Verification of plan models using UPPAAL

Lina Khatib[1], Nicola Muscettola, and Klaus Havelund[2]

NASA Ames Research Center, MS 269-2
Moffett Field, CA 94035
[1] QSS Group, Inc. [2] RECOM Technologies
{lina,mus,havelund}@ptolemy.arc.nasa.gov

Keywords: Model Checking, Verification, Autonomy, Planning, Scheduling

Abstract. This paper describes work on the verification of HSTS, the planner and scheduler of the Remote Agent autonomous control system deployed in Deep Space 1 (DS1)[8]. The verification is done using UPPAAL, a real time model checking tool [6]. We start by motivating our work in the introduction. Then we give a brief description of HSTS and UPPAAL. After that, we give a mapping of HSTS models into UPPAAL and we present samples of plan model properties one may want to verify. Finally, we conclude with a summary.

1 Introduction

AI technologies, and specifically AI planning, facilitates the elicitation and automatic manipulation of system level constraints. However, the models used by the planner still need to be verified, i.e., it is necessary to guarantee that no unintended consequences will arise. One question that comes to mind is whether the most advanced techniques used in software verification, specifically model checking, can help.

The most used model checking formalisms, however, cannot easily represent constraints that are naturally represented by HSTS, namely continuous time and other continuous parameters. Also, the goal of HSTS is to provide an expressive language to facilitate knowledge acquisition by non AI experts (e.g., system engineers). So, HSTS models cannot be easily translated into a model checking formalism. To allow model checking algorithms to operate on HSTS models we, therefore, need a mapping between a *subset* of the HSTS Domain Description Language to a model checking formalism. An earlier attempt to analyze HSTS planner models, where no continuous parameters were considered, is described in [10].

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 applied to several verification cases of real-time systems of industrial interest [4, 3].

Some of the issues that we are interested in addressing in this research are:

1. Whether model checking techniques can address problems of the size of a realistic autonomy planning application;

2. Once we have a mapping from a planner model into a model checking formalism, what the core differences between the search method used in model checking and that used by a planner are; and
3. Lessons, if any, that planning can take from model checking and vice versa regarding representation, search control, and other aspects. Related work can be found in [1, 2].

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 [8].

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

An HSTS plan is a complete assignment of tokens for all the state variables that satisfy all compatibilities. HSTS ensures robustness of schedules by allowing for flexible temporal representations, *ranges* of durations, and disjunction of constraints [9, 5].

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.

A set of compatibilities between predicates is specified. The compatibilities are temporal constraints, which may involve durations between end points of tokens. For example, "token1 meets token2" indicates that the end of token1 should coincide with the start of token2; and "token1 before[3,5] token2" indicates that the distance between the end of token1 and the start of token2 is between 3 and 5. The compatibilities among tokens are structured in the form of a master, the constrained token, and servers, the constraining tokens, and stated in the form of AND/OR trees (see the Rover and Rock example for illustration). HSTS has a rich language for expressing temporal relation constraints which is beyond the scope of this paper. Details can be found in [9, 5].

HSTS also allows for natural and efficient handling of concurrent processes, and the modeling language is simple in its uniform representation of actions and states. The following example will be used for illustration throughout the paper.

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

Compats:

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

Fig. 1: DDL Model for the Rover and Rock Example

Example (Rover and Rock) Figure 1 shows an HSTS 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 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).

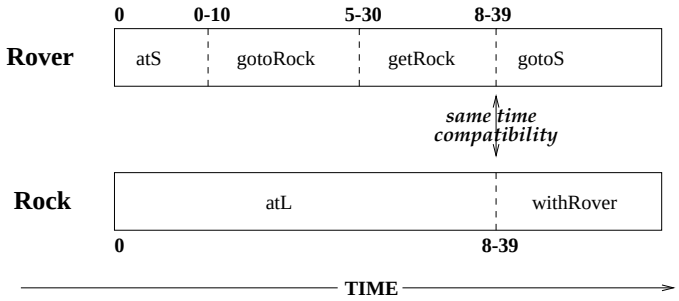

Fig. 2: An HSTS plan for the goal of Rock.withRover given the initial state (Rover.atS,Rock,atL). Dashed lines indicate token boundaries. The numbers at boundaries indicated ranges of earliest and latest execution times. A constraint link is attached between the end of Rover.getRock and the start of Rock.withRover to insure the satisfaction of their equality constraint.

Figure 2 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. The quality of generated plans is improved by interleaving planning and scheduling, rather than performing them separately.

3 UPPAAL

UPPAAL, an acronym based on a combination 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 save during a course of reachability analysis [11, 7].

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, for enforcing transitions out of the node. An arc may be associated with a guard 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.

UPPAAL had been proven to be a useful model_checking tool for many domains including distributed multimedia and power controller applications [4, 3].

4 UPPAAL for HSTS

Figure 3 shows the overall structures of UPPAAL (represented as Model Checking) and HSTS (represented as Planning). There is an apparent similarity between their components. Model checking takes a model and a property as input and produces the truth value of the property in addition to a diagnosis trace. Planning takes a 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. Table 1 summarizes the differences.

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 the UPPAAL reasoning engine to verify HSTS models as well as verifying the HSTS reasoning engine. The first step is to find a mapping from HSTS models into UPPAAL. Then, a set of properties should be carefully constructed and checked.

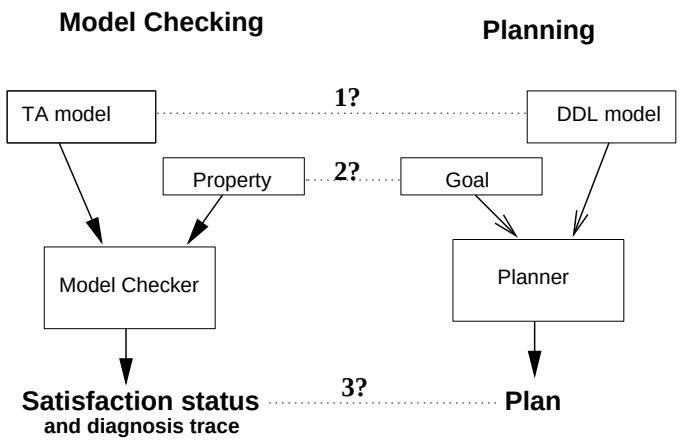

Fig. 3: Model Checking and Planning Structures. The dotted arrows represent possible component interchangeability.

	UPPAAL	HSTS
INPUT Domain Model: Problem:	 Timed Automata Check Property	 DDL Accomplish Goal
OUTPUT Solution:	 Property status + Diagnosis trace	 A Plan
INNER WORK Database: Search Techniques:	 DBM or CDD + explored states Forward Reachability + Shortest Path Reductions	 Temporal and Constraint networks Propagations + Heuristics directed backtracking

Table 1: Representation and Reasoning of UPPAAL and HSTS. DBM stands for Difference Bounded Matrices and CDD stands for Clock Difference Diagrams.

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 4. Each state variable is represented as a UPPAAL automaton where each value of the state variable is represented as a node. Transitions of an automaton represent value ordering constraints of the corresponding state variable. 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 conditional *incoming* arcs into its node. Similarly, constraints on the end point of a predicate are mapped into conditional *outgoing* arcs from its node. Instead of presenting the lower level details of the mapping algorithm, we choose to illustrate them through the example. For this purpose, we apply ddl2uppaal on the Rover and Rock specification and show the results in Figure 5. Studying Rover.getRock node, we find the duration constraint represented as the c1<=9 invariant and the c1>= 3 guard on the outgoing arc.

```
DDL2UPPAALmain()
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 a predicate corresponding to a node P
            for max duration constraint, add invariant on P
            for min duration constraint, add a guarded 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)
```

Fig. 4: ddl2uppaal: An algorithm for mapping HSTS models into UPPAAL

The constraint of metby 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!'. This transition is the incoming arc to Rock.withRover.

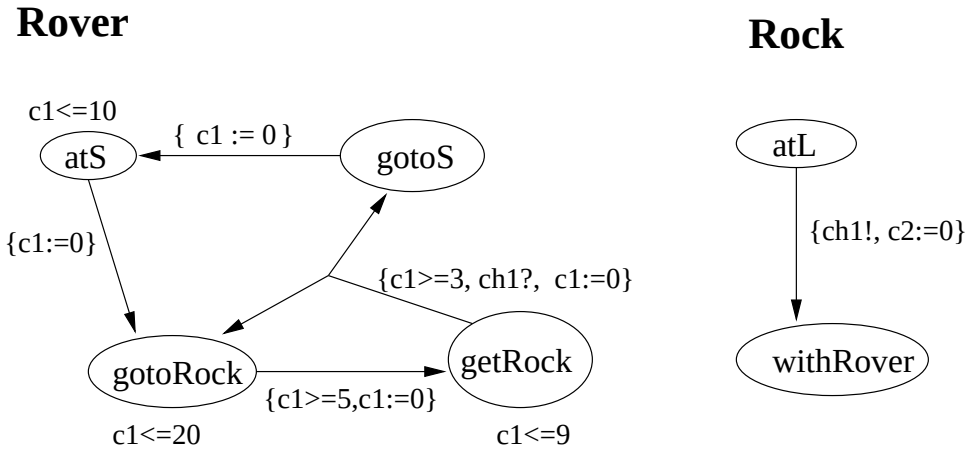

Fig. 5: UPPAAL model of the Rover and Rock example. c1 is the local clock of Rover and c2 is the local clock of Rock.

4.2 Properties for Verification

UPPAAL allows for verifying properties that are useful for ensuring correctness and detecting inconsistencies and flaws in HSTS plan models. For example, UPPAAL is capable of detecting violations of mutual exclusion properties of predicates, which is useful for detecting an incomplete specification of compatibilities in an HSTS model. Also, from checking the reachability of predicates, inconsistencies in an HSTS model may be detected.

Initial state: (Rover.atS, Rock.atL)
Property: E<>Rock.withRover

OUTPUT: PROPERTY IS SATISFIED
Diagnosis Trace:
(Rover.atS, Rock.atL)
(Rover.gotoRock, Rock.atL)
(Rover.getRock, Rock.atL)
(Rover.gotoS, Rock.withRover)

Fig. 6: Verifying Property in UPPAAL (Example).

Goals in HSTS can be mapped into properties in UPPAAL and execution traces in UPPAAL correspond to plans in HSTS. Figure 6 shows a UPPAAL property and diagnosis trace that correspond to the HSTS goal and plan of Figure 2. Note the generated symbolic states in the trace and compare them to the tokens of the plan.

Based on the above, UPPAAL is able to verify the existence of complete plans that satisfy given constraints. This can be used to verify HSTS models as well as verifying the HSTS engine.

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. The algorithm works well for translating models of limited size and complexity. 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.

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. This is possible because goals can be mapped into

properties and traces correspond to plans. We have illustrated such correspondence through an example.

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 UPPAAL team, especially Paul Pettersson, for their helpful correspondences.

References

1. 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.
2. 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.
3. 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.
4. K. Havelund, A. Skou, K. G. Larsen, and K. Lund. 1997. Formal Modeling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using UPPAAL. In the Proceedings of the 18th IEEE Real-Time Systems Symposium, pages 14-24. San Francisco, California.
5. 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)
6. 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).
7. 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.
8. Muscettola, P. P. Nayak, B. Pell, and B. William. 1998 Remote Agent: To boldly go where no AI system has gone before. Artificial Intelligence 103(1-2):5-48
9. N. Muscettola. 1994. HSTS: Integrated planning and scheduling. In M. Zweben and M. Fox, eds., Intelligent Scheduling. Morgan Kaufman. 169-212
10. 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.
11. 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.