Mapping Temporal Planning Constraints into Timed Automata

Lina Khatib[1], Nicola Muscettola, and Klaus Havelund[1]
[1] Kestrel Technology
NASA Ames Research Center
Moffett Field, CA, 94035, USA
{lina,mus,havelund}@ptolemy.arc.nasa.gov

Keywords: Time in AI, Temporal Representation and Constraints, Timed Automata, Planning, Scheduling, Model Checking, Validation.

Abstract

Planning and Model Checking are similar in concept. They both deal with reaching a goal state from an initial state by applying specified rules that allow for the transition from one state to another. Exploring the relationship between them is an interesting new research area. We are interested in planning frameworks that combine both planning and scheduling. For that, we focus our attention on real time model checking. As a first step, we developed a mapping from planning domain models into timed automata. Since timed automata are the representation structure of real-time model checkers, we are able to exploit what model checking has to offer for planning domains.

In this paper we present the mapping algorithm, which involves translating temporal specifications into timed automata, and list some of the planning domain questions someone can answer by using model checking.

1 Introduction

Planning and Model Checking are similar in concept. They both deal with reaching a goal state from an initial state by applying specified rules that allow for the transition from one state to another. Exploring the relationship, and synergies, between them is an interesting new research area. Related work in this area is in the use of Model Checking to do Planning [5][6][7] and in the use of Model Checking to Validate Planning Domain Models[13]. However, none of these works, to our knowledge, deals with temporal specifications other than simple sequences. Due to the major role of temporal specifications in planning domains, specifically when scheduling is part of it, we see it necessary to include them in our framework.

Our work is within the context of the HSTS, the planner and scheduler of the remote agent autonomous control system deployed in Deep Space One (DS1)[9].

Among existing model checkers, we choose UPPAAL because it can represent time (section 3), and is comparable to HSTS in terms of representation and search since they are both constraint-based systems. Furthermore, UPPAAL has been successfully used to model scheduling problems [3][4]. These problems are related to ours but they differ in having no planning component and dealing with point-based, rather than interval-based, temporal specifications.

Thus, the significance of our work is in the inclusion of temporal specifications when applying model checking to assist in validating the domain specifications as well as the planning engine. For the purpose of this paper, we focus our attention on the mapping from HSTS temporal specifications into UPPAAL timed automata. The challenge is in mapping *interval-based* temporal relations into *point-based* temporal model.

We start by giving a description of the HSTS temporal specifications and UPPAAL timed automata. After that, we introduce the algorithm for mapping HSTS domain models into UPPAAL models and we present samples of properties one may want to verify.

2 HSTS

HSTS, Heuristic Scheduling Testbed System, is a general constraint-based planner and scheduler. It is also one of four components that constitutes the Remote Agent Autonomous system which was used for the Remote Agent Experiment (RAX), in May of 1999, for autonomous control of the Deep Space 1 spacecraft [9].

HSTS consists of a planning engine that takes a domain model, in addition to a goal, as input and produces a complete plan. A planning domain model is a description of the planning domain given as a set of objects and constraints.

The produced plan achieves the specified goal and satisfies the constraints in the domain model [12].

2.1 HSTS Model

An HSTS plan model is specified in an object-oriented language called DDL (Domain Description Language). It is based on a collection of objects that belong to different classes. Each object is a collection of state variables (also known as timelines). At any time, a state variable is in one, and only, one state that is identified by a predicate. Tokens are used to represent ”spans”, or intervals, of time over which a state variable is in a certain state. Predicates may be associated with durations that indicate a minimum and a maximum allowed for their token spans (i.e., unary constraints).

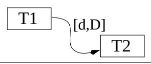
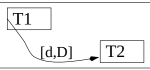
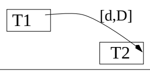
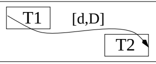
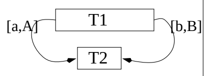
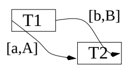
Temporal Relation	Inverse Temporal Relation	Topology
T1 before (d,D) T2 d=D=0, T1 meets T2	T2 after (d,D) T1 d=D=0, T2 met_by T1	
T1 starts_before (d,D) T2 d=D=0, T1 starts T2	T2 starts_after (d,D) T1	
T1 ends_before (d,D) T2 d=D=0, T1 ends T2	T2 ends_after (d,D) T1	
T1 starts_before_end (d,D) T2	T2 ends_after_start (d,D) T1	
T1 contains ((a A)(b B)) T2 a=A=b=B=0, T1 equals T2	T2 contained_by ((a A)(b B)) T1	
T1 parallels ((a A)(b B)) T2	T2 paralleled_by ((a A)(b B)) T1	

Figure 1: HSTS interval-based temporal relations

A set of compatibilities between predicates is specified. The compatibilities are binary temporal constraints which may involve durations between end points of predicate tokens. HSTS allows for 17 temporal relations to be used in specifying compatibilities (see Figure 1). These are interval-based relations and, although different, they are adopted from Allen’s 13 relations [1]. Each compatibility is defined in the form of a master, the constrained predicate, and servers, the constraining predicates. For example, ”pred1 meets pred2” indicates that the end of any token for the master pred1 should coincide with the start of a token for the server pred2; and ”pred1 before[3,5] pred3” indicates that the end of any token for pred1 should precede the start of a token for pred3 with a temporal distance between 3 and 5. In addition, compatibilities may be structured in hierarchical form using AND/OR trees where a leaf is one of the basic temporal relations illustrated above. For example, ”pred1 AND (meets pred2) (before[3,5] pred3)” indicates that both relations should be satisfied while ”pred1 OR (meets pred2) (before[3,5] pred3)” indicates that at least one of the relations has to be satisfied. The following example will be used for illustration throughout the paper.

Example (Rover and Rock) Figure 2 shows an HSTS domain model, in abstract syntax, that describes the domain of a Rover that is to collect samples of Rocks. In this example, there are two objects, the Rover and the Rock, each of which consists of a single state variable. The Rover’s state variable has a value domain of size 4 which includes atS, gotoRock, getRock, and gotoS (where “S” stands for Spacecraft). The Rock’s state variable has a value domain of size 2 which includes atL and withRover (“L” is assumed to be the location of the Rock). Each predicate is associated with a set of compatibilities (constraints). We choose to explain the compatibilities on Rover.getRock, for the purpose of illustration. A token representing the

State Variables: Rover, Rock
Rover predicates: atS, gotoRock, getRock, gotoS
Rock predicates: atL, withRover

Compats:
```
1. Rover.getRock   dur[3,9]
      AND
        met_by Rover.gotoRock
        meets Rock.withRover
        OR
           meets Rover.gotoS
           meets Rover.gotoRock
2. Rover.atS   dur[0,10]
      AND
        met_by Rover.gotoS
        meets Rover.gotoRock
3. Rover.gotoRock   dur[5,20]
      AND
        OR
           met_by Rover.atS
           met_by Rover.getRock
        meets Rover.getRock
4. Rover.gotoS
      AND
        met_by Rover.getRock
        meets Rover.atS
5. Rock.atL
      meets Rock.withRover
6. Rock.withRover
      met_by Rover.getRock
```

Figure 2: DDL Model for the Rover and Rock Example

predicate Rover.getRock should have a duration no less than 3 and no more than 9. It also have to be preceded immediately by Rover.gotoRock and followed immediately by Rock.withRover. The last compatibility indicates that Rover.getRock should be followed immediately by either Rover.gotoS or Rover.gotoRock (to pickup another rock).

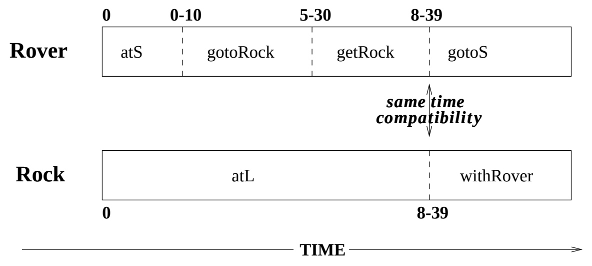


Figure 3: An HSTS plan for the goal of Rock.withRover given the initial state (Rover.atS,Rock,atL). Dashed lines indicate token boundaries. The numbers at boundaries indicated ranges of earliest and latest execution times. A constraint link is attached between the end of Rover.getRock and the start of Rock.withRover to insure the satisfaction of their equality constraint.

Figure 3 shows a plan generated by HSTS for a given goal. In this example, the initial state is: Rover.atS and Rock.atL and the goal is to have Rock.withRover. The returned plan for Rover is the sequence atS, gotoRock, getRock, and gotoS where the allowed span for each of these tokens is as specified in the duration constraints of their compatibility (e.g., getRock token is between 3 and 9 time units). As a result, the goal of Rock.withRover may be satisfied (executed) in 8 to 39 time units.

3 UPPAAL

UPPAAL, an acronym based on the joined work of UPPsala and AALborg universities, is a tool box for modeling, simulation, and verification of real-time systems. The simulator is used for interactive analysis of system behavior during early design stages while the verifier, which is a model-checker, covers the exhaustive dynamic behavior of the system for proving safety and bounded liveness properties. The verifier, which is a symbolic model checker, is implemented using sophisticated constraint-solving techniques where efficiency and optimization are emphasized. Space reduction is accomplished by both local and global reduction. The local reduction involves reducing the amount of space a symbolic state occupies and is accomplished by the compact representation of Difference Bounded Matrix (DBM) for clock constraints. The global reduction involves reducing the number of states to remember during a course of reachability analysis [14, 10, 11].

A UPPAAL model consists of a set of timed automata, a set of clocks, global variables, and synchronizing channels. A node in an automaton may be associated with an invariant, which is a set of clock constraints given in the form of inequalities that should be true while at the node, for enforcing transitions out of the node. An arc may be associated with guards, which are constraints on clocks and global variables that have to be true for enabling the transition, for controlling when this transition can be taken. On any transition, local clocks may get reset and global variables may get re-assigned. A trace in UPPAAL is a sequence of states, each of which containing a complete specification of a node from each automata, such that each state is the result of a valid transition from the previous state. More details on timed-automata may be found in [2] and more details on UPPAAL basics may be found in [10].

4 UPPAAL for HSTS

Figure 4 shows the overall structures of UPPAAL (represented as Model Checking) and HSTS (represented as Planning). There is an apparent similarity between their compo-

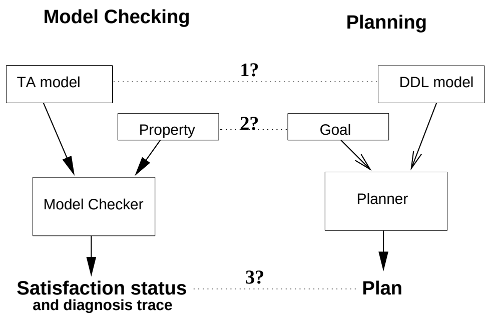


Figure 4: Model Checking and Planning Structures. The dotted arrows represent possible component mappings.

nents. Model checking takes a domain model and a property as input and produces the truth value of the property in addition to a diagnosis trace. Planning takes a domain model and a goal as input and produces a complete plan that satisfies the goal. On the other hand, the representation and reasoning techniques for their components are different. Due to structural similarity of UPPAAL and HSTS, a cross fertilization among their components may be possible. Also, due to the differences in their implemented techniques, this may be fruitful. Our research at this time is to investigate the benefit of using UPPAAL reasoning engine to validate HSTS domain models as well as validating HSTS reasoning engine. The first step, which is the focus of this paper, is to find a mapping from HSTS domain models into UPPAAL (Figure 4, dotted line with label 1?, right to left). Then, a set of properties should be carefully constructed and checked. Since model checking formalisms cannot easily represent all aspects of HSTS domain models, we restrict our mapping to a *subset* of the HSTS Domain Description Lan-

```
DDL2UPPAAL main()
1. Build_Init_Automata() ;
      for each State Variable,
          add an Automaton with a dedicated local clock
      for each predicate,
          add a node in the corresponding automaton
      for each node,
          reset the local clock on the outgoing arc
2. Add_Compatibilities();
      for each compatibility on the predicate corresponding
          to a node P,
          for max duration constraint,
              add invariant on P
          for min duration constraint,
              add a guard on the outgoing arc from P
          Process the AND/OR compatibility tree
              if (root = "AND") process AND-subtree(root)
              elseif (root = "OR") process OR-subtree(root)
              else process simple-Temporal-Relation(root)
```

Figure 5: ddl2uppaal: An algorithm for mapping HSTS domain models into UPPAAL

guage. Nevertheless, this subset is representative enough for our current research and to be extended as needed.

4.1 Mapping HSTS models into UPPAAL

An algorithm for mapping HSTS plan models into UPPAAL models, which is called ddl2uppaal, is presented in Figure 5. Each state variable is represented as a UPPAAL automaton where each value (predicate) of the state variable is represented as a node. Transitions of an automaton represent value ordering constraints of the corresponding state variable which are specified using the *meet* and *met_by* relations. Duration constraints are translated into invariants and guards of local clocks. Temporal relation constraints are implemented through communication channels. In general, constraints on the starting point of a predicate are mapped into a chain of conditional *incoming* arcs into its node. Similarly, constraints on the end point of a predicate are mapped into a chain of conditional *outgoing* arcs from its node. The nodes separating these conditional arcs are declared to be *committed*, which means they have zero durations.

Before we present the details of handling general temporal specifications, we will illustrate how the mapping algorithm works on a complete example that contains only simple, non-durational, point-based temporal compatibilities. This is for the purpose of giving the reader the feel for a complete mapping and results. For that, we apply ddl2uppaal on the the Rover and Rock specification and show the results in Figure 6. Studying Rover.getRock node, we find the duration constraint represented as the c1<=9 invariant and the c1>= 3 guard on the outgoing arc. The constraint of met_by Rover.gotoRock is represented by the incoming arc. The constraint of (meets Rover.gotoS **OR** meets Rover.gotoRock) is represented by branching outgoing arc. Finally, the constraint of meets Rock.withRover is expressed via the label 'ch1?' on its outgoing arc, which indicates a need for synchronization with a transition labeled with 'ch1!'[1]. This transition is the incoming arc to Rock.withRover. In the following, we give details on mapping durations and AND/OR connectors. We will, however, omit details related to distinguishing between *masters* and *servants* in the temporal specifications as this is beyond the scope of this paper.

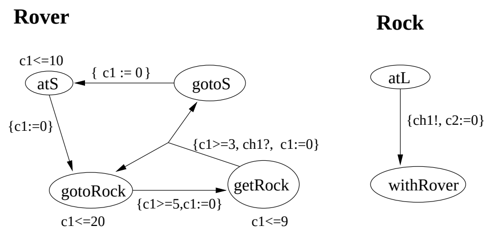


Figure 6: UPPAAL model of the Rover and Rock example. c1 is the local clock of Rover and c2 is the local clock of Rock.

4.1.1 Mapping durations of temporal relations

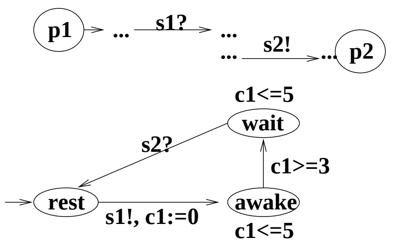


Figure 7: Mapping "p1 before[3,5] p2" into UPPAAL. Showing: part of the post conditions of p1, part of the pre-conditions of p2, and the Timer automaton used to capture the end of p1, wait for 3 to 5 time units, then capture the start of p2.

Direct synchronizations is suitable for duration_less relations as we have demonstrated above for the *meets* and *met_by* relations. However, direct synchronization falls short for representing relations that involve durations such as *before*. To handle relations with durations, we have introduced the the *Timers* which are timed automata that play the role of mediators in the synchronization process. A Timer

[1]The '!' and '?' are used to indicate synchronized transitions and their roles are fully interchangeable. However, there is the convention of using '!' to mean "send" and '?' to mean 'receive'. Accordingly, we are associating '!' with the servant and '?' with the master of the corresponding temporal relations.

receives a signal from the event that is supposed to happen first, reset its clock, and wait for a signal from the event that is supposed to happen next within specified time duration. Figure 7 shows a Timer that is used to support the temporal relation "p1 before[3,5] p2".

4.1.2 Mapping double_sided temporal relations

In all the above, we were concentrating on relations that involve only one pair of end points. For temporal relations that involves two pairs of end points, such as *equal* and *contains*, we need to handle double synchronizations. For duration_less relations, such as *equal*, direct double synchronization at both ends of the events is needed. For relations with durations, the Timers are to perform the synchronization between the first pair of events, same as for single_pair relations, followed by performing the synchronization between the second pair. Figure 8 shows a Timer that is used to support the temporal relation "p1 contains[3,5][1,6] p2".

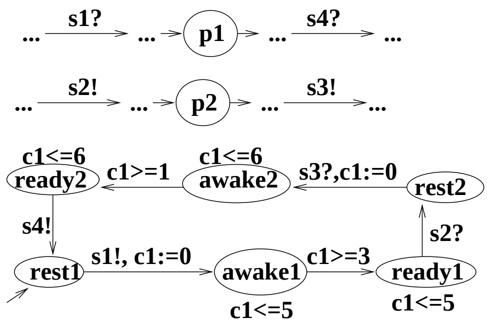


Figure 8: Mapping "p1 contains[3,5][1,6] p2" into UPPAAL. Showing: part of the pre and post conditions of p1, part of the pre and post conditions of p2, and the Timer automaton used to capture the start of p1, wait for 3 to 5 time units, capture the start of p2, rest until capturing the end of p2, wait for 1 to 6 time unites, and then capture the end of p1.

4.1.3 Mapping AND/OR in temporal specifications

The AND for temporal compatibility specifications means that all conditions have to be satisfied. From the above, we assume the reader has become familiar with how simple temporal specification is represented. Representing the AND of several temporal constraints translates into chaining all the pre-conditions and post-conditions of all involved constraints. Figure 9 illustrates how the AND is mapped. On the other hand, the OR means that at least one of the conditions has to be satisfied. For that, we combine all the pre-conditions and the post-conditions of the involved specifications in a bundle-like, or alternative branches, pre-condition and post-condition where only one

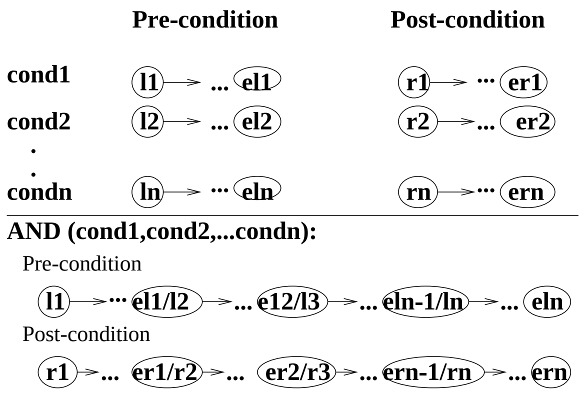


Figure 9: Mapping AND into UPPAAL. Nodes with two names such as 'el1/l2' indicates a merger of the two corresponding nodes.

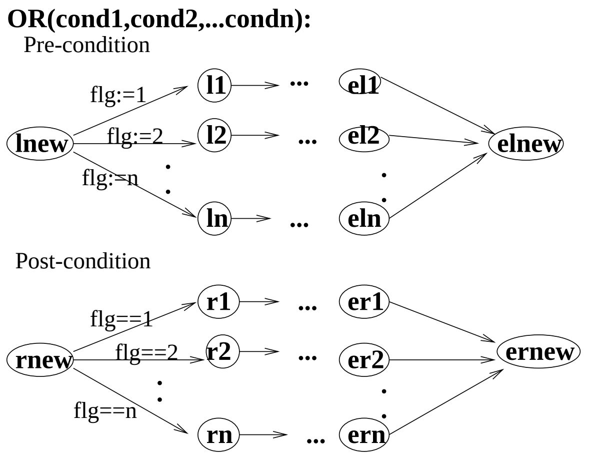


Figure 10: Mapping OR into UPPAAL. This figure builds on the conditional representation of the OR elements given in Figure 9. Two new nodes are introduced to collect the starts and ends of each pre-conditional chain. Same for the post-conditional chains. Flags are added for matching the chosen pre-condition with its post-condition.

of them has to be followed. Figure 10 illustrates the translation of the OR. Notice the use of flags that will force a specific path to be followed on the exit from the concerned node when its corresponding branch was followed at the entrance to this node. This is done by setting the flag value at the entry to a pre-condition branch to a unique value (e.g., flg:=1) and check for the this value at the entry to the corresponding post-condition branch (e.g., flg==1).

4.2 Validation Properties

UPPAAL allows for verifying properties that concern the validation of both: HSTS domain models and HSTS planning engine.

For the validation of domain models, we are in search for properties that are useful for ensuring correctness and detecting inconsistencies and flaws in the domain specifications. One of the important properties that UPPAAL is capable of detecting is the violations of mutual exclusion of predicates, which is useful for detecting an incomplete specification of compatibilities in an HSTS domain model. For example, you may check the property of: (E<>Rover.GetRock and Status.Hazard) which reads ”there exists a trace where at some point in time the rover is picking a sample rock under hazardous condition”. If the property is satisfied, then the user might have forgotten to include the condition that prevents this from happening in the domain model specifications.

Another useful property is to check for the reachability of each predicate (from one specific initial state or from each possible initial state). Finding that certain predicate is unreachable indicates the possibility of inconsistent specifications in the HSTS domain model.

For the validation of the HSTS planning engine, we are interested in checking that the engine works correctly. One aspect of its correctness is the capability of generating a plan whenever there is one. We are also interested in checking the quality of generated plans. For this purpose, we could treat the model checker as an alternative planning engine. We map planning goals into Uppaal properties (Figure 4, dotted line with label 2?, right to left), map Uppaal traces into plans (Figure 4, dotted line with label 3?, left to right), and compare with the plans generated by the the planning engine. Details on these mappings and comparison is beyond the scope of this paper.

5 Summary

Our work tackles the problem of using Model Checking for the purpose of verifying planning systems. We presented an algorithm that maps plan models into timed automata. This involves translating interval-based temporal relations into point-based temporal model where synchronization channels and Timers were used to handle distance constraints.

After translating a plan model, properties can be checked for detecting inconsistencies and incompleteness in the model. In addition, the model checking search engine can be used as an independent problem solving mechanism for verifying the planning engine.

Since complete constraint planning models are much too complex for a complete translation into a model checking formalism, there is a need for building representative “abstract” models. We will investigate such abstraction in the near future.

We are currently working on identifying a set of verification properties that guarantee a certain degree of coverage for HSTS models and the Planning engine. We are also analyzing the benefits, and limitations, of using a model checker for HSTS verification. In addition, we are extending the ddl2uppaal algorithm to handle a larger subset of DDL.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments and Paul Pettersson and Kim Larsen for their helpful correspondence.

References

[1] J. Allen. 1983. Maintaining Knowledge About Temporal Intervals. In Readings in Knowledge Representation, Brachman, R., and Levesque, H., (editors). (SanMateo:Morgan Kaufman), 510-521.

[2] R. Alur and D. Dill. 1990. Automata for Modeling Real-Time Systems. In Proc. of the International Colloquium on Algorithms, Languages, and Programming, vol. 443 of LNCS, pp. 322-225. Springer Verlag.

[3] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, and J. Romijn. 2001. Guiding and Cost-Optimality in UPPAAL. In L. Khatib and C. Pecheur, ed., AAAI Technical Report SS-01-04: Model-Based Validation of Intelligence. American Association for Artificial Intelligence.

[4] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, and J. Romijn. 2001. Efficient Guiding towards Cost-Optimality in UPPAAL. (to appear at TACAS’2001)

[5] A. Cimatti, E. Giunchiglia, F. Giunchiglia, and P. Traverso. 1997. Planning via Model Checking: A Decision Procedure for *AR*. In S. Steeland R. Alami, editors, Proceedings of the Fourth European Conference on Planning, Lecture Notes in Artificial Intelligence, number 1348, pp 130-142. Springer-Varlag.

[6] A. Cimatti, M. Roveri, and P. Traverso. 1998. Strong planning in non-deterministic domains via model checking. In the Proceedings of the 4th International Conference on Artificial Intelligence Planning System (AIPS98), pp. 36-43. AAAI Press.

[7] M. Di Manzo, E. Giunchiglia, and S. Ruffino. 1998. Planning via model checking in deterministic domains: Preliminary report. In the Proceedings of the 8th International Conference on Artificial Intelligence: Methodology, Systems, and Applications (AIMSA98), pp. 221-229. Springer-Verlag.

[8] K. Havelund, K. G. Larsen, and A. Skou. 1999. Formal Verification of a Power Controller Using the Real-Time Model Checker UPPAAL. In the Proceedings of the 5th International AMAST Workshop on Real-Time and Probabilistic Systems.

[9] A. K. Jonsson, P. H. Morris, N. Muscettola, and K. Rajan. 1999. Planning in Interplanetary Space: Theory and Practice. American Association for Artificial Intelligence (AAAI-99)

[10] K. G. Larsen, P. Pettersson, and W. Yi. 1997. UPPAAL in a Nutshell In Springer International Journal of Software Tools for Technology Transfer 1(1+2).

[11] K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. 1997. Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction. In the Proceedings of the 18th IEEE Real-Time Systems Symposium, pages 14-24. IEEE Computer Society Press.

[12] N. Muscettola. 1994. HSTS: Integrated planning and scheduling. In M. Zweben and M. Fox, eds., Intelligent Scheduling. Morgan Kaufman. 169-212

[13] J. Penix, C. Pecheur, K. Havelund. 1998. Using Model Checking to Validate AI Planner Domain Models. In the Proceedings of the 23rd Annual Software Engineering Workshop, NASA Goddard.

[14] W. Yi, P. Pettersson, and M. Daniels. 1994. Automatic Verification of Real-Time Communicating Systems by Constraint-Solving. In Dieter Hogrefe and Stefan Leue, editors, Proceedings of the 7th International Conference on Formal Description Techniques, pages 223-238. North-Holland.