planning and safety properties, both areas aim at finding a possibly short path to a pre-specified set of terminal states or at reporting, that no such path exists. Consequently, deterministic planning can be casted as error detection for safety model checking. In planning the path is called a plan in form of a sequence of actions, while in model checking the path of state transitions is referred to as a counterexample.

Several recent planners apply model checking technology by exploring the planning space with binary decision diagrams (BDDs) as concise representations for sets of states, actions and plans. BDDs have been shown to compare well with other optimal determinstic planning approaches (Edelkamp & Helmert 2001). The succinctness of BDDs is even more evident in solving non-deterministic (Cimatti, Roveri, & Traverso 1998) and conformant planning problems (Cimatti & Roveri 1999). Through the application of algebraic decision diagrams the performance gain has recently be migrated to probabilistic planning for finding optimal policies in factored Markov decision processes (Hoey *et al.* 1999; Feng & Hansen 2002).

In (Dierks, Behrmann, & Larsen 2002) PDDL is converted to the input language of the model checker UPPAAL, namely timed automata. UPPAAL uses constraints to represent states symbolically. Simpler problems can be solved optimally by the tool. The numerical computations are further restricted to difference constraints, to build the internal representation of shortest path reduced temporal networks. Running real-time clocks can only be reset to zero. On the other side, UPPAAL allows to progress and project symbolic states (represented as polytopes) and, therefore, can cope with infinite branching problems.

But algorithmic schemes are also exported in the other direction. Satisfiability planning (Kautz & Selman 1996) has been given a semantics for automated validation and has been coined to the term *bounded model checking* (Biere *et al.* 1999). *Directed model checking* applies heuristic search (Pearl 1985) to accelerate error detection. It exploits information of the goal distance to focus the search. Algorithms A* (Hart, Nilsson, & Raphael 1968) and IDA* (Korf 1985) additionally incorporate the estimated path length into the evaluation function to obtain optimal solution paths (Behrmann *et al.* 2001).

Heuristic search is very successful in planning. The first heuristic search planner, HSP (Bonet & Geffner 2001), computes the heuristic values of a state by adding (maximizing) depth values for each fluent for an overestimating (admissible) estimate. These values are retrieved from the fixpoint of a relaxed exploration. Since the technique is similar to the first phase of building the layered graph structure in Graphplan (Blum & Furst 1995), HSPr (Haslum & Geffner 2000) extends the approach by excluding *mutuals*; the *max-pair heuristic* computes a distance value to the goal for each pair of atoms. HSP has inspired the planners like FF (Hoffmann & Nebel 2001), which solves a relaxed planning problem for each encountered state in a combined forward *and* backward traversal. With enforced hill climbing, FF furtherly employs another greedy search strategy to reduce the explored portion of search space. It makes use of the fact that phenomena like big plateaus or local minima do not likely occur in benchmark planning problems.

Alternatively, the pattern database heuristic can be used to approximate goal distances (Edelkamp 2002b). It computes a large lookup-table prior to the search. The database is queried in the overall search process and consists of pattern / distance pairs and is generated through a complete backward exploration of certain problem abstractions. The estimate is consistent and different pattern databases can be merged, by means that their respective heuristic values are to be maximized or added. This heuristic can also be included in BDD exploration engines.

This paper introduces recent advances in planning to model checking. In difference to the expected need for adding planning technology into a model checking tool, the paper puts forth the question of how far one can utilize and extend PDDL to express software verification problems. We exhibit the advantages PDDL planners in model checking domains will have, including static analysis tools, automated symmetry detection, refined heuristic estimators, durative transitions, objective functions, and parallel solution paths. By exploiting the representation of protocols as communicating asynchronous processes, a PDDL domain is devised, accessible for all current metric planners. In the experiments we consider automated falsification of simple protocols through PDDL2.1 models with the planners Metric-FF (Hoffmann 2002) and MIPS (Edelkamp 2003b), compared to results obtained by two efficient explicit state model checkers.

As performance drawbacks of the approach we discuss the loss of domain-dependent implementation issues and acceleration techniques, such as partial order reduction and bit-state hashing. Since we are also interested in the evaluation of expressivity in current PDDL2.1, we reflect limits in the modeling process and grade some requirements general software model checking would impose, including indirect variable addressing, complex goals, and – most prominently – dynamic object creation, which is suggested to become a new feature in upcoming description languages.

Communication Protocols

Communication protocols (Holzmann 1990a) are concurrent software systems with main purpose to organize information exchange between individual processes. Due to the interleaving of process executions and the communication load, the number of global system states is large even for simple and moderate sized protocol specifications. By this combinatorial growth many protocol designs contain subtle bugs. Therefore, in the design process, automated model checking procedures are needed to certify that stated assertions or global invariants are valid, and that no deadlock occurs. Validating these kinds of properties corresponds to solving a reachability problem in the state space graph.

The protocol model refers to a collections of extended communicating finite state machines as described, for instance, in (Brand & Zafiropulo 1983) and (Gouda 1993), where communication between two processes is either realized via synchronous or asynchronous message passing on communication channels or via global variables. Sending or receiving a message is an event that causes a state transition. Successful protocol software model checkers like the SPIN validation tool (Holzmann 1997) interpret the state space as a cross product of such asynchronous finite state machines. For SPIN, the communication protocol has to be provided in PROMELA syntax, a c-like modeling language, extended by non-deterministic choices governed by selecting conditions (Holzmann 1990a).

The protocol model checker SPIN explores the resulting finite search space. Besides depth-first search variants, SPIN may also apply the Supertrace algorithm and sequential hashing to examine various beams in the search trees up to a certain threshold depth. The explicit state model checker also features partial order reduction (Peled 1998),

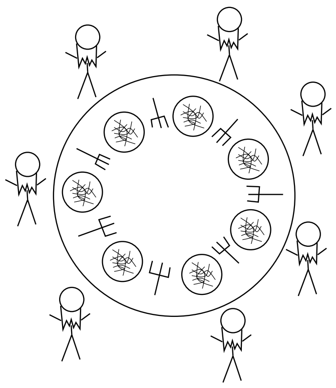

Figure 1: The dining philosophers problem.

```
mtype={fork}
chan forks[3] = [1] of {bit};

active [3] proctype philosopher() {
        forks[_pid]!fork;
        do
        :: forks[_pid]?fork ->
                forks[(_pid+1)%3]?fork;
                forks[_pid] !fork;
                forks[(_pid+1)%3]!fork

        od
}
```

Table 1: Three dining philosophers in PROMELA.

fast hash functions based on a cyclic polynomial representation of state encodings, and different state compression techniques (Holzmann 1990b).

Consider the following simple protocol domain in Figure 1: Dijkstra's dining philosophers problem; n philosophers sit around a table to have lunch. There are n plates, one for each philosopher, and n forks side by side to the plates. Since two forks are required to eat the spaghetti on the plates, not all philosopher can eat at a time. Moreover, no communication except taking and releasing the forks is allowed. The task is to devise a local strategy for each philosopher that lets all philosophers eventually eat. The simplest solution to access the left fork followed by the right one, has an obvious problem. If all philosopher wait for the second fork to be released there is no possible progress; a deadlock has occurred. Modeled as a scalable protocol, the according state space of all philosophers' activities (meditating, waiting, and eating) can rise rapidly.

A protocol in PROMELA is defined by a set of processes. A process consists of statements on a set of local and global variables. Furthermore, sending and receiving messages via communication queues are supported. The PROMELA specification for the dining philosopher solution is given in Table 1. The problem has been implemented in a pure message-passing form. The set of forks is represented as an array `forks` of queues of size 1. The only operation is to access or release a fork. For a philosopher i, `forks[i]` represents the left fork and `forks[(i+1)%3)]` represents the right one. A philosopher is modeled as a process (`proctype` in PROMELA) that performs an endless loop. For grabbing a fork a philosopher tries to receive it from the corresponding queue, while releasing a fork corresponds to send a message. Receive and send operations are expressed as $queue?message$ and $queue!message$, respectively.

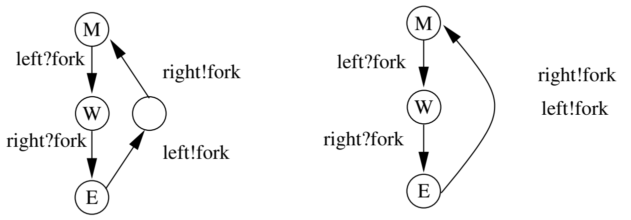

Figure 2: Two state transition diagrams for one philosopher.

The communicating structure between processes can best be viewed in the FSM structure of the problem. Figure 2 shows the enlarged and the compact automata representation for the philosopher process with four and three states, where M, W, and E denote the moods meditating, waiting and eating, respectively.

The first one with four states is similar to the one that is applied in SPIN. It contains an additional state to release the initial fork. In subsequent PDDL models we will refer to the five states as `state-1` (additional state), `state-6` (M), `state-3` (W), `state-4` (E), and `state-5` (◯), respectively. The second representation is added to concisely illustrate the growth of the state space in Figure 3. For three philosophers the states are members of the cross product of the three automata, so that the reachable state space is a subset of all 3^3 states.

Directed Software Validation

Large state spaces call for different algorithmic aspects to reduce the search efforts for exploration, one of which is guided exploration with respect to the set of errors to be found. Heuristic search algorithms take additional information in form of an evaluation function into account and return a number measuring the desirability of expanding a state. When the states are ordered so that the one with the best evaluation is expanded first and if the evaluation function estimates the cost of the cheapest path from the current state to a desired one, hill-climbing finds solutions fast. However, it may suffers from the same defects as depth-first search – it is not optimal and may be stuck in dead-ends or local minima. Early approaches (Lin, Chu, & Liu 1988; Yang & Dill 1998) propose greedy best-first search with simpler heuristics like the Hamming distance. Recent attempts, e.g. (Cobleigh, Clarke, & Osterweil 2001) success-

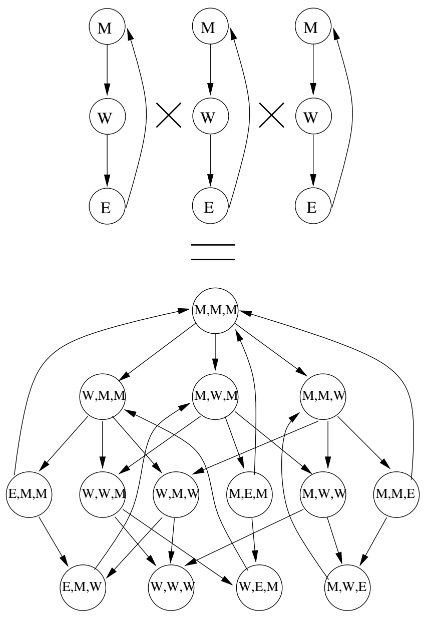

Figure 3: The state space for 3 philosophers.

fully integrate different heuristic search algorithms to accelerate the exploration. Most of the techniques can be applied to the detection of safety properties only. In addition, (Edelkamp, Lluch-Lafuente, & Leue 2001) presents an approach for shortening given error traces. (Godefroid & Khurshid 2002) propose genetic algorithms for finding errors in very large state spaces with different heuristics for deadlock detection and assertion violation.

In explicit state model checking partial order reduction is one of the other most effective techniques to avoid the state explosion problem. It exploits the commutativity of asynchronous system and constructs a reduced state space that is equivalent to the original one. (Lluch-Lafuente, Leue, & Edelkamp 2002) have shown that heuristic search and partial order reduction are more or less orthogonal approaches, i.e., one scheme does not loose its effectiveness if the other technique is applied beforehand or in parallel.

The efficiency of guided exploration mainly depends on the quality of the estimator function. We briefly recall the estimates that have proven to be effective. The first two are suited to deadlock detection, while the latter one allows complex error formulae.

A first intuitive idea for estimating the distance to a deadlock is to count the number of non-blocked processes in the current state. This estimate is a lower bound for all protocols (except for those with rendezvous communication).

For the FSM distance heuristic we pre-compute for each finite state machine M a matrix $D_M = (d_{ij})$, where d_{ij} denotes the minimal number of transitions from local state i to local state j in M. Each matrix D_M is computed with an all-pairs shortest-path algorithm in cubic time. To calculate the estimate, we additionally locate the set of dangerous states for each FSM. A dangerous state is a state from which all transitions are dangerous, and a dangerous transition in turn is a transition that is not throughout executable. For example, a transition representing an assignment is not dangerous but transitions representing operations over queues or conditions over shared variables are not always executable, e.g send or receive operations. If all transitions from a state are dangerous, it is possible that no transition is executable and that the corresponding process is blocked.

The most general heuristic is the formula-based heuristic: given a error formula e, devise an estimate $H(S, e)$ of how many transitions from the current state S are necessary to encounter a state, in which the formula e is true. $H(S, e)$ is defined recursively on the structure of e.

The guided experimental model checking platform HSF-SPIN (Edelkamp, Leue, & Lluch-Lafuente 2003) chooses SPIN as a basis for the implementation with same in- and output format, additionally providing an interface for AI search methods. It currently features heuristic search algorithms like best-first search, A* (Hart, Nilsson, & Raphael 1968), IDA* (Korf 1985) with bit-state hashing or transposition tables (Reinefeld & Marsland 1994), and some forms of partial order reduction.

PDDL Modeling

Instead of the PROMELA file itself, we start with the automata representation that is produced by SPIN[1]. As said, we assume all processes to be static. This imposes some restrictions to the original PROMELA model, since in this case individual processes cannot invokes sub-processes. Fortunately, in our set of benchmark protocol domains[2], all PROMELA processes (`proctypes`) are invoked by a single init process loop, which can be compiled away. The PROMELA modifications are more or less textual substitutions, which presumably can be provided by automated procedures. So far these changes have been performed manually, but even for involved examples this step is not a burden for the protocol designer. In the general case of validating software, however, dynamic process invocation and new object creation is crucial. This defect of PDDL expressivity is discussed to the end of this paper.

A good model of a planning problem keeps the number of parameters small. Grounding actions and predicates with more than five object parameters causes problems for almost any planner that we know. Fewer parameters can best be obtained by some additional flags that are set by one ac-

[1]We take the PROMELA input file, generate the corresponding c-file, and run the executable with option `-d` to obtain the finite state representations of all processes. We avoid state merging by setting parameter `-o3` as a option to SPIN.

[2] `www.informatik.uni-freiburg.de/∼lafuente/hsf-spin`

```
(:action activate-trans
   :parameters
      (?p - process ?t - transition ?s1 ?s2 - state)
   :precondition
      (and
         (trans ?t ?s1 ?s2)
         (at-process ?p ?s1)
      )
   :effect
      (activate ?p ?t)
)

(:action perform-trans
   :parameters
      (?p - process ?t - transition ?s1 ?s2 - state)
   :precondition
      (and
         (trans ?t ?s1 ?s2)
         (ok ?p ?t)
         (at-process ?p ?s1)
      )
   :effect
      (and
        (at-process ?p ?s2)
        (not (at-process ?p ?s1))
        (not (ok ?p ?t))
      )
)
```

Figure 4: Preparing and executing a process state transition.

tion and queried by another one.

Next we identify processes, proctype, and a propositional description of the finite state system with states and state transitions as objects. The array dimensions of process types, variables, and queues as well as queue capacity are read from the PROMELA input file[3]. Variables in the PROMELA specification are also to be found and declared as objects. Proper handling of shared variables in PDDL is more involved and considered later on.

At first we concentrate on communication via channels, where channels are defined by their channel type and content configuration. All these objects are inferred in the parser by instantiating the process identifier `_pid` with respect to the established array bounds in the PROMELA description. Figure 4 shows how we prepare and execute state transitions. The Appendix provides the entire PDDL2.1 specification in the dining philosophers problem[4]. Action `activate-trans` activates transition t in process P if in the current local state we have an option to perform t starting from local state s_1. Action `perform-trans` triggers the transition t in process P to move from s_1 to s_2. It queries flag `ok`, which is then deleted.

[3]This is the only additional information, that is not present in the finite state representation file. To avoid conflicts with pre-compiler directives, we substitute all `defines` beforehand with the c-compiler command line option `-E`, which runs the pre-compiler only.

[4]For a proper parsing process in MIPS brackets in transition descriptions were automatically substituted with underscores. For Metric-FF more involved changes are necessary, reducing the readibility of the generated code.

```
(:action queue-read
   :parameters (?p - process ?t - transition
                ?q - queue ?v - variable)
   :precondition
      (and
        (activate ?p ?t)
        (settled)
        (reads ?p ?q ?t)
        (reads-val ?p ?t ?v)
        (>= (queue-size ?q) 1)
        (= (queue-head-msg ?q) (trans-mess ?t))
       )
    :effect
      (and
        (advance-head ?q)
        (ok ?p ?t)
        (not (activate ?p ?t))
        (not (settled ?q))
        (assign (value ?v) (head-value ?q))
      )
)
```

Figure 5: Reading variables from a queue.

```
(:action increase-head
   :parameters (?q - queue ?qt - queuetype
                ?qs1 ?qs2 - queue-state)
   :precondition
     (and
        (queue-next ?qt ?qs1 ?qs2)
        (is-a-queue ?q ?qt)
        (queue-head ?q ?qs1)
        (advance-head ?q)
        (>= (queue-size ?q) 1)
     )
   :effect
     (and
        (settled)
        (queue-head ?q ?qs2)
        (not (queue-head ?q ?qs1))
        (not (advance-head ?q))
        (assign (queue-head-value ?q) (queue-value ?q ?qs2))
        (assign (queue-head-msg ?q) (queue-msg ?q ?qs2))
        (decrease (queue-size ?q) 1)
     )
)
```

Figure 6: Increasing the head pointer to settle the queue.

This implies that for each transition t in the automaton there has to be an appropriate action that performs all necessary changes according to t and that sets the according flag `ok` to true.

Queue organization is obtained with explicit head and tail pointers. Figure 5 gives an action specification for reading a variable v from the queue q in transition t querying message m, i.e. $Q?m(v)$. The PDDL representation of $Q!m(v)$ is analogous. We can see that the state transition enabling flag `ok` is set. The according queue update action `increase-head` is shown in Figure 6. As the name indicates it actualizes the head position and eliminates the *settled* flag, which is preconditioned in any queue access action. To disallow actions to be activated twice, before an action is performed we additionally precondition `activate-trans` and `perform-trans` with the settlement of the queues.

	Metric-FF		MIPS		SPIN		HSF-SPIN	
p	l	e	l	e	l	s	l	e
3	6	7	6	7	18	10	14	17
4	8	13	8	9	54	45	18	22
5	10	21	10	11	66	51	22	27
6	12	31	12	13	302	287	26	32
7	14	43	14	15	330	309	30	37
8	16	57	16	17	1.362	1,341	34	42
9	18	73	18	19	1,466	1,440	38	47
10	20	91	20	21	9,422	9,396	42	52
11	22	111	22	23	9,382	9,349	46	46
12	24	133	24	25	9,998	43,699	50	62
13	26	157	26	27	9,998	722,014	54	67
14	28	183	28	29	o.m	o.m	58	72
15	30	211	30	31	o.m	o.m	62	58

Table 2: Results for finding the deadlock in the n dining philosophers problem.

Table 2 displays obtained exploration result for a slightly simpler PDDL encoding of the dining philosopher problem [5]. It shows the number of expanded/stored node e/s and counterexample path length l: o.m abbreviates out of main memory for the two planners *Metric-FF* and *MIPS* and the two model checkers *SPIN* and *HSF-SPIN*. For more and refined experimental results, e.g. in the Optical Telegraph and Leader Election protocol, and for some objections to the fairness for such a comparison, the reader is referred to (Edelkamp 2003a).

In the example heuristics are effective, since *SPIN* searches to large depths until it finds an error. Its standard DFS exploration order misses shallow errors. The suboptimal numbers of expansions in the model checkers are due to graph contraction features, especially due to state merging. The planner *Metric-FF* has a node expansion routine about as fast as HSF-SPIN, which is very good, considering that the planner uses a set of propositional information, computes an involved estimate and does not have any information on the application domain.

Table 3 depicts a plan to the 4 philosopher problem with transition activation in competition format as produced by MIPS. It was found in 35 node expansions. The leading number show the start time the action has. As indicated, MIPS does exploit parallelism. With durative actions, MIPS also allows to devise temporal and metric aspects to the protocol validation task.

In pure message passing domains like the example considered here, no communication via shared variables is necessary. If variables are used, the parser has to generates action schemas for variable conditioning and change. However, these operation appear to be more difficult than other

[5]Explicit activation of transitions was not implemented in this case. State transition activation and execution were merged into one PDDL action

```
0: (activate-trans philosopher-3 forks[-pid]!fork state-1 state-6) [1]
0: (activate-trans philosopher-2 forks[-pid]!fork state-1 state-6) [1]
0: (activate-trans philosopher-1 forks[-pid]!fork state-1 state-6) [1]
0: (activate-trans philosopher-0 forks[-pid]!fork state-1 state-6) [1]
1: (queue-write philosopher-3 forks[-pid]!fork forks[3]) [1]
2: (increase-queue-tail1 forks[3] queue-1 qs-0 qs-0) [1]
3: (queue-write philosopher-2 forks[-pid]!fork forks[2]) [1]
4: (increase-queue-tail1 forks[2] queue-1 qs-0 qs-0) [1]
5: (queue-write philosopher-1 forks[-pid]!fork forks[1]) [1]
6: (increase-queue-tail1 forks[1] queue-1 qs-0 qs-0) [1]
7: (queue-write philosopher-0 forks[-pid]!fork forks[0]) [1]
8: (increase-queue-tail1 forks[0] queue-1 qs-0 qs-0) [1]
9: (perform-trans philosopher-0 forks[-pid]!fork state-1 state-6) [1]
9: (perform-trans philosopher-1 forks[-pid]!fork state-1 state-6) [1]
9: (perform-trans philosopher-2 forks[-pid]!fork state-1 state-6) [1]
9: (perform-trans philosopher-3 forks[-pid]!fork state-1 state-6) [1]
10: (activate-trans philosopher-0 forks[-pid]?fork state-6 state-3) [1]
10: (activate-trans philosopher-1 forks[-pid]?fork state-6 state-3) [1]
10: (activate-trans philosopher-2 forks[-pid]?fork state-6 state-3) [1]
10: (activate-trans philosopher-3 forks[-pid]?fork state-6 state-3) [1]
11: (queue-read philosopher-3 forks[-pid]?fork forks[3]) [1]
12: (increase-queue-head forks[3] queue-1 qs-0 qs-0) [1]
13: (queue-read philosopher-2 forks[-pid]?fork forks[2]) [1]
14: (increase-queue-head forks[2] queue-1 qs-0 qs-0) [1]
15: (queue-read philosopher-1 forks[-pid]?fork forks[1]) [1]
16: (increase-queue-head forks[1] queue-1 qs-0 qs-0) [1]
17: (queue-read philosopher-0 forks[-pid]?fork forks[0]) [1]
18: (increase-queue-head forks[0] queue-1 qs-0 qs-0) [1]
19: (perform-trans philosopher-0 forks[-pid]?fork state-6 state-3) [1]
19: (perform-trans philosopher-1 forks[-pid]?fork state-6 state-3) [1]
19: (perform-trans philosopher-2 forks[-pid]?fork state-6 state-3) [1]
19: (perform-trans philosopher-3 forks[-pid]?fork state-6 state-3) [1]
```

Table 3: Counterexample for the four philosopher problem.

```
(:action V0=V1
   :parameters
      (?p - process ?t - transition ?v0 ?v1 - variable)
   :precondition
      (and
        (activate ?p ?t)
        (is-V0=V1 ?t ?v0 ?v1)
        (inside ?p ?t ?v0)
        (inside ?p ?t ?v1)
      )
   :effect
      (and
        (ok ?p ?t)
        (not (activate ?p ?t))
        (assign (value ?v0) (value ?v1))
      )
)
```

Figure 7: Assigning variables.

operations, since they require both changes to the instance and problem domain file.

To tame the number of actions, for each condition or assignment the parser generates a pattern structure. For example, setting any variable to 1 corresponds to a `V0=1` pattern. The assignment of any variable to the content of another corresponds to a `V0=V1` pattern.

Inferred patterns generate actions and initial state patterns. E.g. `V0=V1` generates a `is-V0=V1` predicate, to be grounded in the initial state for each variable-to-variable assignment according to the given transition and process for the initial state. The inferred action declaration for the domain file is shown in Table 7. The `inside` predicate avoids the definition of a fourth parameter in the `is-V0=V1` predicate.

Limits and Possibilities of PDDL Modeling

In this section we reflect the contributed work in form of lessons to be learned with respect to current PDDL expressivity. On the positive side, we highlight the following aspects.

1. We have seen a new approach to protocol validation through compiling a selection of PROMELA specifications into PDDL, with first encouraging results for simple examples. The language capabilities in PDDL appear to be general and flexible enough to model even larger protocols. The intermediate representation in PDDL is handy for designers, that prefer some representation of communicating state machines instead of PROMELA sources. These PDDL encodings may also be exploited by other model checkers, like the one proposed by (Esparza, Römer, & Vogler 2002) that looks at *unfoldings*.

2. Bypassing model checkers through planning is an apparent alternative to current options in (directed) protocol validation. Metric planners like FF, LPG and MIPS seem to exploit more information on the goals by inferring refined estimates and including different forms of static information, such as goal ordering, landmark approximation, clustering of mutual exclusive atoms, state invariances, and automated symmetry detection. On the other hand, the exploration efficiency of model checkers due to a refined implementation is a challenge for planners.

3. It is also true, that current PDDL2.1 planners, like LPG and MIPS can handle certain forms real-time aspects and concurrencies in plans. In PDDL2.1, actions can be attached to time intervals. Subsequently, a plan is no longer a sequence of action but a schedule of them.

4. PDDL 2.1 is capable of very general plan objective functions, so that different plans can be ranked according to their quality or execution cost. For software verification such an option can be favorable in order to optimize counterexamples based on parameters different to the number of transitions that they contain.

5. PDDL protocol domain specifications are concise and consist of a few state and queue enabling and committing actions, together with some variable update patterns. Through the use of typed parameters it allow to detect object transpositions.

6. The specification process is almost fully automated starting from the process automata representation produced by an existing model checker. So far, we neither changed the model-checker to compile the current instance, nor the planners' internal design, proving their domain-independent implementation.

7. By altering the link structure process communication in PDDL can easily be adapted to other channel implementations.

8. The design patterns for specifying communication protocols in PDDL are very general and likely to transfer to the validation of other model checking exploration tasks, e.g to the verification of security protocols.

From a theoretical point, PDDL2.1 is Turing equivalent (Helmert 2002), such that, in principle any software verification problem can be modeled. However, these transformations are not practical, so there are minor and major restrictions of PDDL for software verification.

1. Some parsers of planners have problems with action and object names in the full ASCII character set. Currently, we substitute misinterpreted characters, by the cost of problem readability. Extending PDDL parsers in planners is likely to be a small coding effort only.

2. Through the propositional representation of automata and communication channels, PDDL specifications may become a bit lengthy. Especially, the lack of object arrays appear uncomfortable for the designer. Since this issue can be viewed as syntactic sugar and since for most planning benchmarks instance files are generated automatically.

3. The work has turned a verification problem containing non-deterministic choice into a deterministic planning problem. However, it has neglected advanced aspects in protocol modeling, such as rendezvous communication and atomic transitions. While the latter seems not to be a severe restriction, the former has to be handled with care, since rendezvous communication affects the asynchronous behavior of the system. It is still open, if (a)synchronicity can be easily exploited in PDDL.

4. We have not considered LTL and liveness properties yet. In PDDL terminology this would correspond to temporarily extended goals, a hot topic tackled by many research groups. SPIN's option is to explicitly model the temporal requirements as an additional (*never-claim*) automaton.

5. Since the general exploration problem is undecidable, PDDL exploration engines may not terminate at all. SPIN, however, insists on finite domain variables in PROMELA to maximally traverse a finite state space graph. Finite variable domains for PDDL variables such as a type `byte` are not supported in PDDL. There is some early work in fixing variable domains automatically to improve planners' performance (Edelkamp 2002a), but since this issue is as hard as planning itself, introducing finite domain variables can be helpful

for many planners. In fact, finite domain variables can then be internally parsed down to propositional variables.

6. For failed assertions and domain invariance violations the explicit statement of the goals causes no problem. However, explicitly stating the deadlock situation appears as a further burden to the user. Of cause, this goal condition can be derived from stopping criteria based on the precondition of each action, but a PDDL language extension at this point is preferable. This may also allow the planners to alter their traversal politics.

7. Language extensions in PDDL should include domain axioms, i.e. certain actions declaration that are triggered by others and that are executed automatically to preserve a valid system state. The work of (Tiebaux, Hoffmann, & Nebel 2003) has come up with a formal semantics for axioms, easily checkable conditions for them to be well-defined, as well as a proof that they can not be compiled away without significant costs. Integrating an explicit treatment of axioms is trivial within a forward search (but might be more complicated in other approaches). In the protocol domain this form of modeling avoids to explicitly settle queues.

8. The transformation proposes to bypass the model checker with a domain-independent action planner. A fairer comparison would be to devise domain-dependent pruning rules concurrently checked during the planning process. Examples of these kinds of planners rules are TL-Plan (Bacchus & Kabanza 2000) and TalPlan (Kvarnström, Doherty, & Haslum 2000).

9. Complex variable constructs like indirectly addressed variables are not properly tackled yet. Variable patterns derived so far seem to be too weak at this point. This problem has been encountered by translating the elevator simulator of A. Biere[6]. We expect that the problem can be circumvented by deflating the array index in the precondition at the price of a neat domain description.

10. The fixed state representation of current planners cannot yet model dynamic process creation. For general software verification this problem is crucial, since almost all state facets, e.g. to represent variables and threads refer are dynamic. In PDDL dynamic domains call for object creation and deletion in action effects. One would require additional effects that are of the form `(new (?a - <name>) (:init <formula>*))` and `(delete ?a)`. The former construct generates a new object, the latter construct deletes an existing one. These small PDDL extensions are also desirable in several planning benchmarks and real-world setting such as the ones at NASA AMES. Another example is the Settlers domain, where vehicles (carts, trains, ships) have to be created to provide the necessary transport infrastructure.

    Introducing dynamic domains will lead to substantial internal changes of planners and static analyzers. At least for explicit state forward chaining planners, extendible state representations can be made available. As it is desirable to have planners that can deal with the new feature one would like existing planners to solve transformed benchmark domains without it.

    Such a transformation to ordinary PDDL is possible, if one has a superset of all possible objects to be generated available. In general, however, finding such a superset is as difficult as the plan existence problem, so that we cannot expect a fully automated compiling scheme without designer help.

    We briefly sketch some transformation details to clarify the semantics of the language extension. Through this will not be formal construction, it indicates necessary changes. The transformation uses some special predicates as flags to govern the appropriate object handling. Most importantly we include a predicate `(is-created ?a - <name>)` in the domain description, and precondition every object in every action, if it is already created.

    An action with the effect `(new (?a - <name>) (:init (<formula>*))` is transformed introducing an ordinary add-effect `(is-created ?a)`. Since `?a` is not yet a parameter of the action the compilation will include an extra parameter into the transformed action.

    For example consider action `gen-block` that has no parameter and the only effect is `(new (?a - block) (:init (holding ?a)))`. It will be transferred to an action `gen-block′ (?a - block)` having `(holding ?a)` in the effect list. Certainly, the transferred action should require `(not (is-created ?a))` to be included in the precondition list. Recall that by the observation of Gazen and Knoblock negated preconditions can be easily compiled away, as e.g. performed by planners like FF. Actually it would only be necessary to include the additional predicate `not-is-created` into the precondition list and to guarantee that in each action the predicates `not-is-created` and `is-created` are complement to each other.

    Similarly, for the delete pattern `(delete ?a)` the flag `(is-created ?a)` has to be added to the delete list of the according action.

Conclusions

PDDL models of communication protocols allow the first immediate application of action planners to software model

[6] www.inf.ethz.ch/personal/biere/ applets/elsim

checking domains. Hence, from a principal point of view, we contributed and evaluated a new alternative to current research efforts to model check protocols. Instead of introducing planning technology (like heuristic search) to model checkers we propose to use PDDL specification directly, even if some drawbacks to the expressivity are quite obvious. The performance gain has still to be indicated in a broader example set.

From a practical point of view we contributed a new benchmark domain to evaluate current planning technology and suggest to include the domain in the next planning competition in the track for real-world application of deterministic (domain-independent or handcoded) planning.

The approach shows that protocol validation can be implemented as a deterministic exploration problem. The nondeterminism in the input language PROMELA selected by guards is available by the set of actions to be selected by the planner. Through modeling state transitions as action, the asynchronous behaviour is implicit. To keep specifications small in current planners, in the proposed model we tried to reduce the set of parameters.

The work targets the transfer between the state space exploration areas AI planning and model checking. It provides a practical study of the expressivity of different input specifications, introduces concurrency and quality metrics to counterexample traces, and compares search, pruning and acceleration methods, e.g. the *relaxed plan heuristic*, which comes along with an *enforced hill climbing* search engine. Through the inferred intermediate planner input format, we provide the basis for a new algorithmic aspects to the exploration process, aim to further bridge the gap between the two research areas of model checking and action planning. In the long term we expect planner and model checking technology to merge.

Through the introduction of numbers and duration and by the design of efficient domain-dependent and hand-coded planners, the practical impact of planning technology has greatly increased. Other rising application besides verification are e.g. pipelining transportation, airport scheduling, bio-informatics applications, circuit diagnosis, etc. Extending PDDL is an open project and subject to many discussion. We contributed practical aspects in form of limits and possibilities to current PDDL, by comparing its expressivity with the requirements imposed by software model checking problems. We suggest to include dynamic object creation as a new language feature.

In future research we plan to look at Java and C++ verification as well as probabilistic model checking. For the latter factored Markov Decision Processes (FMDPs) as a welcome theoretical fundament. There is current work in extend PDDL specification in order to attach probabilities to action effects. By means of the contributed work, we think of using a FMDP solver like SPUDD (Hoey *et al.* 1999) and to transfer annotated PROMELA into an intermediate probabilistic action planning description language.

Acknowledgments S. Edelkamp is supported by DFG in the projects *Directed Model Checking*, and *Heuristic Search*. Thanks to A. Lluch-Lafuente and Jörg Hoffmann for helpful discussions on the topic.

References

Allen, J. F.; Hendler, J.; and Tate, A., eds. 1990. *Readings in Planning*. San Franscico, CA: Morgan Kaufmann.

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116:123–191.

Behrmann, G.; Fehnker, A.; Hune, T.; Larsen, K.; Petterson, P.; Romijn, J.; and Vaandrager, F. W. 2001. Efficient guiding towards cost-optimality in uppaal. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

Biere, A.; Cimatti, A.; Clarke, E.; and Zhu, Y. 1999. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science. Springer.

Blum, A., and Furst, M. L. 1995. Fast planning through planning graph analysis. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, 1636–1642.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence*.

Brand, D., and Zafiropulo, P. 1983. On communicating finite-state machines. *Journal of the ACM* 30(2):323–342.

Cimatti, A., and Roveri, M. 1999. Conformant planning via model checking. In *European Conference on Planning (ECP)*, 21–33.

Cimatti, A.; Roveri, M.; and Traverso, P. 1998. Automatic OBDD-based generation of universal plans in non-deterministic domains. In *National Conference on Artificial Intelligence (AAAI)*, 875–881.

Clarke, E. M.; Grumberg, O.; and Peled, D. A. 1999. *Model Checking*. MIT Press.

Cobleigh, J. M.; Clarke, L. A.; and Osterweil, L. J. 2001. The right algorithm at the right time: Comparing data flow analysis algorithms for finite s tate verification. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, 37–46. IEEE Computer Society.

Dierks, H.; Behrmann, G.; and Larsen, K. G. 2002. Solving planning problems using real-time model checking (translating pddl3 into timed automata). In *Artificial Intelligence Planning and Scheduling (AIPS)-Workshop on Planning via Model-Checking*, 30–39.

Edelkamp, S., and Helmert, M. 2001. The model checking integrated planning system MIPS. *AI-Magazine* 67–71.

Edelkamp, S.; Leue, S.; and Lluch-Lafuente, A. 2003. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology (STTT)*.

Edelkamp, S.; Lluch-Lafuente, A.; and Leue, S. 2001. Trail-directed model checking. In *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers.

Edelkamp, S. 2002a. Mixed propositional and numerical planning in the model checking integrated planning system. In *International Conference on AI Planning & Scheduling (AIPS), Workshop on Temporal Planning*, 47–55.

Edelkamp, S. 2002b. Symbolic pattern databases in heuristic search planning. In *Artificial Intelligence Planning and Scheduling (AIPS)*, 274–283.

Edelkamp, S. 2003a. Promela planning. In *Model Checking Software (SPIN)*, Lecture Notes in Computer Science, To Appear. Springer.

Edelkamp, S. 2003b. Taming numbers and durations in the model checking integrated planning system. *Journal of Artificial Research (JAIR)*. Submitted, A draft is available at PUK-Workshop 2002.

Esparza, J.; Römer, S.; and Vogler, W. 2002. An improvement of mcmillan's unfolding algorithm. *Formal Methods in System Design* 20(3):285–310.

Feng, Z., and Hansen, E. 2002. Symbolic heuristic search for factored markov decision processes. In *National Conference on Artificial Intelligence (AAAI)*.

Fox, M., and Long, D. 2001. PDDL2.1: An extension to PDDL for expressing temporal planning domains. Technical report, University of Durham, UK.

Giunchiglia, F., and Traverso, P. 1999. Planning as model checking. In *European Conference on Planning (ECP)*, 1–19.

Godefroid, P., and Khurshid, S. 2002. Exploring very large state spaces using genetic algorithms. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

Gouda, M. G. 1993. Protocol verification made simple: a tutorial. *Computer Networks and ISDN Systems* 25(9):969–980.

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

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *Artificial Intelligence Planning and Scheduling (AIPS)*, 140–149.

Helmert, M. 2002. Decidability and undecidability results for planning with numerical state variables. In *Artificial Intelligence Planning and Scheduling (AIPS)*, 44–53.

Hoey, J.; Aubin, R.; Hu, A.; and Boutilier, C. 1999. Spudd: Stochastic planning using decision diagrams. In *Conference on Uncertainty in Artificial Intelligence (UAI)*.

Hoffmann, J., and Nebel, B. 2001. Fast plan generation through heuristic search. *Artificial Intelligence Research* 14:253–302.

Hoffmann, J. 2002. Extending FF to numerical state variables. In *European Conference on Artificial Intelligence*.

Holzmann, G. J. 1990a. *Design and Validation of Computer Protocols*. Prentice Hall.

Holzmann, G. 1990b. Algorithms for automated protocol verification. *AT&T Technical Journal* 69(2). Special Issue on Protocol Testing, Specification, and Verification.

Holzmann, G. J. 1997. The model checker Spin. *IEEE Transactions on Software Engineering* 23(5):279–295.

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

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97–109.

Kvarnström, J.; Doherty, P.; and Haslum, P. 2000. Extending TALplanner with concurrency and ressources. In *European Conference on Artificial Intelligence (ECAI)*, 501–505.

Lin, F. J.; Chu, P. M.; and Liu, M. 1988. Protocol verification using reachability analysis: the state space explosion problem and relief strategies. *ACM* 126–135.

Lluch-Lafuente, A.; Leue, S.; and Edelkamp, S. 2002. Partial order reduction in directed model checking. In *Workshop on Model Checking Software (SPIN)*, Lecture Notes in Computer Science, 112–127. Springer.

McDermott, D. 2000. The 1998 AI Planning Competition. *AI Magazine* 21(2).

Pearl, J. 1985. *Heuristics*. Addison-Wesley.

Peled, D. 1998. Partial order reductions. In Inan, M. K., and Kurshan, R. P., eds., *Verification of Digital and Hybrid Systems*, 117–137. Springer.

Reinefeld, A., and Marsland, T. 1994. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16(7):701–710.

Tiebaux, S.; Hoffmann, J.; and Nebel, B. 2003. In defense of PDDL axioms. Technical Report ARP-01-03, Automated Reasoning Group, Research School of Information Sciences and Engineering Australian National University.

Yang, C. H., and Dill, D. L. 1998. Validation with guided search of the state space. In *Conference on Design Automation (DAC)*, 599–604.

Appendix: PDDL2.1 Specification for the Four Dining Philosopher Example as Inferred by the Promela-To-PDDL Compiler

In the following we give the full PDDL2.1 specification for the protocol domain as inferred by parsing the 4 dining philosopher PROMELA input. Since this domain contains no global variable handling, according source lines are commented.

We start with the protocol domain file.

```
(define (domain protocol)
 (:requirements :typing)
 (:types process proctype state queue transition variable position
        queuetype queue-state
        - object)
 (:predicates
   (is-a-queue ?q - queue ?pt - queuetype)
   (at-process ?p - process ?s - state)
   (inside ?p - process ?t - transition ?v - variable)
   (trans ?t - transition ?s1 ?s2 - state)
   (writes ?p - process ?q - queue ?t - transition)
   (reads ?p - process ?q - queue ?t - transition)
   (writes-val ?p - process ?t - transition ?v - variable)
   (reads-val ?p - process ?t - transition ?v - variable)
   (ok ?p - process ?t - transition)
   (activate ?p - process ?t - transition)
   (queue-next ?qt - queuetype ?qs1 ?qs2 - queue-state)
   (queue-head ?q - queue ?qs - queue-state)
   (queue-tail ?q - queue ?qs - queue-state)
   (advance-tail ?q - queue)
   (advance-head ?q - queue)
   (settled)
)
 (:functions
   (value ?v - variable)
   (queue-size ?q - queue)
   (queue-max ?qt - queuetype)
   (queue-value ?q - queue ?p - queue-state)
   (queue-msg ?q - queue ?p - queue-state)
   (queue-head-value ?q - queue)
   (queue-tail-value ?q - queue)
   (queue-head-msg ?q - queue)
   (queue-tail-msg ?q - queue)
   (trans-msg ?t - transition)
)
(:action queue-read
   :parameters (?p - process
                ?t - transition
                ?q - queue
;;              ?v - variable
   )
   :precondition
      (and
         (activate ?p ?t)
         (settled)
         (reads ?p ?q ?t)
;;       (reads-val ?p ?t ?v)
         (= (queue-head-msg ?q) (trans-msg ?t))
         (>= (queue-size ?q) 1))
    :effect
      (and
;;       (assign (value ?v) (queue-head-value ?q))
         (advance-head ?q)
         (ok ?p ?t)
         (not (activate ?p ?t))
         (not (settled))
      )
)
(:action queue-write
   :parameters (?p - process
                ?t - transition
                ?q - queue
;;              ?v - variable
   )
   :precondition
      (and
         (activate ?p ?t)
         (settled)
         (writes ?p ?q ?t)
;;       (writes-val ?p ?t ?v)
      )
   :effect
      (and
         (ok ?p ?t)
         (not (settled))
         (not (activate ?p ?t))
         (advance-tail ?q)
         (assign (queue-tail-msg ?q) (trans-msg ?t))
;;       (assign (queue-tail-value ?q) (value ?v))
     )
)

;; due to queue-read

(:action increase-queue-head
   :parameters (?q - queue ?qt - queuetype ?qs1 ?qs2 - queue-state)
   :precondition
     (and
        (queue-next ?qt ?qs1 ?qs2)
        (is-a-queue ?q ?qt)
        (queue-head ?q ?qs1)
        (>= (queue-size ?q) 1)
        (advance-head ?q)
     )
   :effect
     (and
        (settled)
        (queue-head ?q ?qs2)
        (not (advance-head ?q))
        (not (queue-head ?q ?qs1))
;;      (assign (queue-head-value ?q) (queue-value ?q ?qs2))
        (assign (queue-head-msg ?q) (queue-msg ?q ?qs2))
        (decrease (queue-size ?q) 1)
     )
)

;; due to queue-write

(:action increase-queue-tail1
   :parameters (?q - queue ?qt - queuetype ?qs1 ?qs2 - queue-state)
   :precondition
     (and
        (queue-next ?qt ?qs1 ?qs2)
        (is-a-queue ?q ?qt)
        (queue-tail ?q ?qs1)
        (advance-tail ?q)
        (< (queue-size ?q) (queue-max ?qt))
        (= (queue-size ?q) 0))
   :effect
     (and
        (settled)
        (not (advance-tail ?q))
        (queue-tail ?q ?qs2)
        (not (queue-tail ?q ?qs1))
;;      (assign (queue-value ?q ?qs2) (queue-tail-value ?q))
        (assign (queue-msg ?q ?qs2) (queue-tail-msg ?q)  )
;;      (assign (queue-head-value ?q) (queue-tail-value ?q))
        (assign (queue-head-msg ?q) (queue-tail-msg ?q) )
        (increase (queue-size ?q) 1)
     )
)
(:action increase-queue-tail2
   :parameters (?q - queue ?qt - queuetype ?qs1 ?qs2 - queue-state)
   :precondition
     (and
        (queue-next ?qt ?qs1 ?qs2)
        (is-a-queue ?q ?qt)
        (queue-tail ?q ?qs1)
        (advance-tail ?q)
        (< (queue-size ?q) (queue-max ?qt))
        (> (queue-size ?q) 0)
   )
   :effect
      (and
         (settled)
         (not (advance-tail ?q))
         (queue-tail ?q ?qs2)
         (not (queue-tail ?q ?qs1))
;;       (assign (queue-value ?q ?qs2) (queue-tail-value ?q))
         (assign (queue-msg ?q ?qs2) (queue-tail-msg ?q))
         (increase (queue-size ?q) 1)
     )
   )

(:action perform-trans
   :parameters (?p - process
                ?t - transition ?s1 ?s2 - state
   )
   :precondition
      (and
         (settled)
         (trans ?t ?s1 ?s2)
         (ok ?p ?t)
         (at-process ?p ?s1)
      )
   :effect
      (and
         (at-process ?p ?s2)
         (not (at-process ?p ?s1))
         (not (ok ?p ?t))
      )
)
(:action activate-trans
   :parameters (?p - process
                ?t - transition ?s1 ?s2 - state
   )
   :precondition
      (and
         (settled)
         (trans ?t ?s1 ?s2)
         (at-process ?p ?s1)
      )
   :effect
      (and
         (activate ?p ?t)
      )
)
)
```

Next we give the problem specific file.

```
(define (problem phil4.fsm)
(:domain protocol)
(:objects
          philosopher-0
          philosopher-1
          philosopher-2
          philosopher-3
           - process
          forks[0]
          forks[1]
          forks[2]
          forks[3]
           - queue
          queue-1
           - queuetype
          qs-0
           - queue-state
          philosopher
           - proctype
          state-1
          state-6
          state-3
          state-4
          state-5
           - state
          forks[-pid]!fork
          forks[-pid]?fork
          forks[__-pid+1_%4_]?fork
          forks[__-pid+1_%4_]!fork
           - transition
)
(:init
  (queue-next queue-1 qs-0 qs-0)
  (= (queue-max queue-1) 1)
  (at-process philosopher-0 state-1)
  (at-process philosopher-1 state-1)
  (at-process philosopher-2 state-1)
  (at-process philosopher-3 state-1)
  (is-a-queue forks[0] queue-1)
  (settled)
  (queue-head forks[0] qs-0)
  (queue-tail forks[0] qs-0)
  (= (queue-head-msg forks[0]) -1)
  (= (queue-size forks[0]) 0)
  (is-a-queue forks[1] queue-1)
  (settled)
  (queue-head forks[1] qs-0)
  (queue-tail forks[1] qs-0)
  (= (queue-head-msg forks[1]) -1)
  (= (queue-size forks[1]) 0)
  (is-a-queue forks[2] queue-1)
  (settled)
  (queue-head forks[2] qs-0)
  (queue-tail forks[2] qs-0)
  (= (queue-head-msg forks[2]) -1)
  (= (queue-size forks[2]) 0)
  (is-a-queue forks[3] queue-1)
  (settled)
  (queue-head forks[3] qs-0)
  (queue-tail forks[3] qs-0)
  (= (queue-head-msg forks[3]) -1)
  (= (queue-size forks[3]) 0)
  (writes philosopher-0 forks[0] forks[-pid]!fork)
  (= (trans-msg forks[-pid]!fork) 0) ;; fork
  (reads philosopher-0 forks[0] forks[-pid]?fork)
  (= (trans-msg forks[-pid]?fork) 0) ;; fork
  (reads philosopher-0 forks[1] forks[__-pid+1_%4_]?fork)
  (= (trans-msg forks[__-pid+1_%4_]?fork) 0) ;; fork
  (writes philosopher-0 forks[0] forks[-pid]!fork)
  (= (trans-msg forks[-pid]!fork) 0) ;; fork
  (writes philosopher-0 forks[1] forks[__-pid+1_%4_]!fork)
  (= (trans-msg forks[__-pid+1_%4_]!fork) 0) ;; fork
  (writes philosopher-1 forks[1] forks[-pid]!fork)
  (= (trans-msg forks[-pid]!fork) 0) ;; fork
  (reads philosopher-1 forks[1] forks[-pid]?fork)
  (= (trans-msg forks[-pid]?fork) 0) ;; fork
  (reads philosopher-1 forks[2] forks[__-pid+1_%4_]?fork)
  (= (trans-msg forks[__-pid+1_%4_]?fork) 0) ;; fork
  (writes philosopher-1 forks[1] forks[-pid]!fork)
  (= (trans-msg forks[-pid]!fork) 0) ;; fork
  (writes philosopher-1 forks[2] forks[__-pid+1_%4_]!fork)
  (= (trans-msg forks[__-pid+1_%4_]!fork) 0) ;; fork
  (writes philosopher-2 forks[2] forks[-pid]!fork)
  (= (trans-msg forks[-pid]!fork) 0) ;; fork
  (reads philosopher-2 forks[2] forks[-pid]?fork)
  (= (trans-msg forks[-pid]?fork) 0) ;; fork
  (reads philosopher-2 forks[3] forks[__-pid+1_%4_]?fork)
  (= (trans-msg forks[__-pid+1_%4_]?fork) 0) ;; fork
  (writes philosopher-2 forks[2] forks[-pid]!fork)
  (= (trans-msg forks[-pid]!fork) 0) ;; fork
  (writes philosopher-2 forks[3] forks[__-pid+1_%4_]!fork)
  (= (trans-msg forks[__-pid+1_%4_]!fork) 0) ;; fork
  (writes philosopher-3 forks[3] forks[-pid]!fork)
  (= (trans-msg forks[-pid]!fork) 0) ;; fork
  (reads philosopher-3 forks[3] forks[-pid]?fork)
  (= (trans-msg forks[-pid]?fork) 0) ;; fork
  (reads philosopher-3 forks[0] forks[__-pid+1_%4_]?fork)
  (= (trans-msg forks[__-pid+1_%4_]?fork) 0) ;; fork
  (writes philosopher-3 forks[3] forks[-pid]!fork)
  (= (trans-msg forks[-pid]!fork) 0) ;; fork
  (writes philosopher-3 forks[0] forks[__-pid+1_%4_]!fork)
  (= (trans-msg forks[__-pid+1_%4_]!fork) 0) ;; fork
  (trans forks[-pid]!fork state-1 state-6)
  (trans forks[-pid]?fork state-6 state-3)
  (trans forks[__-pid+1_%4_]?fork state-3 state-4)
  (trans forks[-pid]!fork state-4 state-5)
  (trans forks[__-pid+1_%4_]!fork state-5 state-6)
)
(:goal
 (and
  (at-process philosopher-0 state-3)
  (at-process philosopher-1 state-3)
  (at-process philosopher-2 state-3)
  (at-process philosopher-3 state-3)
  (settled)
 )
)
(:metric minimize total-time) )
```