ICAPS05

WS5

Workshop on Verification and Validation of Model-Based Planning and Scheduling Systems

Maria Fox
University of Strathclyde, UK

Allen Goldberg
Kestrel Technology, USA

Klaus Havelund
Kestrel Technology, USA

Derek Long
University of Strathclyde, UK

ICAPS 2005
Monterey, California, USA
June 6-10, 2005

CONFERENCE CO-CHAIRS:

Susanne Biundo
University of Ulm, GERMANY

Karen Myers
SRI International, USA

Kanna Rajan
NASA Ames Research Center, USA

Cover design: L.Castillo@decsai.ugr.es

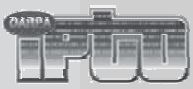
Workshop on Verification and Validation of Model-Based Planning and Scheduling Systems

Maria Fox
University of Strathclyde, UK

Allen Goldberg
Kestrel Technology, USA

Klaus Havelund
Kestrel Technology, USA

Derek Long
University of Strathclyde, UK

Workshop on Verification and Validation of Model-Based Planning and Scheduling Systems

Table of contents

Workshop on Verification and Validation of Model-Based Planning and Scheduling Systems

Preface

Planning and scheduling (P&S) systems are finding increased application in safety- and mission-critical systems that require a high level of assurance. However, tools and methodologies for verification and validation (V&V) of P&S systems have received relatively little attention. The primary goal of this workshop is to initiate an interaction between the P&S and V&V communities to identify specialized and innovative V&V tools and methodologies that can be applied to P&S. A secondary goal is simply to engage the two communities in the exploration of cross-cutting technologies.

Model-based P&S systems have unique architectural features that give rise to new V&V challenges. Most significantly, these systems consist of a planning engine that is largely stable across applications and a declaratively-specified domain model specialized to a particular application. Planners use heuristic search to compute detailed plans that achieve high level objectives stated as an input goal set. Experience has shown that most errors are in domain models, which can be inconsistent, incomplete or inaccurate models of the target domains. There are currently few tools to support the model construction process itself, and even fewer that can be used to validate the structures of the domains once they are constructed. Another challenge to V&V of P&S systems is to demonstrate that specific heuristic strategies have reliable and predictable behaviors.

Among the presented papers, some address the topic very directly, by discussing what forms of checks could be performed on plan models. Two papers address how formal methods tools such as a theorem prover and a rewriting system can be applied to model analysis. Two papers target the problem slightly differently and discuss how a real-time model checker can be used as a planning engine, hence giving an understanding of how verification and planning tools relate. The reverse direction is also represented by a paper that discusses how a planning tool can be used for verification. Other papers deal with compositional verification and runtime monitoring. A single paper addresses the synthesis of robust plans, often stated as an alternative approach to correctness. The invited talk by Steve Chien, NASA's Jet Propulsion Laboratory in Pasadena, California, USA, presents work on validating the autonomous EO-1 science agent.

Organizers

- *Maria Fox, University of Strathclyde, UK*
- *Allen Goldberg, Kestrel Technology, USA*
- *Klaus Havelund, Kestrel Technology, USA*
- *Derek Long, University of Strathclyde, UK*

Programme Committee

- *Howard Barringer, University of Manchester, UK*
- *Saddek Bensalem, Verimag, France*
- *Enrico Giunchiglia, University of Genova, Italy*
- *Tom Henzinger, Ecole Polytechnique Federale de Lausanne, Switzerland*
- *Gerard Holzmann, JPL/NASA, USA*
- *Kim Guldstrand Larsen, Aalborg University, Denmark*
- *David Musliner, Honeywell Technology Center, USA*
- *Nicola Muscettola, NASA Ames Research Center, USA*
- *Joseph Sifakis, Verimag, France*
- *Douglas Smith, Kestrel Institute, USA*
- *Carolyn Talcott, SRI, USA*
- *Brian Williams, MIT, USA*

Model Validation in Planware

Marcel Becker and Douglas R. Smith
Kestrel Technology
3260 Hillview Avenue
Palo Alto, California 94304
{becker,smith}@kestrel.edu

Introduction

Planware II is an integrated development environment for the domain of complex planning and scheduling systems. At its core is a model-based generator for planning and scheduling applications. Its design and implementation aim at supporting the entire planning and scheduling process including domain analysis and knowledge acquisition; application development and testing; and mixed-initiative, human-in-the-loop, plan and schedule computation. Based on principles of automatic software synthesis, Planware addresses the problem of maintaining the synchronization between a dynamic model describing the problem, and the corresponding system implementation. Planware automatically generates optimized and specialized planning and scheduling code from high-level models of complex problems.

Resources and tasks are uniformly modeled using a hierarchical state machine formalism that represents activities as states, and includes constructs for expressing constraints on states and transitions. A resource model represents all possible sequences of activities this resource can execute. Figure 1 shows a simple problem description with 3 resources – *Aircraft*, *Crew*, and *Airport*; and a top-level task representing a *movement requirement*. A schedule or plan will be composed of concrete activity sequences that can be generated by simulating the execution, or trace, of these state machines. For example, a sequence [*Idle, Transporting, Unloading, Returning, Idle*] represents a valid sequence of activities for the Aircraft resources. Figure 2 and 3 shows parts of the source code for the Aircraft resource.

The model-based generator analyzes the state machine models to instantiate program schemas generating concrete implementations of backtrack search and constraint propagation algorithms. Coordination between resources and tasks is achieved through the use of *services:* tasks require services, and resources provide services. Planware's scheduler generator component matches providers with requesters, and automatically generates the code necessary to verify and enforce, at schedule computation time, the service constraints imposed in the model. Planware's user interface is based on Sun's NetBeans platform and provides integrated graphic and text editors for modeling complex resource systems, automatically generating batch schedulers, and executing the generated schedulers on test data sets.

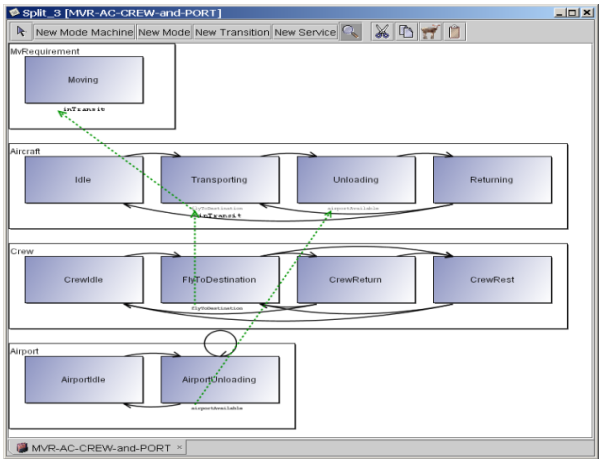


Figure 1: Planware Model Example

Developing scheduling applications and computing schedules using a model-based generator approach like the one provided by Planware creates a new set of challenges to the developer of such tools, namely establishing a connection between the high-level models describing the problem and the solution produced by the generated application – the schedule computed for a given set of input data. We assume the user modeling the problem is the final user of the generated scheduling application. The planning expert trying to solve a particular planning problem, must be able to validate, test, and debug her models without any knowledge about the intermediate software artifacts produced by the generator. All communication with the user should happen either at the model level, or at the computed solution level.

There are two main set of questions regarding V&V of Planware models:

Model Validity: Does the model correctly express the intentions of the designers? Does the modeler understand the semantics, assumptions, and limitations of the modeling formalism? Does the model respect physical and logical constraints? Does the model accurately capture

```
mode-machine Aircraft is
    constant baseLoc           :  Location
    constant fuelBurnRate      :  BurnRate
    constant maxFuelCapacity   :  Capacity
    input-variable origin, dest :  Location
    internal-variable duration  :  Time
    external-variable st, et    :  Time
end-mode-machine
```

Figure 2: State descriptor parameters for Aircraft Resource

the physical process it is trying to represent? Can the generator produce an application from the model?

Implementation Correctness: Is the executable behavior of the generated application consistent with the model? Any testing performed on the executable depends on this consistency.

This position paper presents several ways that Planware could be extended to support the validation and verification of its models.

Validity of the Model

Model-based software generation reduces the application development effort by simplifying the problem specification activity, and by hiding lower level, domain independent implementation details. In an ideal scenario, a user without any knowledge about the generation engine should be able to describe the planning problem using only the high-level modeling interface, and the generator would then automatically produce a high-quality, running application. In reality, however, the best results are achieved by users with some understanding of how models are translated into code, and how the generated code implements the scheduling algorithms. Power users can even slightly modify the models to generate more efficient or more effective implementations. This tells us that model-driven v&v requires exposing, also in a model-oriented fashion, some of the internal details of both the generator and the generated code to the modeler, and providing additional facilities for model debugging and testing.

There are 3 basic aspects of model validation:

Model Structure: This relates to the well-formedness of the model. This activity can be performed by a model compiler that verifies if the model is syntactically correct, and if the model structure is consistent so that the generator can produce an application for it.

Planware already provides several syntactical and structural checks and inform the user about the identified problems. For example, it requires that all state descriptors defined for a given resource model be initialized, and it complains if some of the defined parameters have not been initialized.

Some errors, however, like typos and type mismatches, are only identified when the generated code gets compiled. For example, if there is a type mismatch in a variable assignment, this error will be reported by the compiler. The user will be responsible to find the place in the model where this wrong statement is located.

```
mode-machine Aircraft is
    ...
    mode Idle has end-mode
    mode Loading has
        required-invariant loadCargo (st, et, ...)
    end-mode
    mode Positioning has end-mode
    mode Refueling has end-mode
    mode Deploying has
        provided-invariant deliverCargo(st, et, ...)
    end-mode
    transition from Idle to Loading
        when {} is {
        duration := loadingDuration }
    transition from Loading to Positioning
        when {} is {
        dest       :=  (if (consumedFuel() ≥
                            maxFuelCapacity)
                        then findAirRefLoc(...)
                        else targetLoc),
        airRefTrack :=  (if (consumedFuel() ≥
                            maxFuelCapacity)
                        then dest
                        else zeroLoc) }
    transition from Positioning to Refueling
        when { airRefTrack != zeroLoc } is {}
    transition from Positioning to Deploying
        when { dest = targetLoc } is {}
end-mode-machine
```

Figure 3: Modes, Transitions and Services for Aircraft Resource

There are 2 possible solutions to this problem: We can enhance the simple model compiler to check for all possible sources of structural errors, or we can enhance the generator and compiler so that when a compiler error is encountered, the error message identifies the line of code in the model responsible for generating the compilation error. Although the first solution may seem simpler, the second has the advantage of greatly facilitate the tracing and debugging of models if the connection to the model source code is maintained by the generated application.

Model Semantics: This relates to matching, or at least communicating, the assumptions and design decisions made by the developer of the generator system with the user's representation and execution requirements. The designer of the generation framework assumes some execution or system behavior model into which the description defined in the problem specification is embedded. For example, Planware implements the scheduling algorithm as an auction-based mechanism in which different resource instances bid for tasks. The sequencing of the bids for different tasks and subtasks is based on the hierarchical structure of the service tree defined in the model. The generator synthesize the code implementing the bidding mechanism for each task and resource type requesting and providing a certain service (see examples of services in Figure 3).

The schedule computed by the generated code will reflect the design decisions made by the generator developer on how to translate the models into an executable application. The main question here is how to communicate to the user these design assumptions and decisions. The proposed solution is to implement a model simulator or interpreter capable of symbolically executing the model. In Planware, this would correspond to animating the search tree used by the scheduler, and simulating the bid cre-

ation mechanism for each service requester and available service provider. This animation would expose to the user the details of the algorithms that would be implemented for a particular model.

Model Implementation: This relates to debugging and tracing the execution of the generated application on real data. One of the major problem we encounter while developing applications from models is identifying why a computed solution is different from the expected one, or, said another way, why the system fails to schedule tasks. Sometimes this is simply due to the sophisticated reasoning performed by constraint propagation algorithms – the user may not understand the subtle consequences of runtime choices available to the scheduler. On the other hand, the problem may be due to faulty input data, or to the implementation of the automatically generated algorithm, or to modeling mistakes. The simulator described above may help reduce modeling mistakes but will not completely eliminate them since they will probably not exercise all possible scenarios.

The solution is, of course, to instrument the source code, to trace the execution of the system, and to use this trace to debug and correct the problem. As developers of the generation engine, we can add trace statements to the generated code and execute the application with tracing turned on. For a model developer, however, this is not appropriate.

The modeler should be able to request the code generator to add testing and tracing statement directly in the model. Furthermore, she should also be able to see these trace statements expressed in terms of the model, and not in terms of application implementation. For example, the bidding mechanism for each service request is implemented by a chain of 10 different methods. For the modeler, an error message describing that a particular method in this chain failed is not very useful. A better message would be one that describes, for example, that a bid could not be generated for a particular service because a particular sequence of transitions needed to provide the service could not be executed. The user could then ask the system for more detailed information about the particular failure, without having to go through a long list of implementation-specific tracing statements.

The best solution would be achieved by stepping through the execution using the animation previously described. At each step, the generated application would execute a number of instrumented methods, and the execution results, as well as the trace statement would be communicated to the user using an interface based on the defined model. This would correspond to a model debugger that would filter the regular, programming language level tracing of the application and map into the debugging and symbolic execution of the model.

Verifiably Correct Code from Models

One aspect of V&V is one's degree of assurance that the executable behavior is consistent with the model. Planware II automatically generates an executable scheduler from models of the tasks and resources of the problem domain. It has no explicit assurance guarantees, but compared with manual production of similar code, (1) it is far faster (typically a few minutes for 100k LOC), and (2) it is far more likely to produce correct code (since the model-to-code transformations are general and have been tested extensively).

Here are a few approaches to providing more explicit guarantees of consistency.

1. Design by Classification (Smith 1996) - An earlier implementation, called Planware I (Blaine *et al.* 1998), used a refinement process to generate code from a simpler modeling language (see also Designware (Smith 1999)). Each refinement was expressed as a specification morphism. Planware I drew from a library of abstract, reusable refinements that embodied knowledge about global search algorithms and various data type refinements. A refinement in a library can be preverified. A refinement is applied by means of a composition operation that preserves proofs.

To make the verification guarantees of this approach more formal, suppose that specification objects are extended with explicit proof objects. The intention is that all proof obligations of the specification (e.g. consistency between axioms on a function and its definition, type consistency, and so on) are packaged together with the specification. If S is a specification and P is its corresponding proof set, let $\langle S, P\rangle$ be a spec-proof object. We can make these the objects of a category that has a composition operator (aka colimit) in a fairly straightforward way. If $m : S \rightarrow T$ is a specification morphism that transform specification S to specification T, we can extend m to a spec-proof morphism as follows:

$$m : \langle S, P\rangle \;\rightarrow\; \langle T, m(P) \cup \mathit{PrfObl}(T) \cup \mathit{PrfObl}(m)\rangle$$

where $PrfObl$ is a function that maps specs and morphisms to proofs of their respective proof obligations. For simplicity of presentation, we are assuming a complete logic. The Specware system (Kes 2003) has an automatic generator of proof obligations for specs and morphisms and uses the Snark prover to produce proofs.

The advantage of use this extended category of spec-proofs is to support the simultaneous development of programs (e.g. planners and schedulers) from models/specs together with explicit proofs of consistency between model and code. A certifying authority can examine the spec-proofs and check for themselves whether the putative proofs are in fact proofs and that they establish the desired consistency results. It is much easier to check proofs than to generate proofs and there is no easier time to generate proofs of the consistency of models and code than at code generation time.

2. Refinement Generators – KIDS (Smith 1990) and Planware I also used a small library of expression optimizers that were implemented as metaprograms. As metaprograms, they can in principle produce arbitrary transformations of an intermediate design. How can we get verification help in this context? The basic approach is, again, to work in the spec-proof category discussed above. We treat each refinement generator as a transformer not only of the source specification, but also of its proof set.

More formally, if R is a refinement generator that applies to specification S, then $R(S)$ is a morphism $m : S \rightarrow T$. Then the work is to extend R so that it generates a spec-proof morphism from $\langle S, P \rangle$ to $\langle T, Q \rangle$ where Q is the proof-set of T. For the refinement generators in KIDS and Planware I this work would be fairly straightforward since a constructive prover does most of the work in the generator, so proofs are available.

Concluding Remarks

Early work on model-based code generation was mainly focused on rich design spaces (showing that a range of applications could be generated) and scaling up (showing that useful work could be done). As generative techniques move toward production, it becomes more important to integrate with existing processes for code certification and assurance. A key advantage of Kestrel's category-theoretical and logical foundations is its intrinsic ability to produce and carry along assurance/certification documentation as part of normal development and evolution. This should result in cost savings since the information needed for assurance and certification is available during development.

References

Blaine, L.; Gilham, L.; Liu, J.; Smith, D. R.; ; and Westfold, S. 1998. Planware — domain-specific synthesis of high-performance schedulers. In *Proceedings of the Thirteenth Automated Software Engineering Conference*, 270–280. IEEE Computer Society Press.

Kestrel Institute. 2003. *Specware System and documentation*. http://www.specware.org/.

Smith, D. 1990. KIDS: A semiautomatic program development system. *IEEE Transaction of Software Engineering* 16(9):1024–43.

Smith, D. R. 1996. Toward a classification approach to design. In *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology, AMAST'96*, volume LNCS 1101, 62–84. Springer-Verlag.

Smith, D. R. 1999. Mechanizing the development of software. In Broy, M., and Steinbrueggen, R., eds., *Calculational System Design, Proceedings of the NATO Advanced Study Institute*. IOS Press, Amsterdam. 251–292.

Inspection and Verification of Domain Models with PlanWorks and Aver

Tania Bedrax-Weiss[*†] and **Jeremy Frank**[‡] and **Michael Iatauro**[§] and **Conor McGann**[¶]
Computational Sciences Division
NASA Ames Research Center, MS 269-2
miatauro@email.arc.nasa.gov
Moffet Field, CA 94035

Introduction and Motivation

When developing a domain model, it seems natural to bring the traditional informal tools of inspection and verification, debuggers and automated test suites, to bear upon the problems that will inevitably arise. Debuggers that allow inspection of registers and memory and stepwise execution have been a staple of software development of all sorts from the very beginning. Automated testing has repeatedly proven its considerable worth, to the extent that an entire design philosophy (Test Driven Development) has been developed around the writing of tests.

Unfortunately, while not entirely without their uses, the limitations of these tools and the nature of the complexity of models and the underlying planning systems make the diagnosis of certain classes of problems and the verification of their solutions difficult or impossible.

Debuggers provide a good local view of executing code, allowing a fine-grained look at algorithms and data. This view is, however, usually only at the level of the current scope in the implementation language, and the data-inspection capabilities of most debuggers usually consist of on-line print statements. More modern graphical debuggers offer a sort of tree view of data structures, but even this is too low-level and is often inappropriate for the kinds of structures created by planning systems. For instance, goal or constraint networks are at best awkward when visualized as trees. Any any non-structural link between data structures, as through a lookup table, isn't captured at all. Further, while debuggers have powerful breakpointing facilities that are suitable for finding specific algorithmic errors, they have little use in the diagnosis of modeling errors.

Automated testing can take several forms, few of them convenient. Writing tests explicitly in code can require deep knowledge of the system in which the model is going to be executed, are therefore not portable to other planning systems, even closely related ones, and will break with changes in the underlying system or the model, adding to the required maintenance work. Tests written at this level will also have to be much more verbose than those written at a higher level of abstraction.

EUROPA(Frank & Jónsson 2003), the predecessor to EUROPA$_2$, as part of its test suite, captured the output of the final plan and compared it against a known-good output. This proved to be quite brittle, since changes to the planner, plan database, model, or heuristics could dramatically alter the output without implying a bug, and hand-verifying output for the new known-good was both tedious and labor intensive. The known-good method also suffers from a limitation of scope—it looks only at the output, and in the case of planning and model rule execution, the path to the final plan is at least as important.

Another verification technique that EUROPA employed was an examination of the final constraint network to ensure compliance with the rules of the model. While suffering from the output-scope problem, it also only detects errors in the code that executes model rules which, while significant, is only one of a plurality of components. This technique also only checks the constraint network's compliance with the model, not the executed model's compliance with the intended model.

Clearly, there is a gap between what traditional tools can provide and what is necessary to debug and test planning systems efficiently. To this end, we have built two tools: *PlanWorks*, a visualization and query tool for plan inspection and *Aver*, a language for the specification of automated tests.

This paper is organized as follows. We first describe some fundamentals of the EUROPA$_2$ constraint-based planning system. We then describe our debugging tool, PlanWorks. We cover in light detail its views and query tools. We then describe our test specification language, Aver. We describe its method of asserting properties of plans with queries and boolean comparisons. We then describe the use of these tools to verify the description of a sample problem domain and instance, the pipesworld, in which we cover test composition, a test failure, its investigation with PlanWorks, and confirmation of the fix with both PlanWorks and the automated test. Finally, we discuss future work for both tools.

The EUROPA$_2$ Paradigm

The context in which these tools have been developed is EUROPA$_2$, which provides plan database services that en-

[*] Authors listed in alphabetical order.
[†] QSS Group, Inc.
[‡] NASA
[§] QSS Group, Inc.
[¶] QSS Group, Inc.

able the integration of automated planning into a wide variety of applications.

A detailed discussion of the EUROPA$_2$ paradigm is beyond the scope of this paper, but a brief discussion is included here. A *plan* is a complete enumeration of the states necessary to achieve a set of goal states from a set of initial states which satisfies the constraints of a planning domain and problem instance. In EUROPA$_2$, states are represented as *predicates*, each of which has a name, start time, end time, duration, and zero or more parameters. Each instance of a predicate in a plan is represented by a *token* and the parameters, timepoints, and duration of the predicate are represented by *variables*. Predicates are associated with *classes* that represent types of *objects*, with specializations like *timelines*, which require that their sequences of states be totally ordered, or *resources*, which allow concurrent states, but require that rules about consumption and production rates and resource levels be obeyed. During planning each token is assigned to an object. *Domain rules* are assertions that if a predicate P is in the plan, then other predicates Q_i must also be in a plan and are related to P by *constraints* among the variables of the predicates. Domain rules may also assert that resources are impacted by predicates; resource impacts are called *transactions* and also have variables that represent them.

It is important to emphasize that EUROPA$_2$ does not implement any planning algorithm; rather, it provides services that support different planning algorithms according to the application, like maintaining plan state and evaluating plan consistency. The EUROPA$_2$ plan database maintains the current plan state and an external planner performs the search by resolving flaws through variable restrictions, which amount to operations on the plan database. As such, it can be used to support progression planners, regression planners, sequential or causal link planners, and so on. To enable this generality, EUROPA$_2$ distinguishes between *free* tokens (consequences of rules that haven't been inserted into plans), *active* tokens, and *merged* tokens. Planners can insert free tokens into plans, making them active, or co-designate free tokens with active tokens, making them merged.

PlanWorks

Introduction

PlanWorks is a browse-based system for debugging constraint-based planning and scheduling systems. It assumes a strong transaction model of the entire planning process, including adding and removing parts of the constraint network, variable assignment, and constraint propagation. A planner logs transactions and plan states for importation into a relational database that is tailored to support queries for a variety of components. *Visualization* components consist of specialized views to display different forms of data (e.g. constraints, activities, resources, and causal links). Each view allows user customization in order to display only the most relevant information. Inter-view navigation features allow users to rapidly exchange views to examine the trace of the process from different perspectives. *Transaction query* mechanisms allow users access to the logged transactions to visualize activities across the entire planning process.

PlanWorks is implemented in Java and employs a MySQL relational database back-end. It can be used either online while planning is performed or offline after capturing the entire planning process. Furthermore, PlanWorks is an open system allowing for extensions to the transaction model to capture new planner algorithms, different classes of entity, or novel heuristics. While PlanWorks was specifically developed for EUROPA$_2$, the underlying principles behind PlanWorks make it useful for many constraint-based planning systems.

Views

The first view the user is presented with is an overview of the entire planning sequence, an inverted histogram of the counts of the tokens, variables, and constraints in the plan at each step. Moving the mouse over a histogram element will reveal the the number of elements of a particular type at that step. At a glance, the user sees how the plan's size evolved throughout planning and can see patters (such as thrashing in a chronological backtracking algorithm, or local optimum in a local search planner). An indicator above each histogram bar indicates whether the data for that step is in the file system or in the PlanWorks database.

The *Timeline View* is designed to show the sequence of predicates on a timeline. Since tokens can be co-designated, the Timeline View shows the number of co-designated tokens that each token supports.

Because the EUROPA$_2$ structure can be treated as a directed graph (Objects→Tokens→Variables↔Constraints), it is useful to visualize the entire graph or certain subgraphs. Of particular interest are the causal tree, or *token network*, and the *constraint network*. All PlanWorks graph views use an incremental expansion method for navigation. Clicking on a node will expand all of its arcs and place in the view any connected nodes not already visible. Clicking on such an "open" node closes it, and will cause any entities to which it is related that are not connected to other open entities to be removed from the view. To assist navigation, the graph views provide "find by key" and "find path" to locate a particular entity in the graph and find a path between two entities, respectively.

The Token Network View visualizes the causal chain resulting from planner decisions and model rules. Initially only the root tokens—those created in the initial state—are visible. Expanding a token node causes the appearance of *rule* nodes, which represent the model rules that executed because of the presence of the parent token in the plan. Rule nodes can be expanded to see the text of the rule as written in the model as well as to see the tokens created through application of the rule.

The Constraint Network View begins with model invariants, objects, tokens, and instances of rule execution. Each of these entities is associated with a set of variables, which in turn are in the scope of constraints. "Opening" a starting entity will reveal its variables, each of which will reveal its constraints when opened.

The Navigator View is the union of the Token Network and Constraint Network views as well as information not

explicit in any other view. Beginning from an entity present in some other view and every immediate neighbor entity, the Navigator view allows incremental exploration of every entity connection present in the plan.

The amount of information in a plan quickly exceeds that which can be easily treated by these views, so PlanWorks offers a *Content Filter* to restrict the visual elements to those related to particular predicates, the predicates of particular objects, or predicates within a specified window of time.

Transactions

EUROPA$_2$ has a rich transaction set describing the various transformations within the plan database, constraint network, and rules engine that it uses for internal notification, but which also has value in debugging. PlanWorks offers a mechanism for querying the transactions on individual entites, of a particular type, that represent the state transformation from one step to the next, or a combination of these.

Planner Control

Planning can be quite expensive in terms of time and logging data after every planner decision only slows the process down, which can be counterproductive when one is attempting to determine the existence of a bug, trace its cause, or verify a fix. In order to alleviate this, PlanWorks has the ability to execute the planner on-line, breakpoint, and write only specified steps.

This planner control mechanism is achieved through the EUROPA$_2$ notion of a model as a compiled shared library. From within PlanWorks, the model, planner, and initial state are initialized and the user is presented with a control panel offering the ability to execute the next step and write, execute and write the next n steps, execute the next n steps without writing, execute to the end and write the final plan, or terminate the current run. Execution causes dynamic updates of the Sequence Steps View, ensuring that the user has an up-to-date view of what the planner is doing.

Beyond this, because models in EUROPA$_2$ are shared objects and initial states are files loaded at planner execution time, both can be swapped for different models or states without re-starting PlanWorks.

Aver

Introduction

"Aver" is a language for specifying run-time tests to verify proper behavior of a planning system, from the plan database to the model to the planner. It allows the description of partial or complete plans and events that occur during planning that constitute correct behavior. Files containing tests in Aver are converted to XML, which is then compiled to an internal byte-code and executed at planner run-time.

Aver is used to define tests over a *sequence of steps*, each corresponding to a partial plan logged by a planner during search. This assumption is very generic, as the planner can use any form of search from backtracking to local search. Furthermore, the planner can log plans periodically, e.g. every 5^{th} decision the planner makes.

```
Test('BasicAssertionExample',
//should be true at the beginning
  At first step : 1 == 1;
//should be true at the end as well
  At last step : 1 == 1;
//doomed to fail after the third step
  At each step > 3 : 0 != 0;
//only needs to be true once
  At any step in [0 3] :
    Count({1 2 3 4}) == 4;
//must be true at steps 3, 5, 7, and 9
  At step in {3 5 7 9} : 1 == 1;
);
```

Figure 1: Basic assertions in Aver. The first two assertions show the use of "first" and "last" in specifying steps. The third assertion specifies a subset of steps. The last two assertions show the differences between the "each" and "any" semantics.

Tests and Assertions

The largest unit of Aver is the *test*. Tests are named to allow for selective execution and contain sets of tests or *assertions*. An assertion consists of a specification of the set of steps at which the assertion must hold followed by a boolean assertion about the plan state.

A *step specification* consists of a specification of a subset of the sequence of steps, with an additional predicate of "any" or "each". An assertion with the "each" predicate must be true at all steps matching the step specification for the assertion to be considered true, assertions with the "any" predicate must be true for at least one step matching the specification. "Each" semantics is assumed if the predicate is omitted. Aver also has two special step identifiers, "first" and "last", to refer to those steps logically rather than numerically.

The boolean part of an assertion is a combination of queries for plan entites, built-in function calls, value specifications, and comparisons. All values in Aver are represented as *domains*; sets of values represented as either enumerations (i.e. "{1 2 3 4}") or intervals (i.e. "[1 4]" or "[2.5 2.9]"). Domains that contain only one value or whose upper and lower bounds are equal are called *singleton* domains. This is done because, most often, values specified in Aver are compared with the values of EUROPA$_2$ variables, which are themselves domains. Figure 1 offers some trivial example assertions.

Queries and Functions

Aver provides direct queries for three types of EUROPA$_2$ plan entities: objects, tokens, and transactions. These queries allow for the definition of subsets of entities in the partial plans matching the step specification through the specification of relevant properties of the entity type. The "Objects" query can be restricted by the object name or the values of object variables. The "Tokens" query can be restricted by the predicate name or the values of the temporal

```
Test('AnotherExample',
//there should be tokens in the plan
  At last step : Count(Tokens()) > 0;
//no backtracking in this plan
  At each step :
    Count(Transactions(type ==
      'RETRACTION')) == 0;
//a rover can't exceed the speed of
//light after the 10th step
At step > 10: Property('m_maxSpeed',
                    Objects(name ==
                        'SpiritRover'))
  < 300000000;
//there is only one location the rover
//can be at initially
  At first step :
    Count(Property('m_location',
      Tokens(predicate == 'Rover.at'
      object == 'SpiritRover'
      start == 0)))
  == 1;
);
```

Figure 2: Some more complex assertions. The first assertion uses the "Count" function and a query on the set of Transactions to ensure that no backtracking occurred during planning. The second assertion uses the "Property" function and a query on the set of Objects to ensure that a property holds. The last assertion demonstrates a query on the set of Tokens to check a property of the initial state.

or parameter variables. The "Transactions" query can be restricted by the exact name of the transaction, the type of the transaction, or the object transacted upon.

Aver has three built-in functions: "Count", "Entity", and "Property". "Count" returns the number of entities in its domain argument. "Entity" returns the nth entity in its domain argument, and "Property" returns the domain of the named variable of its single entity argument. The semantics of "Entity" are defined only for finite ordered domains, and the semantics of "Property" are only defined for single entities. Figure 2 has some more complex examples of assertions using queries and functions.

All boolean operators in Aver are defined at the level of domains, so Aver supports the usual equality, less than, greater than, less than or equal, and greater than or equal comparison operators as well as set subset-of, intersection, and exclusion operators.

A rough analogy can be drawn between Aver assertions and the `assert()` facility available in many programming languages. The common `assert()` marks a condition that must be true at a location determined by its position in code, and an Aver assertion marks a condition that must be true at a location determined by its step specification. Also, both indicate upon failure a problem that needs to be examined with a second tool; with `assert()`, this is a debugger, with Aver, PlanWorks.

Application

To demonstrate these tools, we present a model of the "pipesworld" domain, described in detail in (Milidiú, dos Santos Liproace, & de Lucena 2003), developed for EUROPA$_2$ in the modeling language developed for it, NDDL (New Domain Description Language).

Pipesworld is a domain describing the behavior of the systems used to store and transport petroleum derivative products. The peculiar constraints in this domain are:

1. The pipes must be pressurized (full) at all times.
2. The tanks have per-product capacities.
3. Because of (1), and the fluid nature of the products, it is economical to have only specific combinations of products present in a pipe simultaneously.

Products can be shifted onto a pipe from either end, forcing the product present in the pipe at the opposite end into the tank at that end.

The petroleum products are transported in units called "batches." We chose to represent a batch as a timeline with predicates representing its status in a pipe or tank or being shifted from a tank to a pipe, or vice-versa with parameters for the tank or pipe.

We chose to model only so-called "unitary" pipes—pipes that contain only one batch at a time—in the interest of simplicity. The model is, however, still interesting because there is an intermediate time between when the old batch is in the tank and the new batch occupies the pipe in which both batches are partially present in the pipe. We represent pipes as an extension of timelines, which offer automatic mutual exclusion, that are parameterized on the two tanks they connect.

Finally, tanks are represented as objects containing collections of EUROPA$_2$ resources, one for each type of product, each of which is parameterized with the number of batches of the particular product that tank can hold.

Moving a batch from a pipe to a tank creates a consumption transaction on the tank's appropriate batch-capacity resource at it's end time and moving a batch from a tank to a pipe creates a production transaction on the tank's batch-capacity resource at it's end time. The semantics of a resource in EUROPA$_2$ ensure that capacity is never exceeded.

In our initial state, there are three identical tanks; A_1, A_2, and A_3. There are two pipes, one connecting A_1 and A_2, called S_{12}, and one connecting A_1 and A_3, called S_{13}. There are also 14 batches of various products, two of which start in the pipes an the rest are in tanks.

The details of the initial and goal states are fairly uninteresting, but for the purposes of this discussion, we point out that batch 12 begins in A_3 and should end in A_2. Having constructed the model and the initial and goal states, we constructed the test in Aver before knowing what the final plan looks like.

The most trivial aspects of the Aver test confirm that the initial and goal states of the test are present in the final plan. To compose the rest of the test, we had to consider the model in conjunction with the initial and goal states. While it isn't currently possible to test the direct application of model

```
At last step :
  Count(Tokens(predicate='Batch.inPipe'
               object = 'B12'
               variable(name = 'm_pipe'
                        value= 'S13')
               start >
         Property('end',
         Tokens(predicate='Batch.inTank'
               start = 0 object = 'B12'
               variable(name = 'm_tank'
                        value = 'A3')))))
  > 0;
```

Figure 3: A rule-checking assertion.

rules, it is possible to make assertions about their necessary effects, and it is this type of assertion that composes the majority of the test suite. For example, the assertion in Figure 3 checks the property that batch 12 must be in pipe S_{13} sometime after it's in tank A_3, which it must necessarily be to end in tank A_2.

We mention this assertion in particular because it was the first to fail. Inspection in PlanWorks confirms this. A look at the Timeline View shows that batch B12 is shifted from tank A_3 to pipe S_{12}, a clear violation of the intended semantics of the model. Images from the Constraint Network View are shown in Figure 4 to make the parameter values more visible. This indicates a missing constraint.

Looking at the model text in Figure 5, we see that there is, indeed, a missing constraint.

This constraint can be achieved using NDDL's existential quantification, which selects objects based on filtering criteria. If we add the code in Figure 6 to the rule, where the comment about the missing constraint occurs, the test should pass. And, indeed, we find that it does. This is further confirmed by PlanWorks as seen in Figure 7.

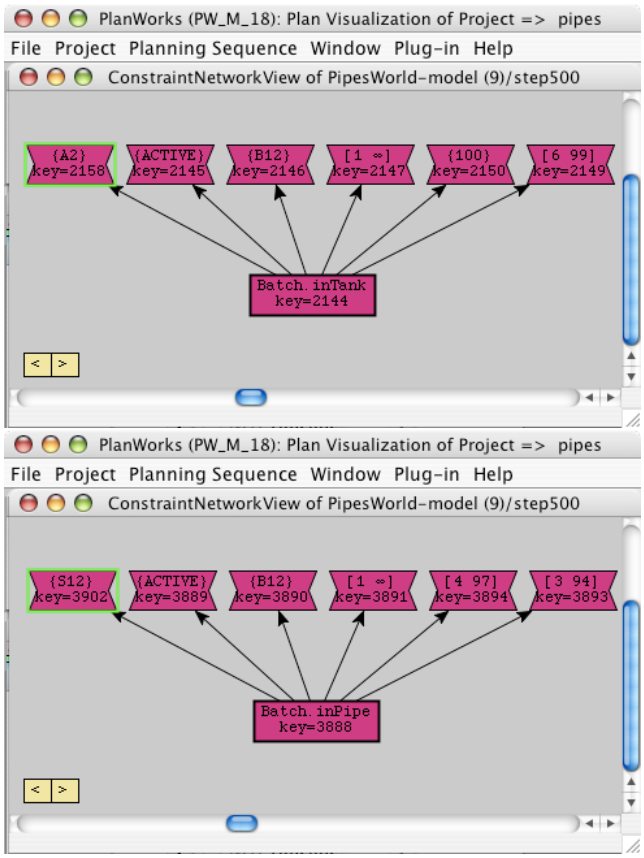


Figure 4: Top: The inTank token. Bottom: The erroneous inPipe token.

Future Work

PlanWorks

PlanWorks was originally conceived of as an integrated development environment for building and managing projects with EUROPA$_2$ and it is our intention to continue to develop features to aid in those tasks. In the near future, PlanWorks will be extended to handle model visualization and visual model building, and visualizing simple temporal networks. We also will use PlanWorks' plugin system to create planner-specific views of decision structures and heuristics.

We believe that features like automated examination of the constraint network and its execution trace to determine nogoods and the ability to alter the plan state during planner execution through the planner control mechanism will greatly add value.

Aver

As Aver becomes a more integral part of EUROPA$_2$'s test suite, we will add features to extend it's power. In particular, extending the step specification to deal with properties of

```
Batch::inTank {
  meets(object.shiftingToPipe stp);
  //should be a constraint here
  //requiring that the pipe
  //have the current tank as an endpoint
  starts(Resource.change tx)
  eq(tx.quantity, 1);
  if(object.m_product == lco) {
    eq(tx.object, m_tank.m_lco);
  }
  if(object.m_product == gasoline) {
    eq(tx.object, m_tank.m_gasoline);
  }
  //...
}
```

Figure 5: An erroneous rule.

```
bool b;
if(b == true) {
  PipeSegment p1 : {
    eq(p1.m_to, m_tank);
  }
  eq(stp.m_pipe, p1);
}
if(b == false) {
  PipeSegment p2 : {
    eq(p2.m_from, m_tank);
  }
  eq(stp.m_pipe, p2)
}
```

Figure 6: Existential quantification to fix the model.

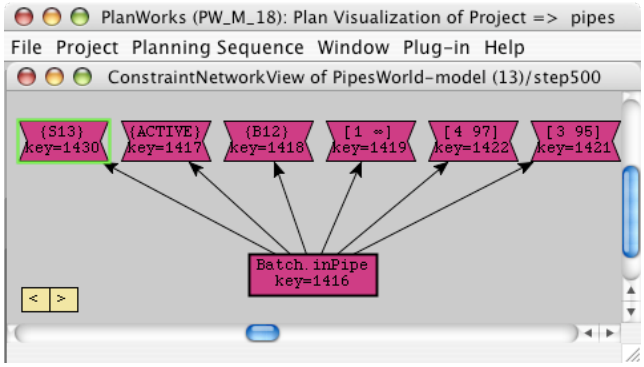


Figure 7: The inPipe token correctly constrained.

the step beyond just its number would reduce the fragility of Aver tests as well as allowing for implicative assertions, which are much more useful when verifying models.

We will extend the query capabilities to include structural assertions (entities with properties X are connected to things with properties Y), add configurable transaction sets to allow querying for custom transactions, and allow queries based on the model types of entities.

The assertion mechanism will be improved to allow for arithmetic expressions and disjunctive assertions, as well as optional assertions.

Acknowledgments

The authors would like to acknowledge Andrew Bachmann's contributions to the NDDL language used to describe EUROPA$_2$ planning domains, Will Taylor for his work on PlanWorks, Sailesh Ramakrishnan for his contributions as PlanWorks' prototype user, Bob Kanefsky for the Potato prototype that ultimately evolved into Planworks, and Mitchell Chang for his work with Conor McGann on the Tiny Test Language, which heavily informed the design of Aver.

References

Frank, J., and Jónsson, A. K. 2003. Constraint based attribute and interval planning. *Journal of Constraints*.

Milidiú, R. L.; dos Santos Liproace, F.; and de Lucena, C. J. P. 2003. Pipesworld: Planning pipeline transportation of petroleum derivatives. In *ICAPS Workshop on the Competition*.

Optimal Scheduling using Priced Timed Automata

Gerd Behrmann and **Kim G. Larsen** and **Jacob I. Rasmussen***
BRICS, Aalborg University, Denmark

Abstract

This contribution reports on the considerable effort made recently towards extending and applying well-established timed automata technology to optimal scheduling and planning problems. The effort of the authors in this direction has to a large extent been carried out as part of the European projects VHS (VHS 2005) and AMETIST (AMETIST 2005) and are available in the recently released UPPAAL CORA (UPPAAL CORA 2005), a variant of the real-time verification tool UPPAAL (Larsen, Pettersson, & Yi 1997; Behrmann, David, & Larsen 2004) specialized for cost-optimal reachability for the extended model of so-called priced timed automata.

Introduction and Motivation

Since its introduction by Alur and Dill (Alur & Dill 1994) the model of timed automata has established itself as a standard modeling formalism for describing real-time system behaviour. A number of mature model checking tools (e.g. KRONOS, UPPAAL, IF (Bozga *et al.* 1998; Larsen, Pettersson, & Yi 1997; IF 2005)) are by now available and have been applied to the quantitative analysis of numerous industrial case-studies (UPPAAL 2005).

An interesting application of real-time model checking that has recently been receiving substantial attention is to extend and retarget the timed automata technology towards optimal scheduling and planning. The extensions include most importantly an augmentation of the basic timed automata formalism allowing for the specification of the acculumation of cost during behavior (Behrmann *et al.* 2001a; Alur, La Torre, & Pappas 2001). The state-exploring algorithms have been modified to allow for "guiding" the (symbolic) state-space exploration in order that "promising" and "cheap" states are visited first, and to apply branch-and-bound techniques (Behrmann *et al.* 2001b) to prune parts of the search tree that are guaranteed not to improve on solutions found so far. Also new symbolic data structures allowing for efficient symbolic state-space representation with additional cost-information have been introduced and implemented in order to efficiently obtain optimal or near-optimal solutions (Larsen *et al.* 2001). Within the VHS and AMETIST projects successful applications of this technology have been made to a number of benchmark examples and industrial case studies. With this new direction, we are entering the area of Operations Research with a well-established and extensive list of existing techniques (MILP, constraint programming, genetic programming, etc.). However, what we put forward is a completely new and promising technology based on the efficient algorithms/data structures coming from timed automta analysis, and allowing for very natural and compositional descriptions of highly non-standard scheduling problems with timing constraints.

Abstractly, a scheduling or planning problem may be understood in terms of a number of *objects* (e.g. a number of different cars, persons) each associated with various distinguishing attributes (e.g. speed, position). The possible plans solving the problem are described by a number of *actions*, the execution of which may depend on and affect the values of (some of) the objects attributes. Solutions, or feasible schedules, come in (at least) two flavors:

Finite Schedule: a finite sequence of actions that takes the system from the initial configuration to one of a designated collection of desired final configurations.

Infinite Schedule: an infinite sequence of actions that – when starting in the initial configuration – ensures that the system configuration stays indefinitely within a designated collection of desired configurations.

In order to reinforce quantitative aspects, actions may additionally be equipped with constraints on durations and have associated costs. In this way one may distinguish different feasible schedules according to their accumulated cost or time (for finite schedules) or their cost per time ratio in the limit (for infinite schedules) in identifying *optimal* schedules. It is understood that independent actions, in terms of the set objects the actions depend upon and may affect, may overlap time-wise.

One concrete scheduling problem is that of optimal *task graph scheduling* (TGS) consisting in scheduling a number of interdependent tasks (e.g. performing some arithmetic operations) onto a number of heterogenous processors. The interdependencies state that a task cannot start executing before all its predecessors have terminated. Furthermore, each task can only execute on a subset of the processors. An example task graph with three tasks is depicted in Figure 1. The task t_3 cannot start executing until both tasks t_1 and t_2

*Work partially done within the European IST project AMETIST.

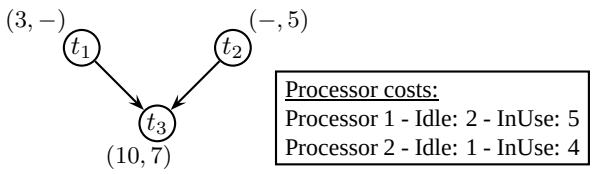


Figure 1: Task graph scheduling problem with 3 tasks and 2 processors.

have terminated. The available resources are two processors p_1 and p_2. The tasks (nodes) are annotated with the required execution times on the processors, that is, t_1 can only execute on p_1, t_2 only on p_2 while t_3 can execute on both p_1 and p_2. Furthermore, the idling costs per time unit of the processors are 2 and 1, respectively, and operations costs per time unit are 5 and 4, respectively.

Now, scheduling problems are naturally modeled using networks of timed automata. Each object is modelled as a separate timed automaton annotated with local, discrete variables representing the attributes associated with the object. Interaction often involves only a few objects and can be modeled as synchronizing edges in the timed automata models of the involved objects. Actions involving time durations are naturally modeled using guarded edges over clock variables. Furthermore, operation costs can be associated with states and edges in the model of priced timed automata (PTA) which was, independently, introduced in (Behrmann *et al.* 2001a) and (Alur, La Torre, & Pappas 2001). The separation of independent objects into individual processes and representing interaction between objects as synchronizing actions allows timed automata to make explicit the control flow of scheduling problems. In turn, this makes the models intuitively understood and easy to communicate. Figure 2 depicts PTA models for the task graph in Figure 1 and is explained in detail in Section .

The outline of the remainder of the paper is as follows: in Sections 2 and 3 we introduce the model of PTA, the problem of cost-optimal reachability and sketches the symbolic branch-and-bound algorithm used by UPPAAL CORA for solving this problem. Then in Section 4 we show how to model a range of generic scheduling problems using PTA, provide experimental evaluation and describe two industrial scheduling case-studies. Finally, in Section 5, we comment on other PTA-related optimization problems to be supported in future releases of UPPAAL CORA.

Priced Timed Automata

In this section we give a more precise and formal definition of priced timed automata (PTA) and their semantics[1]. Let X be a set of clocks. Intuitively, a clock is a non-negative real valued variable that can be reset to zero and increments at a fixed rate with the passage of time. A priced timed automaton over X is an annotated directed graph with a vertex set L, an edge set E and a distinguished vertex $l_0 \in L$ called the initial location. In the tradition of timed automata, we call vertices *locations*. Edges are labelled with guard expressions and a reset set. A guard is a conjunction of simple constraints $x \bowtie k$, where x is a clock in X, k is a non-negative integer value, and $\bowtie \in \{<, \leq, =, \geq, >\}$. We say that an edge is enabled if the guard evaluates to true and the source location is active. A reset set is a subset of X. The intuition is that the clocks in the reset set are set to zero whenever the edge is taken. Finally, locations are labelled with invariants. An invariant is a conjunction of simple conditions $x \prec k$, where x is a clock in X, k is a non-negative integer-value, and $\prec \in \{<, \leq\}$. Intuitively, an invariant must evaluate to true whenever its location is active. The previous definition is in fact that of a timed automaton. To form a priced timed automaton, we annotate the edges and the locations with costs and cost rates, respectively. Formally, this is done by introducing a function $P : L \cup E \to \mathbb{N}_0$.

The semantics of a PTA is easily defined as a *priced transition system*. Transitions of a priced transition system are labelled with a non-negative real-valued cost p. We skip the formal definition of a priced transition system and proceed with the semantics of PTA. A state of a PTA contains the active location $l \in L$ and a valuation of all clocks $v : X \to \mathbb{R}_{\geq 0}$ such that the invariant of l evaluates to true for v. There are two types of transitions: *edge transitions* and *delay transitions*. Edge transitions are the result of following an enabled edge in the PTA. As a result, the destination location is activated and the clocks in the reset set are set to zero. The cost of the transition is given by the cost of the edge.
More formally, we have $(l, v) \to_p (l', v')$ if there is an edge e from l to l', such that the guard of e evaluates to true in the source state (l, v), v' is derived from v by resetting all clocks in the reset set of e, and $p = P(e)$ is the cost of the edge.

Delay transitions are the result of the passage of time and do not cause a change of location. A delay is only valid if the invariant of the active location is satisfied by all intermediate states. The cost of a delay transition is given by the product of the length of the delay and the cost rate of the active location.

More formally, we have $(l, v) \xrightarrow{\delta}_p (l, v')$ if $p = \delta \cdot P(l)$, v' is derived from v by incrementing all clocks by δ and the invariant of l is satisfied by (l, v), (l, v') and all intermediary states.

Finally, the initial state is $s_0 = (l_0, v_0)$, where l_0 is the initial location, and v_0 evaluates to zero for all clocks. For networks of timed automata we use vectors of locations and the cost rate of a vector is the sum of cost rates in locations.

[1] We ignore the syntactic extensions of discrete variables and parallel composition of automata and note that these can be added easily.

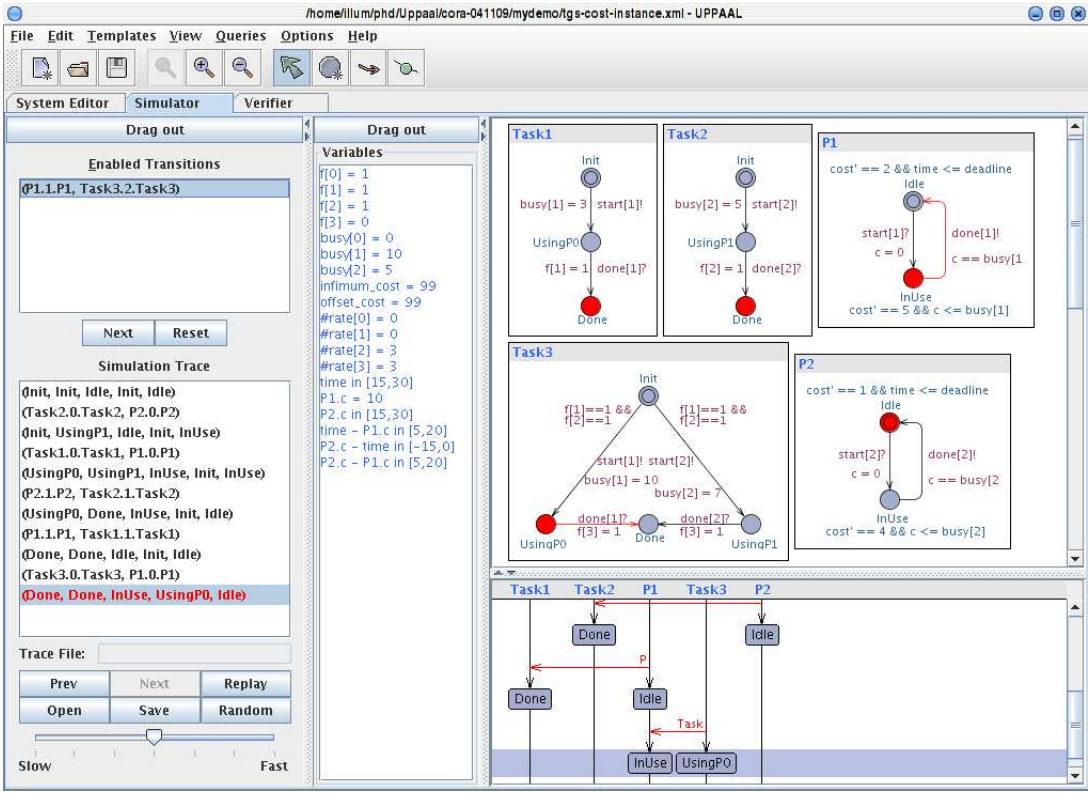


Figure 2: Screen shot of the UPPAAL CORA simulator for the task graph scheduling problem of Figure 1.

Optimal Scheduling

We now turn to the definition of the optimal reachability problem for PTA and provide a brief and intuitive overview of UPPAAL CORA's branch and bound algorithm for cost-optimal reachability analysis.

Cost-optimal reachability is the problem of finding the minimum cost of reaching a given goal location. More formally, an execution of a PTA is a path in the priced transition system defined by the PTA, i.e., $\alpha = s_0 \xrightarrow{a_1}_{p_1} s_1 \xrightarrow{a_2}_{p_2} s_2 \cdots \xrightarrow{a_n}_{p_n} s_n$. The cost, $cost(\alpha)$, of execution α is the sum of all the costs along the execution. The minimum cost, $mincost(s)$ of reaching a state s is the infimum of the costs of all finite executions from s_0 to s. Given a PTA with location l, the *cost-optimal reachability problem* is to find the largest cost k such that $k \leq mincost((l, v))$ for all clock valuations v.

Since clocks are defined over the non-negative reals, the priced transition system generated by a PTA can be uncountably infinite, thus an enumerative approach to the cost-optimal reachability problem is infeasible. Instead, we build upon the work done for timed automata by using *priced symbolic states*. Priced symbolic states provide symbolic representations of possibly infinite sets of actual states and their association with costs. The idea is that during exploration, the infimum cost along a symbolic path (a path of symbolic states) is stored in the symbolic states itself. If the same state is reached with different costs along differ-

COST := ∞
PASSED := $\emptyset$
WAITING := $\{\mathcal{S}_0\}$
while WAITING $\neq \emptyset$ **do**
  select $\mathcal{S} \in$ WAITING //based on branching strategy
  C $\leftarrow mincost(\mathcal{S})$
  if PASSED $\not\leq_{dom} \mathcal{S}$ **and** C $+ remain(\mathcal{S}) <$ COST **then**
    PASSED $\leftarrow$ PASSED $\cup \{\mathcal{S}\}$
    if $S \in$ GOAL **then**
      COST $\leftarrow$ C
    else
      WAITING $\leftarrow \{\mathcal{S}' \mid \mathcal{S}' \in$ WAITING or $\mathcal{S} \rightarrow \mathcal{S}'\}$
return COST

Figure 3: branch and bound algorithm.

ent paths, the symbolic states can be compared, discarding the more expensive state. Analogous to timed automata, the priced symbolic states we encounter for PTA are representable by simple constraint systems over clock differences (often refered to as a clock-zone in the timed automata literature). The cost is given by an affine plane over the clock-zone. For a formal description of priced symbolic states and priced zones we refer to (Larsen *et al.* 2001; Rasmussen, Larsen, & Subramani 2004).

In UPPAAL CORA, cost-optimal reachability analysis is performed using a standard branch and bound algorithm. Branching is based on various search strategies implemented in UPPAAL CORA which, currently, are breadth-first, ordinary, random, or best depth-first with or without random restart, best-first, and user supplied heuristics. The latter enables the user to annotate locations of the model with a special variable called *heur* and the search can be ordered according to either largest or smallest *heur* value. Bounding is based on a user-supplied, lower-bound estimate of the *remaining* cost to reach the goal from each location.

The algorithm depicted in Figure 3 is the cost-optimal reachability algorithm used by UPPAAL CORA. It maintains a PASSED-list of elements that have been explored and a WAITING-list of elements that need to be explored and is instantiated with the initial symbolic state $\mathcal{S}_0$. The variable COST holds the currently best known cost of reaching the goal location; initially it is infinite. The algorithm iterates until no more states need to be explored. Inside the while-loop we select and remove a state, $\mathcal{S}$, from WAITING based on the branching strategy. If $\mathcal{S}$ is dominated[2] by another state that has already been explored or it is not possible to reach the goal with a lower cost than COST, we skip this state. Otherwise, we add $\mathcal{S}$ to PASSED and if $\mathcal{S}$ is a goal location we update the best known cost to the best cost in $\mathcal{S}$. If not, we add all successors of $\mathcal{S}$ to WAITING and continue to the next iteration.

Modeling

As mentioned earlier, one of the main strengths of using priced timed automata for specifying and analyzing scheduling problems is the simplicity of the modeling aspect. In this section, we show how to model generic scheduling problems, provide experimental results, and describe two industrial case studies.

Scheduling problems often consist of a set of passive objects, called resources, and a set of active objects, called tasks. The resources are passive in the sense that they provide a service that tasks can utilize. Traditionally, the scheduling problem is to complete the tasks as fast as possible using the available resources under some constraints, e.g. limited availability of the resource, no two tasks can, simultaneously, use the same resource, etc. The models we provide in this section are all cost extensions of the classical scheduling problem.

[2] A state, $\mathcal{S}'$, dominates another state, $\mathcal{S}$, if $\mathcal{S}'$ contains at least the same actual states as $\mathcal{S}$, all of which have been reached with a lower cost.

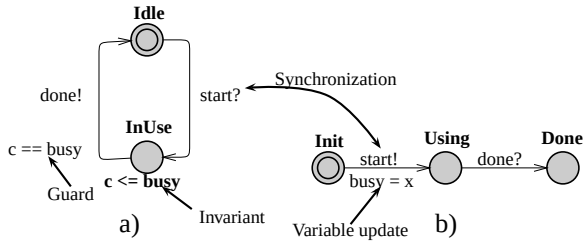


Figure 4: a) Resource template with clock **c**. b) Task template.

A generic resource model (see Figure 4a) is a two-location cyclic process with a single local clock, **c**. The two locations indicate whether the resource is **Idle** or **InUse**. The resource moves from **Idle** to **InUse**, when a task initiates a synchronization over the channel **start** and in the process, **c** is reset. The resource will maintain **InUse** until the clock reaches some usage time, **busy**, it then initiates synchronization over the channel **done**.

A generic task model (see Figure 4b) is an acyclic process progressing from an initial location, **Init**, to a final location **Done**, indicating that the task is complete. Intermediate locations describe acquiring resources and releasing them, i.e. the task will transit to state **Using** by initiating synchronization over a **start** channel and setting the **busy** variable of the resource. The task will remain here until the resource initiates synchronization using the **done** channel.

To solve the scheduling problem, we pose the reachability question of whether we can reach a state in which all tasks are in the location **Done**. In the following three sections we present some classical scheduling problems, all of which are slight modifications of the generic templates.

Job Shop Scheduling

Problem: We are given a number of machines (resources) and a number jobs (tasks) with corresponding recipes. A recipe for a job dictates the subset of machines that the job should be processed by, the order in which the processing should happen, and the duration of each processing step. Now, the scheduling problem is to assign to each job a starting time for every required machine such that no machine is occupied by two jobs at the same time.
Cost: The model can be extended with costs by assigning to each machine an idling cost and a operation cost.
Modeling: Figure 5a depicts a job and a machine. The model of the machine is identical to the resource template, except that both locations have been extended with cost rates. The job model is a "serial" composition of the task template, i.e. the job serially requests the machines described by the recipe, in this case machines 0, 1, and 2 for 7, 5, and 15 time units, respectively.

Task Graph Scheduling

Problem: This problem is described in Section .
Cost: We assign to each processor an energy consumption

rate while idle and while executing. Now, the overall objective is to find the schedule that minimizes the total cost while respecting a global (or task individual) deadline.

Modeling: The models for a task and a processor are depicted in Figure 2. Again, the processor model is an exact instance of the resource template with added cost rates. Tasks 1 and 2 are exact instances of the task template, while task 3 is not. The reason is that tasks 1 and 2 can only execute on one processor each, while task 3 can execute on both, thus, task 3 is an extension of the task template with a nondeterministic choice between the processors. Furthermore, the edges leaving the initial state have been extended with a guard specifying the dependencies of the task graph, i.e. task 3 requires tasks 1 and 2 to be finished, **f[1]** && **f[2]**.

Aircraft Landing

Problem: Given a number of aircrafts (tasks) with designated type and landing time window, assign a landing time and runway (resource) to each aircraft such that the aircraft lands within the designated time window while respecting a minimum wake turbulence separation delay between aircrafts of various types landing on the same runway.

Cost: The cost extended problem associates with each aircraft an additional target landing time corresponding to approaching the runway at cruise speed. Now, if an aircraft is assigned a landing time earlier than the target landing time, a cost per time unit is incurred, corresponding to powering up the engines. Similarly, if an aircraft is assigned a later landing time than the target landing time a cost per time unit is added corresponding to increased fuel consumption while circling above the airport.

Modeling: Figure 5b depicts a runway that can handle aircrafts of types B747 and A420, and an aircraft with target landing time 153, type A420 and time window [129,559]. Unlike the other models, the runway model has only a single location in its cycle indicating both that the resource is **IdleAndInUse**. A single location is used since the duration that a runway is occupied depends solely on the types of consecutively landing aircrafts. Thus, the runway maintains a clock per aircraft type holding the time since the latest landing of an aircraft of the given type and access to the runway is controlled by guards on the edges. The nondeterminism of the aircraft model does not distinguish between the runway to use, but whether to land early ([129,153]) or late ([153,559]). Choosing to land early, the aircraft model moves to the **OnTime** location and must remain here until the target landing time while incurring a cost rate per time unit for landing early, similarly, the aircraft can choose to land late and move to **Delayed** may remain there until the latest landing time while paying a cost rate for landing late.

PTA versus MILP

We only provide experimental results for the aircraft landing problem comparing the PTA approach to that of MILP. For performance results of the job shop and task graph scheduling problems, we refer to (Behrmann *et al.* 2001b; Rasmussen, Larsen, & Subramani 2004; Abdeddaim, Kerbaa, & Maler 2003).

RW		Planes	10	15	20	20	20	30	44
		Types	2	2	2	2	2	4	2
1	MILP (s)		*0.4*	*5.2*	*2.7*	*220.4*	*922.0*	*33.1*	*10.6*
	MC (s)		**0.8**	**5.6**	**2.8**	**20.9**	**49.9**	**0.6**	**2.2**
	Factor		*2.0*	*1.08*	*1.04*	**10.5**	**18.5**	**55.2**	**48.1**
2	MILP (s)		*0.6*	*1.8*	*3.8*	*1919.9*	*11510.4*	*1568.1*	*0.2*
	MC (s)		**2.7**	**9.6**	**3.9**	**138.5**	**187.1**	**6.0**	**0.9**
	Factor		*4.5*	*5.3*	*1.02*	**13.9**	**61.5**	**261.3**	*4.5*
3	MILP (s)		*0.1*	*0.1*	*0.2*	*2299.2*	*1655.3*	*0.2*	N/A
	MC (s)		**0.2**	**0.3**	**0.7**	**1765.6**	**1294.9**	**0.6**	
	Factor		*2.0*	*3.0*	*3.5*	**1.30**	**1.28**	*3.0*	
4	MILP (s)		N/A	N/A	N/A	*0.2*	*0.2*	N/A	N/A
	MC (s)					**3.3**	**0.7**		
	Factor					*16.5*	*3.5*		

Figure 6: Computational result for the aircraft landing problem using PTA and MILP on comparable machines.

Figure 6 displays experimental results for various instances of the aircraft landing problem using MILP and PTA. The results for MILP have been taken from (Beasley *et al.* 2000) and the results for PTA have been executed on a comparable computer. Factors in bold indicate the performance difference in favor of PTA and similarly for italics and MILP. The experiments clearly indicate that PTA is a competitive approach to solving scheduling problems and for one non-trivial instance it is even more than a factor 250 faster than the MILP approach. However, the required computation time of the PTA approach grows exponentially with the number of added runways (and thus clocks) while no similar statement can be made for the MILP approach. Thus, PTA is a promising method for solving scheduling problems, but further experiments need to be conducted before saying anything more conclusive.

Industrial Case Study: Steel Production

Problem: Proving schedulability of an industrial plant via reachability analysis of a timed automaton model was first applied to the SIDMAR steel plant, which was included as a case study of the Esprit-LTR Project 26270 VHS (Verification of Hybrid Systems). The plant consists of five processing machines placed along two tracks and a casting machine where the finished steels leaves the system. The tracks and machines are connected via two overhead cranes. Each quantity of raw iron enters the system in a ladle and depending on the desired final steel quality undergoes treatments in the different machines for different durations. The planning problem consists in controlling the movement of the ladles of steel between the different machines, taking the topology (e.g. conveyor belts and overhang cranes) into consideration.

Performance: A schedule for three ladles was produced in (Fehnker 1999) for a slightly simplified model using UPPAAL. In (Hune, Larsen, & Pettersson 2001) schedules for up to 60 ladles were produced also using UPPAAL. However, in order to do this, additional constraints were included that reduce the size of the state-space dramatically, but also prune possibly sensible behavior. A similar reduced model was used by Stobbe (Stobbe 2000) using constraint program-

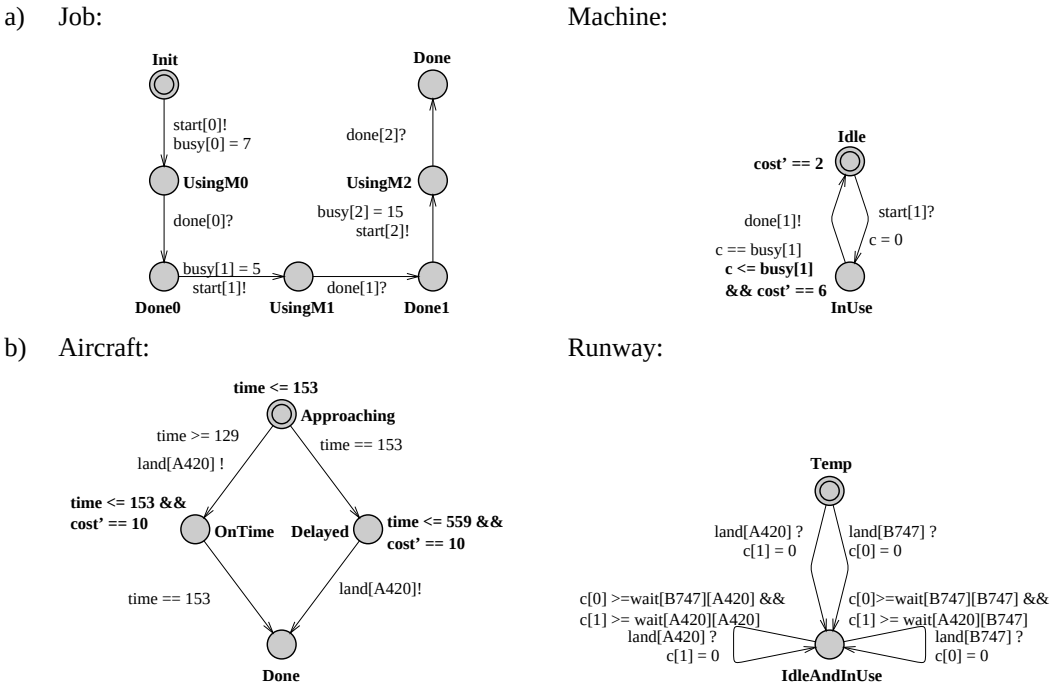


Figure 5: Priced timed automata models for two classical scheduling problems.

ming to schedule 30 ladles. All these works only consider ladles with the same quality of steel. In (Behrmann *et al.* 2001b), using a search order based on priorities, a schedule for ten ladles with varying qualities of steels is computed within 60 seconds cpu-time on a Pentium II 300MHz. The initial solution found is improved by 5% within the time limit. Allowing the search to go on for longer, models with more ladles can be handled.

Industrial Case Study: Lacquer Production

*Problem:*The problem was provided by an industrial partner of the European AMETIST project as a variation on job shop scheduling. The task is to schedule lacquer production. Lacquer is produced according to a recipe involving the use of various resources, possibly concurrently, see Figure 7. An *order* consists of a recipe, a quantity, an earliest starting date and a delivery date. The problem is then to assign resources to the order such that the constraints of the recipes and of the orders are met. Additional constraints are provided by the resources, as they might require cleaning when switching from one type of lacquer to another, or might require manual labor and thus are unavailable during the night or in weekends.

Cost: The cost model is similar to that of the aircraft landing problem. Orders finished on the delivery date do not incur any costs (except regular production costs which are not modeled as these are fixed). Orders finishing late are subject to *delay costs* and orders finishing too early are subject to *storage costs*. Cleaning resources might generate additional costs.

Modeling: Resources are modeled using the resource

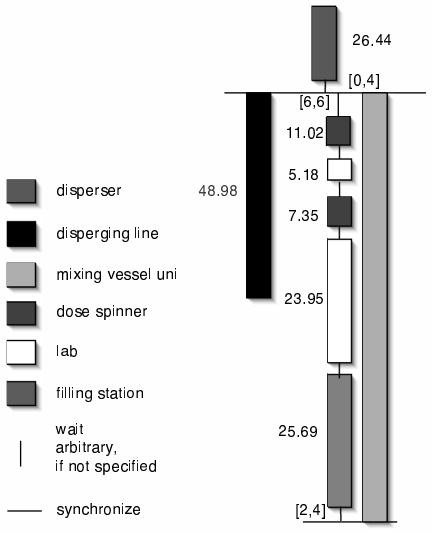


Figure 7: A lacquer recipe. Each bar represents the use of a resource. Horizontal lines indicate synchronization points. Timing constraints for how long resources are used or separation times between the use of resources can be provided either as a fixed time or time window.

template. Resources requiring cleaning are extended with additional information to keep track of the last type of lacquer produced on the resource. Cleaning costs are typically a fixed amount and are added to the cost when cleaning is performed. Orders are modeled similarly to tasks in the task graph scheduling problem, except that multiple resources may be acquired simultaneously. Storage and delay costs are modeled similarly to costs in the aircraft landing problem.

Other Optimization Problems

At present UPPAAL CORA supports cost-optimal location-reachability for PTAs. However, a number of other optimization problems are planned to be included in future releases.

For several planning problems the objective is to *repeat* a treatment or process *indefinitely* and to do so in a cost-optimal manner. Now let $\alpha = s_0 \stackrel{a_1}{\rightarrow}_{p_1} s_1 \stackrel{a_2}{\rightarrow}_{p_2} s_2 \cdots \stackrel{a_n}{\rightarrow}_{p_n} s_n \cdots$ be an infinite execution of a given PTA, let c_n (t_n) denote the accumulated cost (time) after n steps (i.e. $c_n = \sum_{i=1}^{n} p_i$). Then the limit of c_n/t_n when $n \rightarrow \infty$ describes the cost per time of α in the long run and is the cost of α. The optimization problem is to determine the (value of the) optimal such infinite execution α^*. In (Bouyer, Brinksma, & Larsen 2004) this problem has been shown decidable for PTA using an extension of the so-called region-technique. Though this technique nicely demonstrates decidability of the problem (and many other decision problems for timed automata) it does not provide a practical implementation, which is still to be identified. However a method for determining *approximate optimal* infinite schedules have been identified and applied to the synthesis of so-called Dynamic Voltages Scaling scheduling strategies.

Optimization problems may involve *multiple* cost variables (e.g. money, energy, pollution, etc.). Currently UPPAAL CORA is only capable of optimizing with respect to single costs. However, for scheduling problems with multiple costs, there might well be several optimal solutions due to "negative" dependencies between costs: minimizing one cost-variable (e.g. money) might maximize others (e.g. pollution). In (Larsen & Rasmussen 2005) an extension of the priced zone technology for PTA has been extended to multi-price TA allowing efficient synthesis of solutions optimal with respect to a chosen *primary* cost-variable but subject to user-specified upper bounds on the remaining *secondary* cost-variables.

Finally, scheduling problems may involve *uncertainties* due to certain actions being under the control of an adversary. In this case the (optimal) scheduling problem is a game-theoretic problem consisting of determining a winning and optimal strategy for how to respond to any action chosen by this adversary. In (Bouyer *et al.* 2004a) the problem of synthesizing optimal, winning strategies for priced timed games has been shown to be computable under certain non-zenoness assumptions. However, the problem is not solvable using zone-based technology, but needs general polyhedral support in order to represent the optimal strategies (see (Bouyer *et al.* 2004b) for a methodology using HYTECH).

References

Abdeddaim, Y.; Kerbaa, A.; and Maler, O. 2003. Task graph scheduling using timed automata. *Proc. of IPDPS'03*.

Alur, R., and Dill, D. 1994. A theory of timed automata. *Theoretical Computer Science* 126(2):183–235.

Alur, R.; La Torre, S.; and Pappas, G. 2001. Optimal paths in weighted timed automata. *Lecture Notes in Computer Science* 2034:pp. 49–62.

AMETIST. 2005. Advanced methods in timed systems `http://ametist.cs.utwente.nl`.

Beasley, J. E.; Krishnamoorthy, M.; Sharaiha, Y. M.; and Abramson, D. 2000. Scheduling aircraft landings - the static case. *Transportation Science* 34(2):pp. 180–197.

Behrmann, G.; Fehnker, A.; Hune, T.; Larsen, K.; Pettersson, P.; Romijn, J.; and Vaandrager, F. 2001a. Minimum-cost reachability for priced timed automata. *Lecture Notes in Computer Science* 2034:pp. 147+.

Behrmann, G.; Fehnker, A.; Hune, T.; Larsen, K.; Pettersson, P.; and Romijn, J. 2001b. Efficient guiding towards cost-optimality in Uppaal. In *Proc. of TACAS'01*, number 2031 in Lecture Notes in Computer Science, 174–188. Springer–Verlag.

Behrmann, G.; David, A.; and Larsen, K. 2004. A tutorial on Uppaal. In *Formal Methods for the Design of Real-Time Systems*, number 3185 in Lecture Notes in Computer Science, 200–236. Springer Verlag.

Bouyer, P.; Cassez, F.; Fleury, E.; and Larsen, K. 2004a. Optimal strategies in priced timed game automata. In *Proc. of FSTTCS'04*, volume 3328 of *Lecture Notes in Computer Science*, 148–160. Springer–Verlag.

Bouyer, P.; Cassez, F.; Fleury, E.; and Larsen, K. 2004b. Synthesis of optimal strategies using HyTech. In *Proc. GDV'04*, Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers. To appear.

Bouyer, P.; Brinksma, E.; and Larsen, K. 2004. Staying alive as cheaply as possible. In *Proc. of HSCC'04*, volume 2993 of *Lecture Notes in Computer Science*, 203–218. Springer–Verlag.

Bozga, M.; Daws, C.; Maler, O.; Olivero, A.; Tripakis, S.; and Yovine, S. 1998. Kronos: A model-checking tool for real-time systems. In *Proc. of CAV'98*, volume 1427, 546–550. Springer-Verlag.

Fehnker, A. 1999. Scheduling a steel plant with timed automata. In *Proc. of RTCSA '99.*, 280. IEEE Computer Society.

Hune, T.; Larsen, K.; and Pettersson, P. 2001. Guided synthesis of control programs using Uppaal. *Nordic J. of Computing* 8(1):43–64.

IF. 2005. `http://www-verimag.imag.fr/~async/IF`.

Larsen, K., and Rasmussen, J. 2005. Optimal conditional reachability for multi-priced timed automata. *To appear in proceedings of FOSSACS'05*.

Larsen, K.; Behrmann, G.; Brinksma, E.; Fehnker, A.; Hune, T.; Pettersson, P.; and Romijn, J. 2001. As cheap as possible: Efficient cost-optimal reachability for priced timed automata. *Lecture Notes in Computer Science* 2102:pp. 493+.

Larsen, K.; Pettersson, P.; and Yi, W. 1997. Uppaal in a nutshell. *Int. Journal on Software Tools for Technology Transfer* 1(1-2):134–152.

Rasmussen, J.; Larsen, K.; and Subramani, K. 2004. Resource-optimal scheduling using priced timed automata. In *Proc. of TACAS'04*, volume 2988 of *Lecture Notes in Computer Science*, pp. 220–235. Springer Verlag.

Stobbe, M. 2000. Results on scheduling the sidmar steel plant using constraint programming. Internal report.

UPPAAL CORA. 2005. `http://www.cs.aau.dk/~behrmann/cora`.

UPPAAL. 2005. `http://www.uppaal.com`.

VHS. 2005. Verification of hybrid systems `http://www-verimag.imag.fr/VHS/`.

Testing conformance of real-time applications: Case of Planetary Rover Controller *

Saddek Bensalem and **Marius Bozga** and **Moez Krichen** and **Stavros Tripakis**
VERIMAG
Centre Equation, 2, avenue de Vignate, 38610 Gières, France. www-verimag.imag.fr.

Abstract

We propose a methodology for testing conformance of an important class of real-time applications in an automatic way. The class includes all applications for which a specification is available and can be translated into a network of timed automata. The method relies on the automatic generation of an observer from the specification on one hand, and the instrumentation of the system to be tested on the other hand. The testing process consists in feeding the traces generated by the instrumented system to the observer, which is a testing device, used to check conformance of a trace with respect to the specification. We have validated the approach on the NASA K9 Rover case study.

Introduction

Computer-aided verification of programs has been studied for decades by the formal method research community. Different models and specification languages have been proposed to describe systems and express desired properties about them in a precise way. The expressivity and the applicability of such models to various domains has been studied. It has been realized quite early, however, that the approach suffers from two fundamental problems of intractability. First, undecidability, because of Turing-machine expressiveness of many infinite-state models. Second, intractability because of *state-explosion*, that is, prohibitively large state spaces to be explored. A large effort then concentrated in tackling these problems, resulting in a number of significant advances. Powerful theorem-proving techniques, (semi-)automatic abstractions, symbolic representations of state space, on-the-fly algorithms, compositional and assume-guarantee methods, etc. Despite these, intractability remains a major obstacle to the applicability of formal verification.

Another major obstacle is the fact that for a number of systems, a formal model simply does not exist and is too difficult or costly to build.

A complementary or alternative approach widely used in the industry today is testing. Testing is less ambitious than verification, in the sense that it only aims at finding bugs, and not at proving correctness. Indeed, most test methods are not complete (i.e., the system cannot be guaranteed to be correct even if it passes all tests). Nevertheless, confidence in the correctness of the system increases as the number of successful tests increases (Zhu, Hall, & May 1997). This feature of testing is particularly appealing to the industry, because it allows engineers to decide how much effort to put in validation, in contrast to an "all-or-nothing" verification approach.

Moreover, testing does not require a model of the system under test (SUT). Most testing methods are "black box", in the sense that the only knowledge about the SUT is its interface to the outside world (set of inputs and outputs). A model of the specification is necessary for automated test generation, however, this model is usually finite state and much smaller than a model of the SUT. This, and the feature above, makes testing tractable.

In this paper, we propose a new methodology of dynamic testing for real-time applications. It is dynamic in the sense that it makes use of instrumentation of the SUT and of run-time verification technology. The class of systems we are targetting includes all systems where a specification exists and can be translated into (or given directly as) a network of timed automata (TA) (Alur & Dill 1994). Many instances of such systems can be found in the domain of robotics. There, a *plan* defines the steps to be performed to achieve a mission, and also gives detailed information about order, timing, etc., of these steps. The plan is fed as an input to an *execution platform* (the term includes software, middleware and hardware) which must implement it, by performing the specified steps in the specified timing and order. Thus, the plan can be taken to be the specification and the platform executing this plan to be the SUT.

Our methodology is illustrated in Figure 1 and is described in detail in the next section. Let us briefly summarize it here. The starting point is a plan, which is taken to be a high-level specification. This plan is automatically translated into a TA model. From the latter an *observer* is automatically synthesized. The observer is also a testing device, that is, it checks whether a sequence of observations (with time-stamps) conforms to the specification. The execution platform is instrumented, so that it can be interfaced with the observer. This interface must essentially export observable events and time-stamps for these events. The final step

*Work partially supported by European IST projects "Next TTA" under project No IST-2001-32111 and "RISE" under project No IST-2001-38117, and by CNRS STIC project "CORTOS".

is the testing itself. It can be done: (1) either on-the-fly, by running the plan on the instrumented execution platform and feeding the observations to the observer; (2) or off-line, by generating a set of "log-traces" from the execution platform, then feeding these traces to the observer one by one.

The main advantage of our method is that it is potentially fully-automatic. Plans can be automatically translated into networks of TA (A. Akhavan & Orfanidou 2004). Observers for TA can be generated automatically, as we show here. For the case study reported in this paper, we relied on the help of Klaus Havelund and Rich Washington at NASA, for the instrumentation of the execution platform and the generation of the traces. However, it should be possible to automate this part as well, in the general case, by identifying a mapping between platform events and specification events, and automatically scanning the code, adding event/time-stamp exporting commands to the identified platform events. Finally, the observation/testing process is automatic as well.

The rest of the paper is organized as follows. The first part presents the methodology in detail and a short review of the model of timed automata. The second part describes plans and their translation to networks of TA. The third part shows how observers can be generated automatically from TA. In the last part, we discusse the application of our method on the K9 Rover case study, and we conclusions and plans for future work.

Related work (Artho *et al.* 2003) report work very much related to ours. Their scheme is also based on the instrumentation of the SUT and the runtime analysis of the instrumented SUT using an observer. The starting point of their method is a test-input generator, which generates inputs to the instrumented SUT. These inputs are also fed to a property generator, which generates properties that the SUT must satisfy on these particular inputs. The properties and the execution traces are fed to an observer, which checks whether the former are satisfied by the latter. The test-input generator and the property generator are specifically written for the application to be tested, while the instrumentation package and the observer are generic tools used on different applications. In one of the two case studies reported in (Artho *et al.* 2003), namely, the K9 rover controller, the inputs are plans like the ones we use in this paper (see Section "Case Study"). The test-input generator generates all possible plans up to given number of nodes and bounds on timing constraints.

The differences between our work and the work of (Artho *et al.* 2003) are as follows:

- (Artho *et al.* 2003) include a test-input generator and an instrumentation package in their tool-chain. Our work is still incomplete in these aspects. For the K9 rover case study, we have relied on NASA personnel and tools for the input plans, instrumentation and generation of traces.
- In (Artho *et al.* 2003), a set of untimed temporal logic properties are automatically generated from each plan (recently, the work has been extended to real-time temporal-logic (Barringer *et al.* 2003c)). As stated in (Artho *et al.* 2003), "property generation is the difficult step in [the] process" and "[the] set of properties does not fully represent the semantics of the plan, but the approach appearred to be sufficient to catch a large amount of bugs".

  In our work, plans are translated into networks of timed automata. This is a fully-automatic and efficient process, which captures the full semantics of a plan (A. Akhavan & Orfanidou 2004). Notice that, once generated, the TA corresponding to a plan can be also used for other purposes than generating an observer. For instance, to check whether the plan meets certain properties, measure delays of various sub-stages, and so on.
- The observer tools used in (Artho *et al.* 2003; Barringer *et al.* 2003c) (DBRover (Drunsinsky 2000), JPax (Havelund & Roşu 2001; Bensalem & Havelund 2003), Eagle (Barringer *et al.* 2003b; 2003a)), are generic tools. In our work, we automatically generate an observer for each plan. This has the potential of optimizing the observer for the particular plan.

In conclusion, we believe that our work represents an alternative that is worth pursuing.

Methodology

Our methodology is mainly focused at testing robotic applications, such as the NASA K9 Rover (see Section). Such applications are often structured in two layers. A high-level *planning* layer and a low-level *execution* layer. The planning layer follows an input *plan*, which is a detailed description of the steps needed to accomplish the mission at hand. The planning layer issues commands to the execution layer, which tries to implement them and returns the results, including status information about success or failure. The planning layer then plans the next steps depending on this feedback and the instructions in the plan.

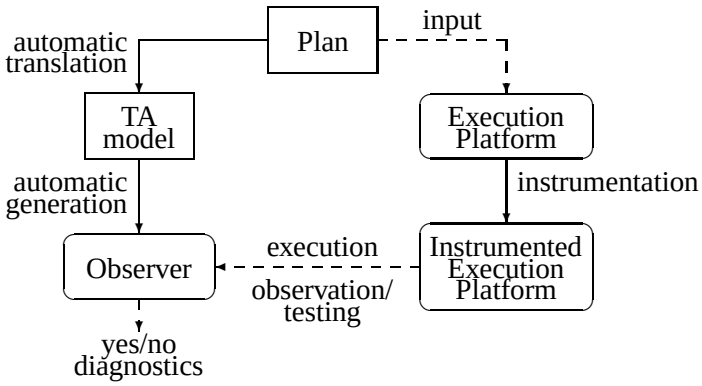


Figure 1: Methodology

A number of planning languages for robotic applications exist, see, for instance (Lyons 1993; Ingrand *et al.* 1996; Konolige *et al.* 1997; Simmons & Apfelbaum 1998; Peterson, Hager, & Hudak 1999; Muscettola *et al.* 2002; Goldman 2002). These languages allow to specify the ordering of the steps, their timing, how to handle exceptions or failures, and so on. Thus, they can be seen as the specification of the mission. A correct execution platform must then meet this specification. Our objective is to check this

by testing. More precisely, our methodology is illustrated in Figure 1. It consists of the following phases:

1. Automatic generation of a timed-automaton specification from the plan.
2. Automatic generation of an observer from the timed-automaton specification.
3. Instrumentation of the system under test, that is, the execution platform.
4. Execution and testing of the instrumented execution platform.

In the figure, solid arrows represent model and program transformations and dashed arrows represent data flow (output/input). We elaborate on each of the above phases in what follows.

The first step is to translate the plan in the form of a timed automaton, or a network of timed automata (TA). The translation must preserve the semantics of the plan, that is, the semantics of the TA and of the plan must be equivalent. It may also be the case that the TA *defines* the semantics of the plan in a formal way, as in (A. Akhavan & Orfanidou 2004) and this paper.

Having obtained the TA specification A, the next step consists in generating automatically an *observer* for A. The observer is a testing device. It observes the system under test (SUT) and checks whether the trace generated by the SUT conforms to A. The observed traces are sequences of observable events and associated time-stamps. The accuracy of the time-stamps depends on the accuracy of the clocks of the observer. In this paper, we consider two types of observers (we follow the terminology of (Henzinger, Manna, & Pnueli 1992)). *Analog* observers, which can observe real-time precisely, and *digital* (or *periodic-sampling*) observers, which measure time with a clock ticking at a given period. Digital-clock observers are clearly more realistic to implement, since in practice the observer will only have access to a finite-precision clock. However, analog-clock observers are still useful, for instance, when the implementation is discrete-time but its time step is not known a-priori.

An observed trace conforms to A if it can possibly be generated by A. Notice that A is typically modeled as a network of TA, which induces non-determinism and internal communication between the automata. These are artifacts of the model, irrelevant to the external behavior and to the specification itself. Thus, we "hide" them, by considering them as unobservable events. This means that the observer checks if the observed trace is a possible observation resulting from some trace of A.

The third step is the instrumentation of the execution platform. It aims at interfacing the latter with the testing device (the observer). Two possibilities exist here. Either testing is performed *on-the-fly* (or *on-line*), that is, during execution of the platform, which is connected to the observer at real-time. Or it is performed *off-line*, that is, by first executing the platform multiple times to obtain a set of *log-traces*, then feeding these traces to the observer. In both cases, the instrumented platform must be able to expose a set of observable events to the observer. In the case of testing off-line, the platform must also record the time-stamps of these events. For testing on-line, time-stamping can be done by the platform or by the observer. In the latter case, possible interfacing delays must be taken into account.

Instrumentation can be done manually or automatically. Depending on the complexity of the SUT, it can be a non-trivial task. Care should be taken so that the instrumentation does not itself alter the behavior of the system. For instance, the overhead of added code should be minimal, so as not to affect execution times of the tasks in the system. These are problems inherent in any instrumentation process, and are beyond the scope of this paper.

The final step is the testing procedure per-se. The traces generated by the instrumented platform are fed to the observer, either in real-time (for on-the-fly testing) or off-line. The observer checks conformance of each trace. If a trace is found non-conforming to the specification, the SUT is non-conforming. Otherwise, no conclusion can be made. However, confidence to the correctness of the SUT is increased with the number of tests. Obtaining a *representative* set of tests, so that some *coverage* criterion is met is an issue in any testing method (e.g., see (Zhu, Hall, & May 1997)), and is beyond the scope of the present paper.

Preliminaries

Timed sequences, projections and digitizations Let N be the set of non-negative integers. Let R be the set of non-negative rational numbers.

Consider a finite set of *actions* Σ. $\mathsf{RT}(\Sigma)$ (resp., $\mathsf{DT}(\Sigma)$) is defined to be the set of all finite-length *real-time sequences* (resp., *discrete-time sequences*) over Σ, that is, sequences of the form $(a_1, t_1) \cdots (a_n, t_n)$, where $n \geq 0$, for all $1 \leq i \leq n$, $a_i \in \Sigma$ and $t_i \in \mathsf{R}$ (resp., $t_i \in \mathsf{N}$), and for all $1 \leq i < j \leq n$, $t_i \leq t_j$. ϵ will denote the empty sequence. t_i will be called the *time-stamp* of a_i. Notice that time-stamps are relative to the beginning of a sequence. Thus, when concatenating sequences, they need to be adjusted. More precisely, given $\rho = (a_1, t_1) \cdots (a_n, t_n)$ and $\sigma = (b_1, t'_1) \cdots (b_m, t'_n)$, $\rho \cdot \sigma$ is the sequence $(a_1, t_1) \cdots (a_n, t_n)(b_1, t_n + t'_1) \cdots (b_m, t_n + t'_m)$. The time spent in a sequence ρ, denoted $\mathsf{time}(\rho)$, is the time-stamp of the last action (zero if the sequence is empty). For example, $\mathsf{time}((a, 0.1)(b, 1.2)) = 1.2$.

Given $\Sigma' \subseteq \Sigma$ and $\rho \in \mathsf{RT}(\Sigma)$ (resp., $\mathsf{DT}(\Sigma)$), the *projection* of ρ to Σ', denoted $P_{\Sigma'}(\rho)$, is a sequence in $\mathsf{RT}(\Sigma')$ (resp., $\mathsf{DT}(\Sigma')$), obtained by "erasing" from ρ all pairs (a_i, t_i) such that $a_i \notin \Sigma'$. For example, if $\Sigma = \{a, b\}$, $\Sigma' = \{a\}$ and $\rho = (a, 0)(b, 1)(a, 3)$, then $P_{\Sigma'}(\rho) = (a, 0)(a, 3)$. For a set of sequences $L \subseteq \mathsf{RT}(\Sigma)$ (or $L \subseteq \mathsf{DT}(\Sigma)$), $P_{\Sigma_{obs}}(L) = \{P_{\Sigma_{obs}}(\rho) \mid \rho \in L\}$.

Consider $\delta \in \mathsf{R}$, $\delta > 0$, and $\rho \in \mathsf{RT}(\Sigma)$. The *digitization* of ρ with respect to δ, denoted $[\rho]_\delta$, is a sequence in $\mathsf{DT}(\Sigma)$, obtained by replacing every pair (a_i, t_i) in ρ by $(a_i, \lfloor \frac{t_i}{\delta} \rfloor)$, where $\lfloor x \rfloor$ is the integral part of x. For example, if $\rho = (a, 0.1)(b, 0.9)(c, 1)(d, 2.3)$, then $[\rho]_1 = (a, 0)(b, 0)(c, 1)(d, 2)$ and $[\rho]_{0.5} = (a, 0)(b, 1)(c, 2)(d, 4)$. For a set of sequences $L \subseteq \mathsf{RT}(\Sigma)$, $[L]_\delta = \{[\rho]_\delta \mid \rho \in L\}$.

Timed automata We use timed automata (TA) (Alur & Dill 1994) with *deadlines* to model urgency (Sifakis &

Yovine 1996; Bornot, Sifakis, & Tripakis 1998). A *timed automaton over* Σ (TA) is a tuple $A = (Q, q_0, X, \Sigma, \mathsf{E})$ where Q is a finite set of *locations*; $q_0 \in Q$ is the initial location; X is a finite set of *clocks*; E is a finite set of *edges*. Each edge is a tuple (q, q', ψ, r, d, a), where $q, q' \in Q$ are the source and destination locations; ψ is the *guard*, a conjunction of constraints of the form $x \# c$, where $x \in X$, c is an integer constant and $\# \in \{<, \leq, =, \geq, >\}$; $r \subseteq X$ is the clock *reset*; $d \in \{\mathsf{lazy}, \mathsf{delayable}, \mathsf{eager}\}$ is the *deadline*; and $a \in \Sigma$ is the action. Intuitively, *eager* transitions must be executed as soon as they are enabled and waiting is not allowed; *lazy* transitions do not impose any restriction on time passing; finally, when a *delayable* transition is enabled, waiting is allowed as long as time progress does not disable it. We will not allow eager edges with guards of the form $x > c$.

A TA A defines an infinite labeled transition system (LTS). Its states are pairs $s = (q, v)$, where $q \in Q$ and $v : X \to \mathsf{R}$ is a clock *valuation*. $\vec{0}$ is the valuation assigning 0 to every clock of A. S_A is the set of all states and $s_0^A = (q_0, \vec{0})$ is the initial state. There are two types of transitions:

- discrete transitions of the form $(q, v) \xrightarrow{a}_A (q', v')$, where $a \in \Sigma$ and there is an edge (q, q', ψ, r, d, a), such that v satisfies ψ and v' is obtained by resetting to zero all clocks in r and leaving the others unchanged;
- timed transitions of the form $(q, v) \xrightarrow{t}_A (q, v + t)$, where $t \in \mathsf{R}, t > 0$ and there is no edge (q, q'', ψ, r, d, a), such that: either $d = \mathsf{delayable}$ and there exist $0 \leq t_1 < t_2 \leq t$ such that $v + t_1 \models \psi$ and $v + t_2 \not\models \psi$; or $d = \mathsf{eager}$ and $v \models \psi$.

We use notation such as $s \xrightarrow{a}_A$, $s \not\xrightarrow{a}_A$, ..., to denote that there exists s' such that $s \xrightarrow{a}_A s'$, there is no such s', and so on. This notation extends to sequences in $\mathsf{RT}(\Sigma)$: $s \xrightarrow{\epsilon}_A s$ and if $s \xrightarrow{\rho}_A s'$ and $s' \xrightarrow{t}_A \xrightarrow{a}_A s''$, then $s \xrightarrow{\rho \cdot (a,t)}_A s''$.

A state $s \in S_A$ is *reachable* if there exists $\rho \in \mathsf{RT}(\Sigma)$ such that $s_0^A \xrightarrow{\rho}_A s$. The set of reachable states of A is denoted $\mathsf{Reach}(A)$.

The set of *traces* of a TA A over Σ is defined to be

$$\mathsf{Traces}(A) = \{\rho \in \mathsf{RT}(\Sigma) \mid s_0^A \xrightarrow{\rho}_A\}. \quad (1)$$

Let $\Sigma_{obs} \subseteq \Sigma$ be a set of *observable* actions. The actions in $\Sigma \setminus \Sigma_{obs}$ are called *unobservable*. The set of *observed traces* of A with respect to Σ_{obs} is defined to be

$$\mathsf{ObsTraces}(A, \Sigma_{obs}) = P_{\Sigma_{obs}}(\mathsf{Traces}(A)). \quad (2)$$

Given $\delta \in \mathsf{R}$, $\delta > 0$, the set of *δ-digital observed traces* of a TA A is defined to be

$$\mathsf{DigTraces}(A, \Sigma_{obs}, \delta) = [\mathsf{ObsTraces}(A, \Sigma_{obs})]_\delta \quad (3)$$

Notice that $\mathsf{Traces}(A) \subseteq \mathsf{RT}(\Sigma)$, $\mathsf{ObsTraces}(A, \Sigma_{obs}) \subseteq \mathsf{RT}(\Sigma_{obs})$ and $\mathsf{DigTraces}(A, \Sigma_{obs}, \delta) \subseteq \mathsf{DT}(\Sigma_{obs})$.

Generating timed-automata from plans

In this section we describe how to obtain TA models from plans. We give the construction for the concrete language of plans performed by the K9 Rover executive, which is actually our case study (see section "Case Study"). Nevertheless, TA models are general enough to capture most of the constraints expressed in plan languages.

Plan →	*Node*
Node →	*Block* | *Task*
Block →	(**block**
	NodeAttr
	:node-list (*Node ... Node*))
Task →	(**task**
	NodeAttr
	:action *Symbol*)
NodeAttr →	**:id** *Symbol*
	:start-condition *Condition*
	:waitfor-condition *Condition*
	:maintain-condition *Condition*
	:end-condition *Condition*
	[**:continue-on-failure**]
Condition →	(**time** [+] *StartTime* [+] *EndTime*)

Figure 2: The concrete grammar of plans.

```
(block
 :id node0
 :continue-on-failure
 :start-condition ((1 5))
 :end-condition ((1 30))
 :node-list (
   (block
    :id node1
    :continue-on-failure
    :start-condition (1 5))
   (block
    :id node2
    :continue-on-failure
    :start-condition (+1 +5)
    :end-condition (+1 +30)
   )
 )
)
```

Figure 3: A plan example.

For the K9 Rover application, a plan is a hierarchical structure of actions that the executive must perform. Traditionally, plans are deterministic sequences of actions. However, increased autonomy requires added flexibility. The plan language therefore allows branching based on conditions that need to be checked, and also for flexibility with respect to the starting time of an action. We give here an example of a language used in the description of the plans that the executive must execute.

Plan Syntax A plan is a node, a node is either a task, corresponding to an action to be executed, or a block, corresponding to a logical group of nodes. Figure 2 shows the grammar for the language that we considered to describe plans. All node attributes except the node id are optional. Each node may specify a set of *conditions*. The *start condition* (that must be true at the beginning of the node execution), the *wait-for conditions* (wait while the condition is not true), the *maintain condition* (that must be true through

the execution of the node) and the *end condition* (that must be true at the end of the node execution). Each condition includes information about relative or absolute time window, indicating a lower and upper bound on the time. The *continue-on-failure* flag indicates what the behavior will be if a node failure is encountered.

We propose hereafter a compilation method allowing to obtain from a plan (that is, a syntactic object) a network of timed automata (that is, a semantic model) encoding all the accepted, reasonable executions of that plan.

For sake of simplificity, we consider the following abstract syntax for plans.

Definition 0.1 (plan syntax)
A plan P is a tuple $(N, \delta, \lambda, n_0)$ where

- *N is a finite set of nodes*
- *$\delta : N \rightarrow N^*$ is the node decomposition function, defined such that the image set relation $\hat{\delta} = \{(n, n') | n' \in \delta(n)\}$ satisfies*
  - *acyclicity: $\forall n \in N.\ n \notin \hat{\delta}^+(\{n\})$*
  - *disjointness: $\forall n_1, n_2 \in N, n_1 \neq n_2.\ \hat{\delta}^+(\{n_1\}) \cap \hat{\delta}^+(\{n_2\}) = \emptyset$*
- *$\lambda : N \rightarrow \Sigma \times \mathcal{I}^4 \times \mathcal{B}$ is the node labeling function, where Σ is a set of action labels, $\mathcal{I} = \{[l, u] \mid l, u \in \mathbb{N}\}$ is the set of interval constraints, and $\mathcal{B}$ are the booleans. That is, $\lambda(n) = (a_n, (s_n, w_n, m_n, e_n), f_n)$ where a_n is the action symbol, s_n, w_n, m_n, e_n are respectively the start, wait-for, maintain and end timed constraints, and f_n is the continue-on-failure flag associated to the node n.*
- *$n_0 \in N$ is the main (or start) node of the plan*

Plan Semantics Nodes are executed sequentially. For every node, execution proceeds through the following steps :

1. Wait until the start condition is satisfied; if the current time passes the end of the start condition, the node times out and this is a node failure.
2. The execution of a *task* proceeds by invoking the corresponding action. The action's status must be fail, if **:fail** is true or the time conditions are not met; otherwise, the status must be success. The execution of a block simply proceeds by executing each of the node in the node-list in order.
3. If the time exceeds the end condition, the node fails.

On a *node failure* occuring in a sequence, the value of the enclosing block node's *continue-on-failure* flag is checked. If true, execution proceeds to the next node in the sequence. If false, the node failure is propagated to the block enclosing the node and so on. If the node failure passes out to the top level block of the plan, the execution is aborted.

We present now the semantics of nodes and plans in terms of timed automata. The semantics is *constructive* in the sense that, automata can be effectively constructed, depending on syntactical description of the nodes. The semantics is also *compositional* in the sense that, the semantics of the plan is obtained directly by composing of timed automata associated to nodes.

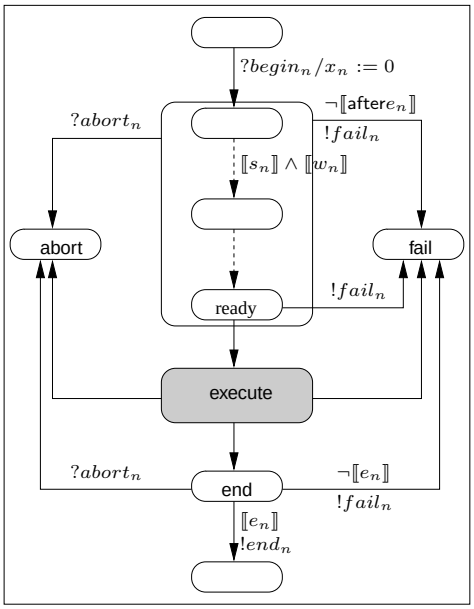


Figure 4: Timed automaton for the common part.

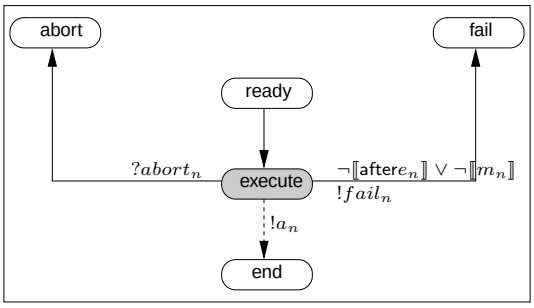


Figure 5: Timed automaton for the task specific part part.

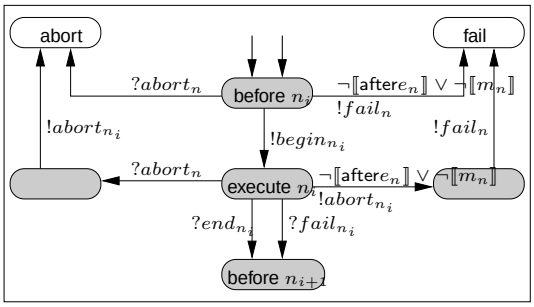


Figure 6: Timed automaton the pattern for the block specific part.

Let us first introduce some notations for some given plan $P = (N, \delta, \lambda, n_0)$. The set of actions Σ_P contains the set of synchronisation actions $begin_n$, $abort_n$, $fail_n$, end_n defined for all nodes n, and the set of elementary actions a_n, defined for task nodes n of the plan P.

The set of clocks $X_P = \{x_n \mid n \in N\}$ contains one clock x_n for each node n of the plan. This clock x_n is set to 0 when the execution of the node n begins. If $c_n = [l, u]$ is some constraint of the node n, we will note with $[\![c_n]\!]$ the timed guard $(l \leq x_n \wedge x_n \leq u)$. We note also with after$c$ the constraint $[-\infty, u]$ where the lower bound of c has been removed.

To each node n of P we associate a timed automaton over clocks X_P and actions Σ_P. The automaton encodes the sequential behaviour described by the node execution algorithm. Note that since the execution algorithm is deterministic, the timed automata obtained are deterministic.

Definition 0.2 (node semantics) *Figures 6, 5 and 4 illustrate the translation. Let n be a node with $\delta(n) = n_1 ... n_k$ and $\lambda(n) = (a_n, (s_n, w_n, m_n, e_n), f_n)$. The semantics of the node n is described by the timed automaton for the common part, shown in the left of the figure. The specific part is filled according to node attributes as follows. For a task node ($\delta(n) = \epsilon$), as shown in the top-right part of the figure. For a block node with continue on failure ($f_n = true$), as shown in the bottom-right part of the figure. For a block node without continue on failure ($f_n = false$), as shown in the bottom-right part of the figure, except that the transition labeled $?fail_{n_i}$ leads back to the right-most grey location. For the sake of simplicity, we represent eager transitions using solid lines and lazy transitions using dashed lines.* ! *and* ? *denote communication via CSP-like message passing.*

Finally, the semantics of the entire plan P is given by the parallel composition i.e, the network of timed automata defined for all of its nodes. Note that, the product automaton is deterministic too.

Definition 0.3 (plan semantics) *Let $P = (N, \delta, \lambda, n_0)$ be a plan, X_P the set of clocks and Σ_P the set of actions defined by P. Let A_{n_i} be the timed automata over X_P and Σ_P associated to nodes $n_i \in N$. The semantics of the plan P is given by the network $A_{n_0} || A_{n_1} || ... || A_{n_k}$.*

An example of a plan and the corresponding timed automata are given in Figure 7.

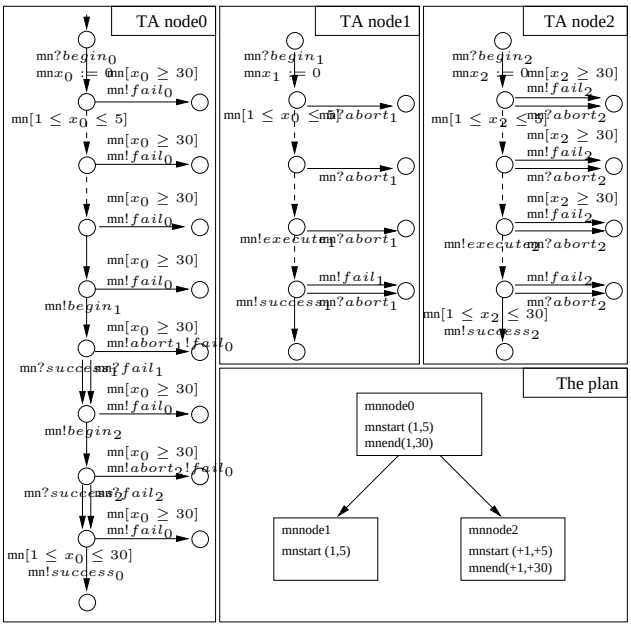


Figure 7: A plan and its translation to a network of three timed automata.

Generating observers from timed-automata

In this section, we define two kinds of observers for a TA specification A. They are distinguished by their observation capabilities with respect to time. The first, called analog observers (the terminology is taken from (Henzinger, Manna, & Pnueli 1992)) can observe a set of observable actions and the exact time-stamps of these actions. Thus, these observers can be thought of possessing a "perfect", real-time clock, which they can consult immediately upon observing an action. The second, called digital observers (or periodic-sampling observers) can also observe a set of observable actions, but they only have access to a digital clock, that is, a counter that ticks with a given period δ. Thus, when observing an action a which occurred at real-time t, the digital observer only knows the current value of its periodic clock, i.e., $\lfloor \frac{t}{\delta} \rfloor$.

The objective of the observers is to determine whether a given trace (generated by a system under observation) could be possibly generated by the specification A. If so, then the system under observation passes this test, otherwise, it fails.

Analog and digital observer definition Let us formalize the above notions. Let A be a TA over Σ and let $\Sigma_{obs} \subseteq \Sigma$ be a set of observable actions.

An *analog observer* for A with respect to Σ_{obs} is a total function

$$\mathsf{O} : \mathsf{RT}(\Sigma_{obs}) \rightarrow \{0, 1\}$$

such that

$$\forall \rho \in \mathsf{RT}(\Sigma_{obs}) \Big(\mathsf{O}(\rho) = 1 \Leftrightarrow \rho \in \mathsf{ObsTraces}(A, \Sigma_{obs}) \Big)$$

Thus, an analog observer performs nothing else but a *membership* test: does the observation ρ belong to the *language* of A, i.e., $\rho \in \mathsf{ObsTraces}(A, \Sigma_{obs})$. Notice that A has no acceptance conditions in our setting, thus, its language is prefix-closed. Also notice that observers are required to be deterministic, that is, to provide the same answer each time they are given the same observation. Thus, analog observers can be seen as deterministic machines accepting the language of A. It follows, from the fact that timed automata are non-determinizable in general (Alur & Dill 1994), that an analog observer cannot always be represented as a timed automaton. Moreover, checking whether this is the case for a particular automaton is undecidable (Tripakis 2004).

A *digital observer* (or periodic-sampling observer) for A with respect to Σ_{obs} and $\delta > 0$ is a total function

$$\mathsf{D} : \mathsf{DT}(\Sigma_{obs}) \to \{0, 1\}$$

such that $\forall \rho \in \mathsf{DT}(\Sigma_{obs})$.

$$\Big(\mathsf{D}(\rho) = 1 \Leftrightarrow \rho \in \mathsf{DigTraces}(A, \Sigma_{obs}, \delta)\Big)$$

Automatic observer generation using the state-estimation technique We next show how, given a timed automaton A, analog and digital observers can be automatically generated for A. The method relies on the *state-estimation* technique proposed in (Tripakis 2002), where it was applied to fault detection.

State estimation consists in computing, given an observation, the set of states of A which "match" this observation, that is, the set of all possible states which can be reached by some trace which yields the observed trace. If the state estimate remains non-empty all along the observation, then the latter can indeed be generated by A, since there exists at least one trace of A matching the observation. If, however, the state estimate becomes empty, then the observation cannot be generated by A.

State estimation is not more expensive than reachability analysis. In fact, in some cases it is cheaper.[1] As shown in (Tripakis 2002), (Krichen & Tripakis 2004) state estimates can be represented using standard data structures for TA, such as DBMs (Dill 1989), and can be computed using various versions of symbolic successor operators, depending on the desired estimator (analog or digital).

Generating analog observers Consider a timed automaton A over Σ and a set of observable actions $\Sigma_{obs} \subseteq \Sigma$. The *analog state-estimator* for A with respect to Σ_{obs} is the total function $\mathsf{SE_a} : \mathsf{RT}(\Sigma_{obs}) \to 2^{\mathsf{Reach}(A)}$, defined as follows:

$$\mathsf{SE_a}(\rho) = \{s \mid \exists \sigma \in \mathsf{RT}(\Sigma) \; . \; s_0^A \xrightarrow{\sigma}_A s \wedge P_{\Sigma_{obs}}(\sigma) = \rho\}$$

$\mathsf{SE_a}(\rho)$ contains all states where A can possibly be after executing a sequence which yields the analog observation ρ.

Now, define

$$\mathsf{O}(\rho) = \begin{cases} 1, & \text{if } \mathsf{SE_a}(\rho) \neq \emptyset \\ 0, & \text{otherwise} \end{cases}$$

It follows easily from the definitions that O defined as above is a valid analog observer for A w.r.t. Σ_{obs}.

We proceed to discuss how $\mathsf{SE_a}$ can be computed. Let S be a set of states of A. Let $a \in \Sigma$ and $t \in \mathsf{R}$. Define the following operators:

$$\mathsf{asucc}(S, a) = \{s' \mid \exists s \in S \; . \; s \xrightarrow{a}_A s'\}$$

$$\mathsf{tsucc}(S, t) = \{s' \mid$$

$$\exists s \in S \; . \; \exists \rho \in \mathsf{RT}(\Sigma \setminus \Sigma_{obs})\mathsf{time}(\rho) = t \wedge s \xrightarrow{\rho}_A s'\}$$

[1] The worst-case complexity of the membership problem in timed automata is studied in (Alur, Kurshan, & Viswanathan 1998). There, it is shown that for automata without epsilon-transitions (i.e., fully observable), the problem is NP-complete whereas for automata with epsilon-transitions the problem is PSPACE-complete (i.e., as hard as reachability).

$\mathsf{asucc}(S, a)$ contains all states which can be reached by some state in S after performing action a. $\mathsf{tsucc}(S, t)$ contains all states which can be reached by some state in S via a sequence ρ which contains no observable actions and takes exactly t time units.

The following proposition shows how $\mathsf{SE_a}(\rho)$ can be computed recursively on ρ.

Proposition 0.1

$$\begin{aligned} \mathsf{SE_a}(\epsilon) &= \mathsf{tsucc}(\{s_0^A\}, 0) \\ \mathsf{SE_a}(\rho \cdot (a, t)) &= \mathsf{asucc}(\mathsf{tsucc}(\mathsf{SE_a}(\rho), t - \mathsf{time}(\rho)), a) \end{aligned}$$

Generating digital observers Consider a timed automaton A over Σ and a set of observable actions $\Sigma_{obs} \subseteq \Sigma$. Let $\delta \in \mathsf{R}$, $\delta > 0$. The *digital state-estimator* for A with respect to Σ_{obs} and δ is the total function $\mathsf{SE_d} : \mathsf{DT}(\Sigma_{obs}) \to 2^{\mathsf{Reach}(A)}$, defined as follows:

$$\mathsf{SE_d}(\rho) = \{s \mid \exists \sigma \in \mathsf{RT}(\Sigma) \; . \; s_0^A \xrightarrow{\sigma}_A s \wedge [P_{\Sigma_{obs}}(\sigma)]_\delta = \rho\}$$

$\mathsf{SE_d}(\rho)$ contains all states where A can possibly be after executing a sequence which yields the digital observation ρ.

Now, define

$$\mathsf{D}(\rho) = \begin{cases} 1, & \text{if } \mathsf{SE_d}(\rho) \neq \emptyset \\ 0, & \text{otherwise} \end{cases}$$

It follows easily from the definitions that D defined as above is a valid digital observer for A w.r.t. Σ_{obs} and δ.

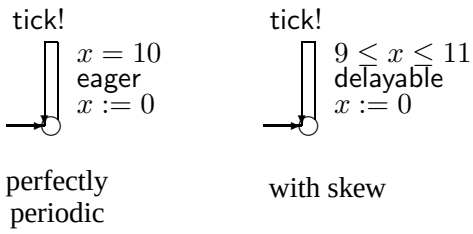


Figure 8: Two possible Tick automata.

We proceed to discuss how D can be computed. We first form the product of A with a Tick automaton like the one shown on the left of Figure 8. This automaton models the digital clock of the observer, assumed to be perfectly periodic with period $\delta = 10$. Other Tick automata can also be used, like the one on the right of the figure, to model phenomena such as clock skew or drift. (Notice that, in these cases, the definition of DigTraces must be modified.) Let the product automaton be $A_{\mathsf{tick}} = A \| \mathsf{Tick}$. We assume that tick is a new observable event, not in Σ. Let S be a set of states of A_{tick}. Define the following operator:

$$\mathsf{usucc}(S) = \{s' \mid \exists s \in S \; . \; \exists \rho \in \mathsf{RT}(\Sigma \setminus \Sigma_{obs}) \; . \; s \xrightarrow{\rho}_{A_{\mathsf{tick}}} s'\}$$

$\mathsf{usucc}(S)$ contains all states which can be reached by some state in S via a sequence ρ which contains no observable actions. Notice that, by construction of A_{tick}, the duration of ρ is bounded: since tick is observable and has to occur after at most 1 time unit, $\mathsf{time}(\rho) \leq 1$.

For $a \in \Sigma \cup \{\mathsf{tick}\}$, define:

$$\mathsf{dsucc}(S, a) = \mathsf{asucc}(\mathsf{usucc}(S), a)$$

$\mathsf{dsucc}^k(\cdot, a)$, for $k \in \mathsf{N}$, denotes the application of $\mathsf{dsucc}(\cdot, a)$ k times. That is, $\mathsf{dsucc}^0(\cdot, a)$ is the identity function and $\mathsf{dsucc}^{k+1}(\cdot, a) = \mathsf{dsucc}(\mathsf{dsucc}^k(\cdot, a), a)$.

The following proposition shows how $\mathsf{SE_d}(\rho)$ can be computed recursively on ρ.

Proposition 0.2

$$\begin{aligned} \mathsf{SE_d}(\epsilon) &= \{s_0^{A_{\mathsf{tick}}}\} \\ \mathsf{SE_d}(\rho \cdot (a, k)) &= \mathsf{dsucc}(\mathsf{dsucc}^k(\mathsf{SE_d}(\rho), \mathsf{tick}), a) \end{aligned}$$

Implementation We have implemented a prototype observer generation tool, called If2Obs, on top of the IF environment (Bozga *et al.* 2000). The IF modeling language allows to specify systems consisting of many processes communicating through message passing or shared variables and includes features such as hierarchy, priorities, dynamic creation and complex data types. The IF tool-suite includes a variety of tools for simulation, model checking and test generation. If2Obs is written in C++ and uses the basic libraries of IF for parsing and symbolic reachability of timed automata with deadlines.

If2Obs takes as main input the specification automaton, written in IF language, and generates an off-line observer. The observer takes as input a trace and the set of unobservable actions of the original IF specification. The observer can function either as an analog observer (default) or as a digital observer (-d option).

Case Study

Our case study is the Mars rover controller K9, and in particular its executive subsystem, developed at NASA Ames. It is an experimental platform for autonomous wheeled vehicles called rovers, targeted for the exploration of the Martian surface. K9 is specifically used to test out new autonomy software, such as the Rover Executive. The Rover Executive provides a flexible means of commanding a rover through plans that control the movement, experimental apparatus, and other resources of the Rover - also taking into account the possibiliy of failure of command actions.

The Rover executive is a software prototype written in C++ by researchers at NASA Ames. It is a multi-threaded program that consists of approximately 35,000 lines of C++ code, of which 9600 lines of code are related to actual functionality. The C++ code was manually translated into Java and C to experiment with tools using three technologies: static analysis, model checking and runtime analysis (Brat *et al.* 2003; Artho *et al.* 2003).

System Description The Rover executive is made up of :

- A main coordinating component named Executive. It provides the main control over how the plan is executed. Executive waits for a plan to be available, and signals at the end of the plan execution. So, the PlanWatcher signals when a plan is ready, and waits for end of execution to send a new plan.

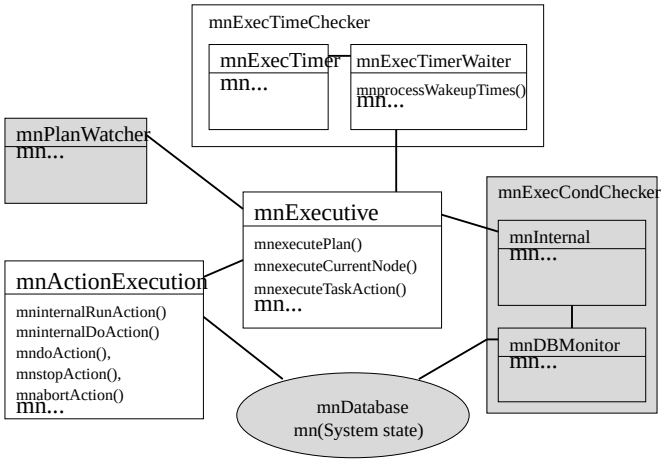


Figure 9: The K9 Rover Architecture.

- The component for monitoring the state condition ExecCondChecker consists of two threads, the database monitor DBMonitor just keeps watching for changes in the database and the thread Internal decides what needs to be done about it.
- ExecTimerChecker is the component for monitoring temporal conditions. It consists of two threads ExecTimer and ExecTimerWaiter. Both are respectively very similar to DBMonitor and Internal
- The ActionExecution thread is responsible for issuing the commands to the Rover. It consists of a list of methods internalDoAction, doAction, stopAction, and abortAction. ActionExecution runs the internalDoAction method, the other methods are just called by the Executive on its own thread by simple calls.
- Database simply receives calls to its methods. All accesses to the database through its methods are controlled by a lock.

Synchronization between these threads is performed through mutex and conditions variables.

Results Due to intellectual property restrictions, we did not have access to the execution platform of the K9 Rover. However, NASA provided us with a set of one hundred plans and traces, generated by the K9 Rover execution platform. We applied our method, using the plan-to-IF translator to obtain IF models for each plan, and If2Obs to generate an observer for each IF model of a plan. The observer was then used to check the traces.

One of the plans is shown in Figure 7. Time units in the plan are in seconds. A trace generated by the execution platform with input this plan is shown in Figure 10 (times in the trace are in milliseconds). The trace says that node 0 starts at time 922, node 1 starts at time 1932 and completes successfully at the same time, and so on. This trace does not conform to the specification, because the latter requires that node 2 finishes at least 1 second after it starts (fifth line of the block of node 2 in the plan). If2Obs takes a few seconds to generate the observer (this is a C++ code generation

```
start node0 922
start node1 1932
success node1 1932
start node2 2942
success node2 2942
success node0 2942
```

Figure 10: A trace corresponding to the plan of Figure 7.

process, compilation and linking with IF libraries). The observer needs less than a second to qualify this claim as non-conforming. In general, all traces were checked in a matter of seconds.

Conclusions and future work

We have proposed a methodology for testing conformance of an important class of real-time applications in an automatic way. The class includes all applications for which a specification is available and can be translated into a network of timed automata. In particular, the class includes robotic applications where specifications are considered to be the plans describing the robot mission. Such plans can be automatically translated to timed automata (A. Akhavan & Orfanidou 2004).

The method relies on the automatic generation of an observer from the specification, on the one hand, and on the instrumentation of the system to be tested, on the other hand. The testing process consists in feeding the traces generated by the instrumented system to the observer, which is a testing device, used to check conformance of a trace to the specification. We have validated the approach on the NASA K9 Rover case study.

Regarding future work, we plan to study the instrumentation and trace generation problems. As mentioned above, instrumentation should be possible to automate, by identifying a mapping between execution platform events and specification events, and automatically scanning the code, adding event/time-stamp exporting commands to the identified platform events. Trace generation has a lot of similarities to test-case generation, since the system under test must be run a number of times, with different inputs, to obtain a set of traces. Input coverage and other techniques can be employed here to obtain an adequate set of traces. In fact, it is an interesting question how to define coverage in the case where the inputs are plans.

We also plan to study the representation of observers as finite automata (timed or untimed). This is not always possible, because timed automata are non-determinizable in general (Alur & Dill 1994). Moreover, checking whether a particular TA is determinizable and determinizing it is algorithmically impossible in general (Tripakis 2004). Identifying classes of TA (e.g. those generated from plans) for which the above problems are solvable is a possible step in this direction.

Finally, we envisage extending our approach to other high-level specification languages and experimenting with more case studies.

Acknowledgments We would like to thank Klaus Havelund and Rich Washington from NASA for their help with the instrumentation of the K9 Rover application. Also, Dimitra Giannakopoulou from NASA for explaining the application.

References

A. Akhavan, S. Bensalem, M. B., and Orfanidou, E. 2004. Experiment on verification of a planetary rover controller. In *4th International Workshop on Planning and Scheduling for Space*.

Alur, R., and Dill, D. 1994. A theory of timed automata. *Theoretical Computer Science* 126:183–235.

Alur, R.; Kurshan, R.; and Viswanathan, M. 1998. Membership problems for timed and hybrid automata. In *19th IEEE Real-Time Systems Symposium (RTSS'98)*.

Artho, C.; Drusinsky, D.; Goldberg, A.; Havelund, K.; Lowry, M.; Pasareanu, C.; Rosu, G.; and Visser, W. 2003. Experiments with test case generation and runtime analysis. In *10th International Workshop on Abstract State Machines (ASM'03)*.

Barringer, H.; Goldberg, A.; Havelund, K.; and Sen, K. 2003a. Eagle does Space Efficient LTL Monitoring. Submitted for publication.

Barringer, H.; Goldberg, A.; Havelund, K.; and Sen, K. 2003b. Eagle Monitors by Collecting Facts and Generating Obligations. Submitted for publication.

Barringer, H.; Goldberg, A.; Havelund, K.; and Sen, K. 2003c. Rule-Based Runtime Verification. In *Proceedings of Fifth International Conference on Verification, Model Checking and Abstract Interpretation, January 2004 – to appear in LNCS*.

Bensalem, S., and Havelund, K. 2003. Deadlock Analysis of Multi-threaded Java Programs. Submitted for publication.

Bensalem, S.; Bozga, M.; Krichen, M.; and Tripakis, S. 2005. Testing conformance of real-time applications: Case of planetary rover controller. In *International Workshop on Verification and Validation of Model-Based Planning and Scheduling Systems*.

Bornot, S.; Sifakis, J.; and Tripakis, S. 1998. Modeling urgency in timed systems. In *Compositionality*, volume 1536 of *LNCS*. Springer.

Bozga, M.; Fernandez, J.; Ghirvu, L.; Graf, S.; Krimm, J.; and Mounier, L. 2000. IF: a validation environment for timed asynchronous systems. In *Proc. CAV'00*, volume 1855 of *LNCS*. Springer.

Brat, G.; Giannakopoulou, D.; Goldberg, A.; Havelund, K.; Lowry, M.; Pasareanu, C.; Venet, A.; and Visser, W. 2003. Experimental evaluation of V&V tools on martian rover software. In *SEI Software Model Checking Workshop*.

Dill, D. 1989. Timing assumptions and verification of finite-state concurrent systems. In Sifakis, J., ed., *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, 197–212. Springer–Verlag.

Drunsinsky, D. 2000. The temporal rover and the ATG rover. In *7th SPIN workshop*, volume 1885 of *LNCS*, 323–330. Springer.

Goldman, R. 2002. Model-based planning and real-time execution in the CIRCA framework. In *AI Planning and Scheduling (AIPS'02)*.

Havelund, K., and Roşu, G. 2001. Monitoring Java Programs with Java PathExplorer. In Havelund, K., and Roşu, G., eds., *Proceedings of the First International Workshop on Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*, 97–114. Paris, France: Elsevier Science.

Henzinger, T.; Manna, Z.; and Pnueli, A. 1992. What good are digital clocks? In *ICALP'92*, LNCS 623.

Ingrand, F.; Chatila, R.; Alami, R.; and Robert, F. 1996. PRS: A high level supervision and control language for autonomous mobile robots. In *IEEE Intl. Conf. on Robotics and Automation*.

Konolige, K.; Myers, K. L.; Ruspini, E. H.; and Saffiotti, A. 1997. The Saphira architecture: A design for autonomy. *Journal of experimental & theoretical artificial intelligence* 9(1):215–235.

Krichen, M., and Tripakis, S. 2004. Black-box conformance testing for real-time systems. In *11th International SPIN Workshop on Model Checking of Software (SPIN'04)*, volume 2989 of *LNCS*. Springer.

Lyons, D. 1993. Representing and analyzing action plans as networks of concurrent processes. *IEEE Transactions on Robotics and Automation*.

Muscettola, N.; Dorais, G.; Fry, C.; Levinson, R.; and Plaunt, C. 2002. IDEA: Planning at the core of autonomous reactive agents. In *AI Planning and Scheduling (AIPS'02)*.

Peterson, J.; Hager, G.; and Hudak, P. 1999. A language for declarative robotic programming. In *IEEE Conf. on Robotics and Automation*.

Sifakis, J., and Yovine, S. 1996. Compositional specification of timed systems. In *13th Annual Symposium on Theoretical Aspects of Computer Science, STACS'96*, volume 1046 of *LNCS*. Spinger-Verlag.

Simmons, R., and Apfelbaum, D. 1998. A task description language for robot control. In *Intelligent Robotics and Systems*.

Tripakis, S. 2002. Fault diagnosis for timed automata. In *Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT'02)*, volume 2469 of *LNCS*. Springer.

Tripakis, S. 2004. Folk theorems on the determinization and minimization of timed automata. In *Formal Modeling and Analysis of Timed Systems (FORMATS'03)*, volume 2791 of *LNCS*. Springer.

Zhu, H.; Hall, P.; and May, J. 1997. Software unit test coverage and adequacy. *ACM Computing Surveys* 29(4).

A Factored Symbolic Approach to Reactive Planning*

Seung H. Chung and **Brian C. Williams**
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
77 Massachusetts Ave.
Cambridge, MA 02139
{chung,williams}@mit.edu

Abstract

Autonomous systems in uncertain dynamic environments must reconfigure themselves in response to unanticipated events and goals in real-time. To provide a high assurance of real-time embedded systems, fault-aware executable specification and verification of this fault-aware specification are necessary. We present a method for synthesizing an executable code from a fault-aware specification. We approach the problem by framing it as model-based reactive planning. Reactive plans are susceptible to exponential state space explosion. We address this problem through *transition-based decomposition* by generating compact *decomposed goal-directed plans*. We further minimize state explosion by adopting a symbolic representation based on Ordered Binary Decision Diagrams. We demonstrate our reactive planner on representative spacecraft subsystem models.

Introduction

Recent failures in NASA's Mars exploration program point to the need for increased autonomous response in spacecraft. The presumed cause of failure for the Mars Polar Lander (MPL) Mission (Casani *et al.* 2000) provides a relevant example. During the final stage of MPL's descent to the Martian surface, one sensor wrongfully signaled the landing of the spacecraft. As a result, the descent engines were prematurely shut off, causing MPL to crash into the surface. During this incident, no communication was possible between MPL and ground control. Even if it were, the outcome would likely have been inevitable due to the communication time delay: when Mars is closest to Earth, commands from the ground take at least 12 minutes to reach the spacecraft. Thus, onboard reactive software is necessary to execute their functions while responding to failures and anomalies.

High assurance of real-time embedded systems is being achieved through the application of executable specification, which requires executable code synthesis from specification and verification of the specification itself. However, with unforseen failures and anomalies, as in the case of MPL Mission, the executable specification must be extended to be fault aware. Hence, code synthesis must be extended to handle faults, and this fault aware specification must be formally verified. This paper presents a method for the former, i.e. automated synthesis of fault tolerant, executable code, by framing the problem as a form of model-based reactive planning.

*This work was supported in part by NASA's Cross Enterprise Technology Development program under contract NAG2-1466, DARPA's MOBIES program under contract F33615-00-C-1702, and NASA Graduate Student Research Program Fellowship.

Motivation for Tractable Reactive Planning

While general-purpose onboard planners could be used for autonomous reconfiguration, due to the PSPACE-complete nature of planning problems, real-time response cannot be guaranteed. In time-critical situations, such as the MPL landing scenario, late response could be disastrous to the mission. Reactive planning is an approach that guarantees real-time response. A reactive planner precompiles a plan offline for all possible situations, and then executes the plan online.

In general, a reactive planner may not be able to optimize resource utilization. However, the irreversibility associated with the use of nonrenewable resources requires careful deliberation to ensure system safety and mission success:

Requirement 1. *A reactive planner shall consider only reversible control actions, unless the effect is to repair failures* (Williams & Nayak 1997).

One of the early approaches to reactive planning is universal planning, first introduced by (Schoppers 1987). Though a universal plan can react to a nondeterministic environment, it cannot react to rapidly changing goals. Furthermore, (Ginsberg 1989) pointed out the intractability of universal planning due to the exponential state space explosion problem. Thus, a new reactive planning approach is necessary.

Handling State Explosion through Decomposition

The method of divide-and-conquer is a well known effective approach to solving problems. Based on this principle, we have developed a new transition-based decomposition method for reactive planning. Though this method is unrelated to the structural decomposition used in constraint satisfaction problems (CSP) (Gottlob, Leone, & Scarcello 1999), the contribution of our decomposition method to planning is analogous to that of the structural decomposition methods for CSP.

CSPs are known to be NP-complete, but (Freuder 1985) has shown that a CSP with a tree-structured constraint graph is solvable in linear time. Similarly, (Williams & Nayak 1997) have shown that if a planning problem has an acyclic dependency, then the problem can be solved within a state space that grows only linear in the number of state variables. For CSPs that do not have tree-structured constraint graph, Dechter and Pearl have shown that the constraint graphs of those problems can be transformed into tree-structured graphs using a tree decomposition technique (Dechter & Pearl 1989). In our approach, for planning problems with cyclic transition dependency graph (TDG), we use transition-based decomposition to transform the cyclic TDG to an acyclic TDG. As a result, even planning problems with cyclic TDG can be solved within a state space that grows approximately linear in the number of state variables.

Handling State Explosion through Symbolic Representation (OBDD)

Through transition-based decomposition, our reactive planner divides a problem into a set of subproblems. While transition-based decomposition addresses the state explosion problem at the global level, we also address this issue at the subproblem level by adopting a symbolic representation.

The logic synthesis and model checking communities have been using Ordered Binary Decision Diagrams (OBDD) (Bryant 1986) for compact state space encoding. An OBDD-based model checking technique has proven particularly successful in dealing with the state explosion problem (Burch *et al.* 1992). Recognizing the similarities between model checking and planning, (Cimatti *et al.* 1997) introduced a new universal planning technique that takes advantage of the OBDD. Since then, several OBDD-based universal planning algorithms have been introduced for operating within nondeterministic domains (Cimatti, Roveri, & Traverso 1998b; 1998a; Jensen 1999). In our approach, we also take advantage of OBDD, but unlike the universal planners, our reactive planner generates *goal-directed plans* (GDP) that can react to the nondeterministic environment as well as rapidly changing goals. We generate these GDPs for each decomposed subproblem such that the resulting *decomposed goal-directed plan* (DGDP) conforms to Requirement 1 while guaranteeing real-time response.

Spacecraft Communication System Example

Throughout the paper, we will use a model of a simplified spacecraft communication system (Figure 1) to present our decomposed symbolic approach to reactive planning. Figure 1 depicts the direction of signal flow among components. The computer sends data to be transmitted through the bus control. When the data is received, the bus control routes it to the transmitter. The transmitter receives the data and generates the corresponding signal. The signal is amplified by the amplifier and is finally transmitted through the antenna. The computer is also responsible for controlling the devices: it may command either the transmitter or amplifier to be turned on or off. Again, these commands are sent to the appropriate devices via the bus control.

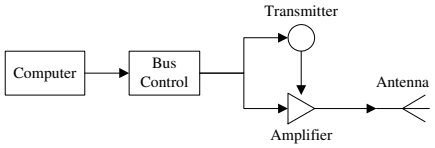


Figure 1: Simplified spacecraft communication system.

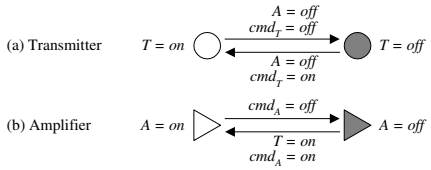


Figure 2: Concurrent automata of a transmitter and an amplifier. Idle transitions are omitted for clarity.

Modelling Behavior with Concurrent Automata

We model a system of concurrently operating components by a set of *concurrent automata*. Figure 2 illustrates the concurrent automata of a transmitter and an amplifier. The transitions between states are conditioned on commands (e.g. $cmd_T = \mathit{off}$) and states of other automata. For instance, the amplifier must be turned off ($A = \mathit{off}$) before we can command the transmitter on or off. This particular condition is necessary for the safety of the system, as the process of switching the transmitter on or off may generate a transient signal spike that could damage the amplifier. For the same reason, the transmitter must be turned on before the amplifier can be turned on. We define concurrent automata formally as follows:

Definition 1 *A set of* concurrent automata $\mathcal{CA} = \{\mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \ldots, \mathcal{A}^{(n)}\}$ *is composed of concurrently operating finite automata. Each concurrent automaton $\mathcal{A}^{(i)}$ is a 3-tuple $\langle Q^{(i)}, \Sigma^{(i)}, \delta^{(i)} \rangle$, where $Q^{(i)}$ is a finite set of states, $\Sigma^{(i)}$ is a finite set of inputs (either commands or states of other concurrent automata) and $\delta^{(i)} : Q^{(i)} \times \Sigma^{(i)} \rightarrow Q^{(i)}$ is a transition function.*

Representing $\mathcal{CA}$ Symbolically

For compactness, we encode the concurrent automata in an OBDD representation. In this representation, state s of concurrent automaton i is defined by a vector of $\log_2(|Q^{(i)}|)$ distinct Boolean variables, where $|Q^{(i)}|$ is the number of elements in $Q^{(i)}$. Similarly, the input a is represented as a vector of Boolean variables. The transition relation for concurrent automaton i is $R^{(i)} : Q^{(i)} \times \Sigma^{(i)} \times Q^{(i)} \rightarrow \mathcal{B}$, where $\mathcal{B}$ is a set of Boolean values and $R^{(i)}(s, a, s') = (s' \in \delta^{(i)}(s, a))$, and s' indicates the state at the next time step. For example, the transition relation $R^{(A)}$ of the amplifier A is as follows:

$$
\begin{aligned}
&[((A = \mathit{off}) \wedge \neg((T = on) \wedge (cmd_A = on))) \Rightarrow (A' = \mathit{off})] \quad \wedge \\
&[((A = \mathit{off}) \wedge (T = on) \wedge (cmd_A = on)) \Rightarrow (A' = on)] \quad \wedge \\
&[((A = on) \wedge (cmd_A = \mathit{off})) \Rightarrow (A' = \mathit{off})] \quad \wedge \\
&[((A = on) \wedge \neg(cmd_A = \mathit{off})) \Rightarrow (A' = on)]
\end{aligned}
$$

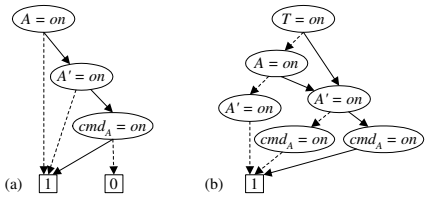


Figure 3: OBDD representation of (a) $((A = on) \wedge (cmd_A = \mathit{off})) \Rightarrow (A' = \mathit{off})$ and (b) amplifier transition relation $R^{(A)}$.

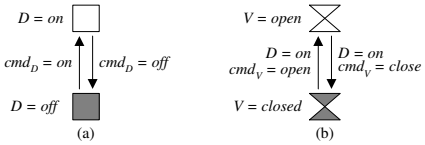


Figure 4: Concurrent automata for (a) Driver, and (b) Valve.

Figure 3(a) illustrates the OBDD representation of the transition $((A = on) \wedge (cmd_A = \mathit{off})) \Rightarrow (A' = \mathit{off})$. Figure 3(b) shows the result of conjoining the OBDDs of the transitions into the transition $R^{(A)}$[1]. Each node of an OBDD represents a Boolean variable, and the dotted and the solid outgoing edges respectively represent false and true evaluations of the Boolean variable. The terminal nodes 1 and 0 represent the evaluation of the Boolean function (i.e. OBDD) where each path from the root to a terminal evaluates to 1 for true or 0 for false. In Figure 3(b), all paths that lead to false have been omitted for simplicity. One of the benefits of using OBDDs to represent transition relations is in relative compactness of OBDDs. (Cimatti, Roveri, & Traverso 1998a) shows that the size of an OBDD does not necessary depend on the number of states, but rather on the structure of the information the OBDD encodes.

Subgoal Serialization through Transition-based Decomposition

A set of subgoals are serializable if and only if a goal can be partitioned into a set of subgoals that can be solved sequentially to achieve the goal (Korf 1987). For example, consider the driver and valve shown in Figure 4. The driver is a device that commands the valve open or closed. Thus, the driver must be on $(D = \mathit{on})$, before the valve can be commanded open or closed. Presume that the current state of the driver and valve system is $(D = \mathit{off}, V = \mathit{closed})$ and the goal state to achieve is $(D = \mathit{off}, V = \mathit{open})$. In this case, we do not have to figure out how to achieve $(D = \mathit{off})$ and $(V = \mathit{open})$ simultaneously. Rather, we can figure out how to achieve $(V = \mathit{open})$ first. Once $(V = \mathit{open})$ has been achieved, we can then figure out how to achieve $(D = \mathit{off})$, without worrying about potential impact on the $(V = \mathit{open})$ subgoal. Hence, the subgoals are serializable.

[1]In this example, we assume cmd_T and cmd_A are on or off at all times.

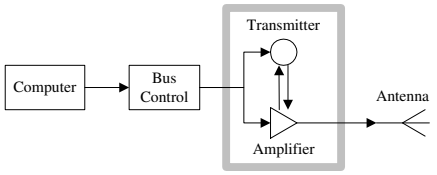


Figure 5: Cyclic transition dependency graph of a spacecraft communication subsystem.

(Williams & Nayak 1997) recognized that a set of subgoals are serializable if the transition dependency graph (TDG) of a system is acyclic, where TDG of a concurrent automata is formally defined as follows:

Definition 2 *A* transition dependency graph $\mathcal{G}$ *of* $\mathcal{CA}$ *is a directed graph whose vertices are the concurrent automata* $\{\mathcal{C}\}$. $\mathcal{G}$ *contains a directed edge from vertex* $\mathcal{C}^{(i)}$ *to vertex* $\mathcal{C}^{(j)}$*, if* $\mathcal{C}^{(i)}$ *occurs in the antecedent (precondition) of one of* $\mathcal{C}^{(j)}$*'s transitions.*

For the driver and valve, the valve's ability to open or close depends on the state of the driver. The driver, however, does not depend on the valve, so we can change the driver state without affecting the valve state. Hence, the dependency relationship is acyclic.

For a system with a cyclic TDG, we can transform it into an acyclic graph through transition-based decomposition. For example, TDG of the communication system is cyclic, as shown in Figure 5. However, if we group the transmitter and the amplifier, and consider them as a single vertex in TDG, the resulting graph is acyclic. We recognize that a set of cyclic vertices in TDG directly corresponds to a strongly connected component (SCC) of the TDG.

Goal-directed Plan

With the TDG decomposed into a set of SCCs, we can generate a GDP for each SCC individually. As the automata within a SCC have cyclic dependency, we must consider the concurrency and interdependence of the automata. With this in mind, we first compose the automata within a SCC into a single automaton. Then, we generate a GDP based on the composed automaton.

Composing Automata

Continuing with the transmitter/amplifier example, we want to construct a single automaton that represents both components, as shown in Figure 6. Notice that one transition seems missing, the transition from $(T = on, A = \mathit{off})$ to $(T = \mathit{off}, A = on)$. According to the model shown in Figure 2, such a transition may occur if the transmitter is commanded off $(cmd_T = \mathit{off})$ and the amplifier is commanded on $(cmd_A = on)$ simultaneously. In controlling concurrent devices, however, such synchronized control cannot be guaranteed; in fact, such commanding is nearly impossible, and taking such an action could be hazardous, as the amplifier may be damaged if $(cmd_A = on)$ precedes $(cmd_T = \mathit{off})$ even by a fraction of a second. Thus, before composing automata, we modify the transition relation for each automaton

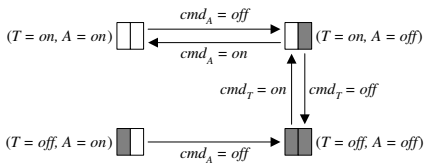


Figure 6: Automaton of composed transmitter and amplifier automata. Idle transitions are omitted for clarity.

Current State	Goal State			
	T=on,A=on	*T=on,A=off*	*T=off,A=off*	*T=off,A=on*
T=on,A=on	*idle*	cmd_A=*off*(1)	cmd_A=*off*(2)	*failure*
T=on,A=off	cmd_A=*on*(1)	*idle*	cmd_T=*off*(1)	*failure*
T=off,A=off	cmd_T=*on*(2)	cmd_T=*on*(1)	*idle*	*failure*
T=off,A=on	cmd_A=*off*(3)	cmd_A=*off*(2)	cmd_A=*off*(1)	*idle*

Figure 7: Goal-directed plan for the transmitter/amplifier system. The number next to each command represents the total number of steps necessary to achieve the goal.

to avoid such hazards. If a transition is conditioned on the state of another automaton, we require that the state condition be true before and after the transition occurs.

For example, consider the amplifier's transition from off to on:

$$((A = \mathit{off}) \wedge (T = on) \wedge (cmd_A = on)) \Rightarrow (A' = on)$$

The transition relies on the transmitter being on $(T = on)$. Thus, we modify the transition to guarantee that the transmitter is on before and after:

$$((A = \mathit{off}) \wedge (T = on) \wedge (T' = on) \wedge (cmd_A = on)) \Rightarrow (A' = on)$$

With the modified transition relations, composing the concurrent automata into a single automaton is trivial. The composed transition relation R_{SCC} of a SCC is

$$R_{SCC} = \bigwedge_{C^{(i)} \in SCC} R^{(i)}$$

Generating the Goal-directed Plan

A GDP is comprised of a set of goal-directed rules, where a goal-directed rule is a 3-tuple $\langle s, a, s' \rangle$. A goal-directed rule can be interpreted as "if the current state is s and the goal state is s', execute a". Figure 7 is a tabular representation of the goal-directed plan for the transmitter/amplifier SCC. Each entry in the table corresponds to a goal-directed rule. While a in this GDP is only composed of commands, a may in general contain states of other automata that precede the SCC in the dependency ordering. For example, one of the goal-directed rules for the valve is

$$\langle (V = open), (D = on, cmd_V = close), (V = closed) \rangle.$$

$(D = on)$ is an *intermediate subgoal* that must be achieved before we can command the valve closed.

We generate the GDP by iteratively searching the state space in parallel, backward, and breadth-first manner. With OBDDs, states within the search space do not have to be enumerated; instead, we can generate goal-directed rules of all goals and initial states simultaneously, thus "in parallel". The search method is also characterized as a "backward search", as the GDP is generated by searching for the states that can reach the goal, instead of searching for the goals that can be reached from the current state. Of the goal-directed rules, we generate the one-step rules (i.e. goal-directed rules with goals that can be achieved in a single transition) first. In Figure 7, one-step rules are those with "(1)" next to the commands. Notice that one-step rules correspond directly to the transitions in transition relation. Next, we generate two-step rules, labelled "(2)" in Figure 7. We continue this process until the fixed-point is reached, thus "breadth-first". In general, the fixed-point of the iterative search is defined by the width of the transition graph of the automaton. In our transmitter/amplifier example, the fixed-point is reached after two iterations (i.e. after the three-step rules are generated). The algorithm for generating the GDP is as follows:

Algorithm 1 COMPUTEGDP(T)

1: $oldPlan \leftarrow \emptyset$
2: $newPlan \leftarrow T$
3: **while** $oldPlan \neq newPlan$ **do**
4: $\quad oldPlan \leftarrow newPlan$
5: $\quad newPlan \leftarrow oldPlan \cup$ COMPUTENEXTSTEPRULES$(T, oldPlan)$
6: **return** $newPlan$

The algorithm COMPUTEGDP takes the transition relation T of an automaton as its input. As we have discussed, the one-step rules are exactly the transition relation as reflected in line 2 of COMPUTEGDP. In lines 3–5, it iteratively searches for two-step rules, three-step rules, etc. while adding them to the newPlan. The procedure exits once the fixed-point is reached (line 3), and returns the plan (line 6).

In line 5, COMPUTENEXTSTEPRULES(T,oldPlan) generates n-step rules when the oldPlan contains all rules of less than n-steps. Assume that a relation $s_i \wedge a_j \Rightarrow s'_k$ is in the transition relation T and an $(n-1)$-step rule $\langle s_k, a_l, s'_m \rangle$ is in the old goal-directed plan oldPlan. Then, $\langle s_i, a_j, s'_m \rangle$ is one of the valid n-step rules returned by COMPUTENEXTSTEPRULES(T,P). For example, from the 1-step rule

$$\langle (T = \mathit{on}, A = \mathit{off}), (cmd_T = \mathit{off}), (T' = \mathit{off}, A' = \mathit{off}) \rangle$$

and the transition relation

$$((T = on, A = on) \wedge (cmd_A = \mathit{off})) \Rightarrow (T' = on, A' = \mathit{off})$$

the 2-step rule

$$\langle (T = on, A = on), (cmd_A = \mathit{off}), (T' = \mathit{off}, A' = \mathit{off}) \rangle$$

can be deduced.

Formally: COMPUTENEXTSTEPRULES(T,P) generates a set of n-step goal-directed rules $\langle s, a, s' \rangle$, where T is a transition relation and P is a goal-directed plan with only m-step rules, where $m < n$. Each rule $\langle s_i, a_j, s'_k \rangle$ is restricted such that $s'_l \subseteq (\mathsf{T} \wedge s_i \wedge a_j)$, $\langle s_l, a_m, s'_k \rangle \in \mathsf{P}$, and $\neg\exists a.\langle s_i, a, s'_k \rangle \in \mathsf{P}$.

$s'_l \subseteq (\mathsf{T} \wedge s_i \wedge a_j)$ states that s'_l must be reachable from state s_i through input a_j. The restriction $\neg\exists a.\langle s_i, a, s'_k \rangle \in \mathsf{P}$

says that $\langle s_i, a_j, s'_k \rangle$ cannot be a new goal-directed rule if a rule for the current state s_i and the goal state s'_k already exists in the plan P. With this restriction, the resulting GDP is guaranteed to be optimal, where an optimal plan is defined as a plan with the shortest control sequence. For example, while

$$\langle (T = on, A = off), (cmd_T = off), (T' = on, A' = on) \rangle$$

is a 3-step rule, it is not a valid rule since the optimal 1-step rule already exists in the plan:

$$\langle (T = on, A = off), (cmd_A = on), (T' = on, A' = on) \rangle$$

The algorithm for COMPUTENEXTSTEPRULES(T,P) is shown in algorithm 2. This algorithm leverages the OBDD representation to efficiently search the state space, without enumeration.

Algorithm 2 COMPUTENEXTSTEPRULES(T, P)

1: $nextPlan \leftarrow T_{[sTemp/s']} \wedge \exists a.P_{[sTemp/s]}$
2: $optimalNextPlanWithNoCmd \leftarrow \exists a.nextPlan - \exists a.P$
3: **return** $nextPlan \wedge optimalNextPlanWithNoCmd$

Line 1 of COMPUTENEXTSTEPRULES(T,P) computes all next step goal-directed rules including the non-optimal ones[2]. Line 2 determines which rules are the valid (i.e. optimal) rules. Finally, line 3 returns only those rules that are valid.

Decomposed Goal-directed Plan

Once the TDG is made acyclic, we can generate a GDP for each SCC successively, instead of generating a single GDP for the whole $\mathcal{CA}$. This set of GDPs for all SCCs in the system is called a *decomposed goal-directed plan* (DGDP). The advantage of this composition is that the footprint of the DGDP is much smaller than a single GDP for the full $\mathcal{CA}$. For example, let us assume that the number of concurrent automata, $|\mathcal{CA}|$, is n, and the average number of states per automaton, $|Q|$, is m. If we generate a single GDP for $\mathcal{CA}$, the number of states in GDP is exponential in $|\mathcal{CA}|$, $O(m^n)$. If the maximum number of automata in an SCC is w, however, the number of states in the corresponding DGDP is only $O(l \cdot m^w)$, where l is the total number of SCCs. Thus, even if the size of a $\mathcal{CA}$ grows, as long as m and w remains constant, the size of the corresponding DGDP grows only linearly in l. The algorithm for generating DGDP is of $\mathcal{CA}$ as follows:

where n is the number of composed automata (i.e. SCCs), R is an array of transition relations of the composed automata sorted in dependency order (i.e. inverse depth-first order of TDG), and q is an array of current states of the composed automata, also in dependency order. Lines 2–5 successively generates the GDP of each SCC in dependece order, storing an array of GDPs in DGDP (line 4). A

[2] [sTemp/s] symbolizes the replacement of variable s with variable sTemp.

Algorithm 3 COMPUTEDGDP(n, R, q)

1: $revReachAncs \leftarrow \emptyset$
2: **for** $i = 0$ to $(n - 1)$ **do**
3: $allwdR \leftarrow R[i] \wedge revReachAncs$
4: $DGDP[i] \leftarrow$ COMPUTEGDP$(allwdR)$
5: $revReachAncs \leftarrow revReachAncs \cup$ COMPUTERRS$(R[i], q[i])$
6: **return** $DGDP$

GDP is computed from a subset of the SCC transition relation, allwdR, where the subset is restricted to the transitions whose antecedents (subgoals in GDP) are reversibly reachable from the current state q. This restriction guarantees the aforementioned requirement 1. In line 5, the reversibly reachable states of the i-th SCC are generated and added to the set of reversibly reachable ancestor states revReachAncs to be used in the next iteration.

DGDP Execution

Figure 8 shows a DGDP for a driver and a valve. We execute DGDP in inverse dependency order (e.g. the valve then the driver). For example, let us assume the driver and valve are off and closed, respectively, and the goal is to turn off the driver and open the valve. First, we must attempt to open the valve, according to the inverse dependency order. To switch the valve open from the closed position, the driver must be on and the valve must be commanded open as shown in Figure 8. Here, $(D = on)$ is a subgoal that must be achieved before executing the command $(cmd_V = open)$. As the driver is currently off, we determine how to turn the driver on by looking up the driver plan in Figure 8. According to the plan, we simply command the driver on $(cmd_D = on)$. Once the driver is turned on, then we can command the valve to open $(cmd_V = open)$. Once the valve is opened, then the drive can be turned off again. (Williams & Nayak 1997) discuss DGDP execution algorithm in detail. The incremental nature of the algorithm allows for robust execution that immediately responds to failures or sudden changes in goals.

Conclusion

Our decomposed symbolic approach to reactive planning is novel in two ways. First, it leverages transition-based decomposition to eliminate the state space explosion problem in reactive planning. When transition-based decomposition is used to solve a problem, the complexity of the problem becomes linear in the size of the SCCs instead of being exponential in the size of $\mathcal{CA}$. As long as the size of the SCCs remains relatively small, the problem remains tractable. Second, we incorporate the use of OBDDs into reactive planning, which gives us two distinct advantages: (1) we can search the state space without the need to enumerate the states, and (2) we can take advantage of the OBDD's compact state space encoding capability.

References

Bryant, R. E. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*

Current State	Goal State	
	D=on	*D=off*
D=on	*idle*	cmd_D *=off*
D=off	cmd_D *=on*	*idle*

Current State	Goal State	
	D=on	*D=off*
V=open	*idle*	*D=on* cmd_V *=close*
V=closed	*D=on* cmd_V *=open*	*idle*

Figure 8: Factored goal-directed plan for a valve driver (left) and a valve (right).

C-35(8):677–691.

Burch, J. R.; Clarke, E. M.; McMillan, K. L.; Dill, D. L.; and Hwang:, L. J. 1992. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation* 98(2):142–170.

Casani, J.; Whetsler, C.; Albee, A.; Battel, S.; Brace, R.; Burdick, G.; Burr, P.; Dippoey, D.; Lavell, J.; Leising, C.; MacPherson, D.; Menard, W.; Rose, R.; Sackheim, R.; and Schallenmuller, A. 2000. Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions. Technical Report JPL D-18709, Jet Propulsion Laboratory, California Institute of Technology.

Cimatti, A.; Giunchiglia, F.; Giunchiglia, E.; and Traverso, P. 1997. Planning via Model Checking: A Decision Procedure for $\mathcal{AR}$. In *Proceedings of the Fourth European Conference on Planning (ECP'97)*.

Cimatti, A.; Roveri, M.; and Traverso, P. 1998a. Automatic OBDD-based Generation of Universal Plans in Non-Deterministic Domains. In *Prodeedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*.

Cimatti, A.; Roveri, M.; and Traverso, P. 1998b. Strong Planning in Non-Deterministic Domains via Model Checking. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS'98)*.

Dechter, R., and Pearl, J. 1989. Tree Clustering for Constraint Networks. *Artificial Inteligence* 38(3):353–366.

Freuder, E. C. 1985. A Sufficient Condition for Backtrack-Bounded Search. *Journal of the ACM (JACM)* 32(4):755–761.

Ginsberg, M. L. 1989. Universal Planning: An (Almost) Universally Bad Idea. *AI Magazine* 10(4):40–44.

Gottlob, G.; Leone, N.; and Scarcello, F. 1999. A Comparison of Structural CSP Decomposition Methods. In Dean, T., ed., *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99)*, 394–399.

Jensen, R. M. 1999. OBDD-based Universal Planning in Multi-Agent, Non-Deterministic Domains. Master's thesis, Technical University of Denmark.

Korf, R. E. 1987. Planning as Search: A Quantitative Approach. *Artificial Intelligence* 33(1):65–68.

Schoppers, M. J. 1987. Universal Plans for Reactive Robots in Unpredictable Environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI'87)*, volume 2, 1039–1046.

Williams, B. C., and Nayak, P. P. 1997. A Reactive Planner for a Model-based Executive. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*.

Validating the Autonomous EO-1 Science Agent

Benjamin Cichy, Steve Chien, Steve Schaffer, Daniel Tran, Gregg Rabideau, Rob Sherwood

Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Dr.
Pasadena, CA 91109
Firstname.Lastname@jpl.nasa.gov

Abstract. This paper describes the validation process for the Autonomous Sciencecraft Experiment, a software agent currently flying onboard NASA's EO-1 spacecraft. The agent autonomously collects, analyzes, and reacts to onboard science data. The agent has been designed using a layered architectural approach with specific redundant safeguards to reduce the risk of agent malfunction to the EO-1 spacecraft. This "safe" design has been thoroughly validated by informal validation methods supplemented by sub-system and system-level testing. This paper describes the analysis used to define agent safety, elements of the design that increase the safety of the agent, and the process used to validate agent safety.

1 Introduction

Autonomy technologies have incredible potential to revolutionize space exploration. In the current mode of operations, space missions involve meticulous ground planning significantly in advance of actual operations. In this paradigm, rapid responses to dynamic science events can require substantial operations effort. Artificial Intelligence technologies enable onboard software to detect science events, replan upcoming mission operations, and enable successful execution of re-planned responses. Additionally, with onboard response, the spacecraft can acquire data, analyze it onboard to estimate its science value, and react autonomously to maximize science return. For example, our Autonomous Science Agent can monitor active volcano sites and schedule multiple observations when an eruption has been detected. Or monitor river basins, and increase imaging frequency during periods of flooding.

However, building autonomy software for space missions has a number of key challenges; many of these issues increase the importance of building a reliable, safe, agent.

1. Limited, intermittent communications to the agent. A typical spacecraft in low earth orbit (such as EO-1) has 8 10-minute communications opportunities per day. This means that the spacecraft must be able to operate for long periods of time without supervision. For deep space missions the spacecraft may be in communications far less frequently. Some deep space missions only contact the spacecraft once per week, or even once every several weeks.
2. Spacecraft are very complex. A typical spacecraft has thousands of components, each of which must be carefully engineered to survive rigors of space (extreme temperature, radiation, physical stresses). Add to this the fact that many components are one-of-a-kind and thus have behaviors that are hard to characterize.
3. Limited observability. Because processing telemetry is expensive, onboard storage is limited, and downlink bandwidth is limited, engineering telemetry is limited. Thus onboard software must be able to make decisions on limited information and ground operations teams must be able to operate the spacecraft with even more limited information.
4. Limited computing power. Because of limited power onboard, spacecraft computing resources are usually very constrained. An average spacecraft CPUs offer 25 MIPS and 128 MB RAM – far less than a typical personal computer. Our CPU allocation for ASE on EO-1 is 4 MIPS and 128MB RAM.
5. High stakes. A typical space mission costs hundreds of millions of dollars, any failure has significant economic impact. The total EO-1 Mission cost is over $100 million dollars. Over financial cost, many launch and/or mission opportunities are limited by planetary geometries. In these cases, if a space mission is lost it may be years before another similar mission can be launched. Additionally, a space mission can take years to plan, construct the spacecraft, and reach their targets. This delay can be catastrophic.

This paper discusses our efforts to build and validate a safe autonomous space science agent. The principal contributions of this paper are as follows:

1. We describe our layered agent architecture and how it provides a framework for agent safety.
2. We describe our knowledge engineering and model review process including identification of safety risks and mitigations.

3. We describe our testing process designed to validate the safe design of our agent's architecture and model.

We describe these areas in the context of the Autonomous Sciencecraft Experiment (ASE), an autonomy software package adapted to NASA's New Millennium Earth Observer One (EO-1) spacecraft [4] from a design originally proposed for flight on the Air Force's Techsat-21 Mission [2].

2 Autonomy Architecture

The autonomy software on EO-1 is organized as a traditional three-layer architecture [8] (See Figure 1.). At the top layer, the Continuous Activity Scheduling Planning Execution and Replanning (CASPER) system [3, 12] plans activities to achieve long-term mission objectives. CASPER submits the planned sequences of activities to the Spacecraft Command Language (SCL) system [10] for execution. Using an internal model, SCL expands the activities into sequences of EO-1 commands, which are then executed through the EO-1 Flight Software (FSW).

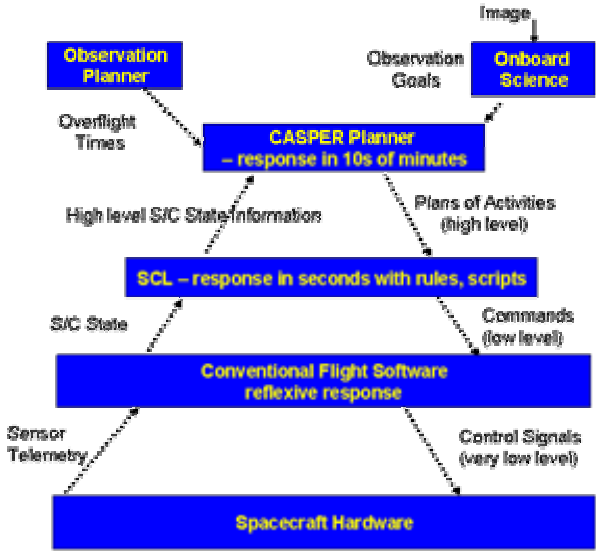


Figure 1. Autonomy Software Architecture

Operating on the tens-of-minutes timescale, CASPER responds to events that have multiple-orbit effects, including scheduling science observations and ground contacts. CASPER commands activities traditionally initiated through sequences uplinked by the EO-1 ground operations team. Consulting internal models of the spacecraft, CASPER searches for plans that combine these basic activities to satisfy high-level goals consistent with spacecraft operational and resource constraints.

Plans generated by CASPER are given to SCL at this basic-activity granularity. SCL expands the CASPER plan to detailed sequences of EO-1 spacecraft commands. Operating on the several-second timescale, SCL responds to events that have local effects, but require immediate attention and a quick resolution. SCL performs activities using scripts to expand activities and rules that monitor and enforce flight constraints.

SCL sends commands to the EO-1 FSW [9], the basic flight software that operates the EO-1 spacecraft. The interface from SCL to the EO-1 FSW is at the same level as ground generated command sequences – in other words the FSW does not know, or care, whether commands were issued by SCL or EO-1 ground operations.

SCL implements the commanding interface through a special component called the Autonomy Flight Software Bridge (FSB). The FSB takes autonomy software messages and issues corresponding FSW commands. The FSB also implements a new set of FSW commands to perform functions such as startup and shutdown of the autonomy software. This single interface point allows the EO-1 operations team to easily turn on and off the ASE commanding path, and thus ASE control, of EO-1.

The FSW accepts low level spacecraft commands. These commands can be either stored command loads uploaded from the ground (e.g. ground planned sequences) or real-time commands (such as commands from the ground during an uplink pass). The autonomy SW commands appear to the FSW as real-time commands. As part of its core, the FSW has a full fault and spacecraft protection functionality designed to:

1. Reject commands (from any source) that would endanger the spacecraft.
2. Execute pre-determined sequences to enter a "safe" mode upon detection of a hazardous state thereby stabilizing the spacecraft for ground assessment and reconfiguration.

For example, if a sequence issues commands that point the spacecraft imaging instruments at the sun, the fault protection software will abort the pointing activity; if a sequence issues commands that would expend power to unsafe levels, the fault protection software will shut down non-essential subsystems (such as science instruments) and orient the spacecraft to maximize solar power generation. While the intention of the fault protection is to cover all potentially hazardous scenarios, it is understood that the fault protection software is not foolproof. Thus, there is a strong desire to not command the spacecraft into any hazardous situation even if it is believed that the fault protection will protect the spacecraft.

Finally, the ASE software package includes a suite of science analysis algorithms. These algorithms process, interpret, and suggest reactions to onboard science observations. CASPER converts the science analysis suggestions to activities, and adds them to the onboard schedule for execution.

This layered architecture enables each lower layer to validate the output of the higher layers – SCL checks CASPER activities prior to sending the corresponding commands to the FSW, while the FSW fault protection checks the command sequences from SCL. This multiple-layer safety check emboldens confidence in the safety of our agent.

3 Model Building & Validation

Both CASPER and SCL rely on high-fidelity internal models of the EO-1 spacecraft. CASPER uses these models to delineate what goals can be achieved, and the scope of possible reactions. SCL uses models to generate command sequences and monitor activity execution. Any inaccuracies in these models could lead to ASE failing to achieve science objectives, or in the extreme, issuing unsafe sequences of commands. As such, these models were the product of a methodical development and review process designed to ensure they correctly encoded the relevant operational and safety constraints of EO-1.

The CASPER and SCL models share many of the same EO-1 constraints – including properties of physical subsystems, operation modes, valid command sequences, command prerequisites, and impacts of commands on spacecraft state. As a general rule however, CASPER models EO-1 at a higher level of abstraction than SCL. The activities commanded by CASPER are more abstract usually requiring tens or hundreds of spacecraft commands to achieve. Conversely, SCL activities sometimes expand to only a few spacecraft commands.

CASPER models the basic activities that must be assembled to complete the high-level mission goals including science observations and downlinks. The decomposition from goals to activities continues until a suitable level is reached for planning – a level that allows CASPER to model spacecraft state and its progression over time, discrete states such as instrument modes, and resources such as memory available for data storage. At this level of abstraction CASPER can commit activities in order to generate and repair schedules, track state, and monitor resources against predicted evolution.

SCL continues to model spacecraft activities and state at finer levels of detail. These activities are modeled as SCL scripts, which when chained together and executed, result in commands to the EO-1 FSW. SCL models spacecraft state through an internal database where each record stores the current value of a sensor, resource, or sub-system mode. The SCL model also includes flight rules that monitor spacecraft state, and execute appropriate scripts in response to transient changes. SCL uses its model to generate and execute sequences that are valid and safe in the current context. But while SCL has a detailed model of spacecraft state and resources, it does not generally model future evolution of state or resources.

The ASE team developed the CASPER and SCL models using an iterative multiple step process, that defined, modeled, reviewed, and validated EO-1 activities. Each of these steps focused on creating a high-fidelity model that was consistent with existing ground operations and constraints of the EO-1 spacecraft.

3.1 Model Development

The model development process began when a new high-level goal was tasked to ASE. At first ASE modeled simple goals, such as instrument calibrations. As we gained experience with the spacecraft, our modeling activities evolved to more complex multi-activity objectives including science observations, data downlinks, and spacecraft pointing.

With a new goal in hand, the ASE team would first identify the set of activities required to achieve the objective. Primarily this process was driven by a review of existing operations documents and engineering reports. For example, when ASE was tasked to begin collecting science data, prior EO-1 data collects were analyzed to see what sequences of commands, and thus activities, were required to image a science target. A science data collect requires activities to calibrate instruments, manage hardware operational modes, and command data recording from both the Hyperion and Advanced Land Imager (ALI) instruments.

With the activities defined, the ASE team reviewed formal EO-1 operations procedures to identify constraints on the selected activities. For example, due to thermal constraints, the Hyperion cannot be left on longer than 19 minutes, and the ALI no longer than 60 minutes. The EO-1 operations team also provided spreadsheets that specified timing constraints between activities. Downlink activities, for example, are often specified with start times relative to the ground station acquisition of signal (AOS) and loss of signal (LOS). Finally, fault protection documents listing fault monitors (TSMs) were consulted, reasoning that acceptable operations should not trigger any TSMs.

3.2 Model Reviews

Next, the ASE team conducted reviews where the latest iterations of the CASPER and SCL models were tabletop reviewed by a team composed of EO-1 spacecraft engineers and operators. Their working knowledge of the spacecraft, and experience over three years of operations, verified that no incorrect parameters or assumptions were represented in the model.

Finally, a spacecraft safety review process was performed. In this process, experts from each of the spacecraft subsystem areas (e.g. guidance, navigation and control,

solid state recorder, Hyperion instrument, power, …) studied the description of the ASE software, including the commands that the ASE software could execute, and derived a list of potential hazards ASE could pose to the spacecraft's health. For each of these hazards, a set of possible safeguards were proposed, and then implemented through operations procedures, and constraints embedded in the CASPER and SCL models. This analysis formed the basis for the testing of agent safety discussed in section 4. A sample analysis for two risks is shown below.

Table 1. Sample safety analysis for two risks.

	Instruments overheat from being left on too long	**Instruments exposed to sun**
Operations	For each turn on command, look for the following turn off command. Verify that they are within the maximum separation.	Verify orientation of spacecraft during periods when instrument covers are open.
CASPER	High-level activity decomposes into turn on and turn off activities that are with the maximum separation.	Maneuvers must be planned at times when the covers are closed (otherwise, instruments are pointing at the earth)
SCL	Rules monitor the "on" time and issue a turn off command if left on too long.	Constraints prevent maneuver scripts from executing if covers are open.
FSW	Fault protection software will shut down the instrument if left on too long.	Fault protection will safe the spacecraft if covers are open and pointing near the sun.

3.3 Code Generation

An interesting aspect of model development was the use of code generation techniques to derive SCL constraint checks from CASPER model constraints. In this approach, certain types of CASPER modeling constraints could be translated into SCL code to ensure consistency at execution time. If the CASPER model specifies that activities use resources, this can be translated into an SCL check for resource availability before the activity is executed. If the CASPER model specifies a state requirement for an activity, a check could be auto-generated to verify a valid state before executing the activity. Additionally, if the CASPER model specifies sequential execution of a set of activities, code can be generated so that SCL enforces this sequential execution.

For example, in calibrating the Hyperion instrument, the solid state recorder (WARP) must be in record mode and the Hyperion instrument cover must be open. Below we show the CASPER model and the generated SCL constraint checks.

```
// Hyperion calibration
activity hsi_img_cal
{
  durat caldur;
  // schedule only when the WARP is in record
  // mode, recording data, and
  // when the hyperion cover is open
  reservations =
    wrmwmode must_be "rec",
    ycovrstat must_be "closed";
  // start and stop the instrument
  decompositions =
    yscistart, yscistop
    where yscistop starts_after
          start of yscistart by caldur;
}

-- Hyperion calibration
script hsi_img_cal caldur
  -- verify that the WARP is in record
  -- mode, recording data, and
  -- that the hyperion cover is open
  verify wrmwmode = rec
     and ycovrstat = closed
         within 5 seconds
  -- start and stop the instrument
  execute yscistart
  wait caldur sec
  execute yscistop
end hsi_img_cal
```

Figure 2. Sample model and script for Hyperion calibration.

Note that this generated code also enforces the sequential execution of the "yscistart" and "yscistop" activities, separated by "caldur" seconds. This shows how code is automatically generated from a CASPER defined temporal constraint over two activities.

As another example, when initiating the WARP recording, there is a limit on the total number of files on the WARP recorder (63). In CASPER we defined the constraint that "wfl" new files are created. We then auto-generated SCL code to verify that number of files can be created without exceeding the file limit before the WARP recording activity is allowed to execute.

```
// Start the WARP recording
activity wrmsrec
{
  ...
    reservations =
      // reserve the required number of
      // files on the WARP
      wrmtotfl use wfl,
      // change the warp to record mode when
      // complete
      wrmwmode change_to "rec" at_end,
  ...
}

-- Start the WARP recording
script wrmsrec
  ...
       verify
        wrmfreebl wrmtotfl + wfl <= 63
        and wrmtotfl + wfl >= 1 and
  ...
end wrmsrec
```

Figure 3. Sample model and script for WARP recording.

3.4 Sequence Generation

With the model defined, CASPER generated preliminary command sequences from past science requests representing a range of potential flight situations. These sequences were compared with the actual sequences generated and uplinked by the EO-1 ground team for the same request. Significant differences between the two sequences identified potential problems with the model. For example, if two commands were sequenced in a different order, this potentially revealed an overlooked constraint on one or both of the commands. The EO-1 team also provided engineering telemetry from the onboard execution of these sequences. This telemetry allowed for execution comparisons to the telemetry generated by ASE. Additionally a novel "played back" capability was developed where the ASE software could be fed the results of commands using the actual effects observed onboard. The command sequences were aligned with the telemetry to identify the changes in spacecraft state and the exact timing of these changes. Again, any differences between the actual telemetry and the ASE telemetry revealed potential errors in the model. We converged on a consistent model after several iterations through this sequence generation process.

The sequence generation effort was in effect the crossover point between our model development process and the beginning of our system-level testing. While feeding directly into the iterative development process, it also allowed the first validation of the ASE model and software.

4 Testing Enforcement of Safety

Testing ASE against prior sequences would not be enough. We needed to show that onboard EO-1 the system would correctly plan, generate, and execute command sequences. Or, more importantly, that the generated command sequences would never endanger the safety of the spacecraft.

As demonstration software, the effort available for testing our agent was severely time and resource constrained. Therefore we decided early in the project that testing should focus primarily on ensuring that our agent executed safely. Missing a data collect would be an unfortunate although tolerable failure - endangering the safety of the EO-1 spacecraft would not.

Leveraging the completed safety analysis, we approached validation by breaking our testing strategy into three verification steps:

1. CASPER generates plans consistent both with its internal model of the spacecraft and SCL's model and constraints.
2. SCL does not issue any commands that violate the constraints of the spacecraft.
3. Both models accurately encode the spacecraft operational and safety constraints.

The first two steps build confidence that the ASE software executes within the constraints levied by the spacecraft model, while the third step verifies that the model encodes sufficient information to protect against potential safety violations.

We validated these requirements by extensive testing of the autonomy software on generated test-cases, using simulation and rule-based verification at each step. Note that the steps enumerated above, and the test cases described below, address only the top-two layers of the onboard autonomy software (CASPER and SCL). The existing EO-1 flight software testing and validation was addressed prior to ASE by a separate, more conventional, test plan. Additionally both CASPER and SCL are mature and tested software systems. The majority of the development effort for ASE was in the two internal models that adapt the systems to EO-1. Accordingly the testing strategy outlined below focuses the majority of the effort on exercising those models.

4.1 Test Case Parameters

Each EO-1 test case spans seven days of spacecraft operations covering multiple science observation and reaction opportunities. Each observation opportunity, referred to as a CASPER schedule window, represents an time period where ASE has been cleared to command EO-1. The test cases vary the state observed by ASE entering schedule windows (spacecraft state parameters), and vary the goals given to ASE through changes to mission and science objectives (mission scenario parameters). Additionally we employed simulators that changed the spacecraft state during test execution to simulate unknown environmental changes.

Mission scenario parameters represent the high-level planning goals passed to ASE. They are derived from a combination of the orbit and long-term science objectives. Mission scenario parameters specify when targets will be available for imaging, the parameters of science observations (i.e. number of targets to image and science analysis algorithms we wish to execute), and reactions to observed science events (i.e. follow-up observations).

Table 3. Mission-scenario parameters.

Parameter	Nominal	Off-nominal	Extreme
schedule windows	0-3	3-5	5+
orbits between windows	2-7	1,8	0,8+
window start time	start of orbit	+/- 10 min	any
window duration	expected time of science analysis	+/- 10 min	any
image start	anytime in orbit, 1 per orbit	1 per 3 orbits	any
image duration	8 s +/- 2	+/- 5	0,60
groundstation AOS	anytime in orbit, 1 per orbit	1 per 3 orbits	any
groundstation LOS	AOS + 10 min +/- 1	+/- 3	any
eclipse start	60 min after orbit start	+/- 5	any
eclipse duration	30 min	+/- 5	any
science algorithm	any	any	any
science goal start	fixed	not-specified	any
number of science goals	1 per orbit	1-2	>2
warp allocated	0	32K blocks	any

Spacecraft state parameters encode the relevant state of EO-1 at the start of a schedule window, and change as a result of commanded sequences. Changes to these parameters are simulated using a software simulator.

Table 2. Sample spacecraft state parameters.

Parameter	Expected Initial State
xband groundstation	unknown
xband controller	enabled
ACS mode	nadir
target selected	unknown
warp electronics mode	stndops
warp mode	standby
warp bytes allocated	0
warp number files	0
fault protection	enabled
eclipse state	full sun
target view	unknown
hyperion instrument power	on
hyperion imaging mode	idle
hyperion cover state	closed
ali instrument power	on
ali active mechanism	telapercvr
ali mechanism power	disabled
ali fpe power	disabled
ale fpe data gate	disabled
ali cover state	closed
groundstation view	Unknown
mission lock	unlocked

To exhaustively test every possible combination of state and observation parameters, even just assuming a nominal and failure case for each parameter and ignoring execution variations, would require 2^{36} or over 68 billion test cases (each requiring on average a few hours to run). The challenge therefore becomes selecting a set of tests that most effectively cover the space of possible parameter variations within a timeframe that allows for reasonable software delivery.

4.2 Design of Test Cases

Traditional flight software is designed to be tested through exhaustive execution of a known set of command sequences. Command sequences usually must be run through a high-fidelity ground testbed before being cleared to run onboard.

Autonomy software however enables the spacecraft to execute in, and react to, a much wider range of possible scenarios. This flexibility enables new paradigms of operations and science, but comes at the price of complexity in testing and validation – tests that must attempt to intelligently cover the range of possible states and mission scenarios.

To trim the set of possible inputs, we took advantage of the scenarios identified by the model review process. For example, we never expect to take more than five science

data collects before a downlink (and usually exactly five as that is the limit of the WARP data storage). A downlink is almost always followed immediately by a format of the WARP. Science collections are always preceded by a slew and wheel bias and followed by a slew to nadir. Together these form a baseline mission scenario covering all the actions to be commanded by our agent.

Instead of testing every possible combination of spacecraft and mission parameters, we instead decided to vary parameters off of this baseline scenario, thus reducing the number of parameter variations for our test cases to consider. This is a similar approach to that used to validate the Remote Agent Planner for NASA's Deep Space 1 mission. [11].

We started the design process by using the nominal parameter values identified in the model review process. Using these assignments we generated test cases that varied each of the parameters across three distinct classes of values – nominal (single value), off-nominal (range of acceptable values), and extreme (failure conditions). For each parameter, we defined a set of five values at the boundaries of these classes – a minimum value, an "off-nominal-min" value at the boundary between the off-nominal and the extreme, a nominal value, an "off-nominal-max", and a maximum value.

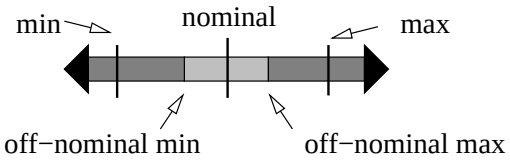


Figure 4. Parameter Decompositions

Using this decomposition we generated three sets of test cases:

1. Baseline scenario test cases that exercised just the baseline mission scenario.
2. Stochastic test cases, grounded in the baseline mission scenario, that varied parameters within nominal, off-nominal, and extreme ranges.
3. Environmental test cases that varied initial state, and inserted execution uncertainty.

4.2.1 Baseline-Scenario Test Set

The baseline mission scenario, identified in the model review process, was used for the first and most basic test set validating ASE.

This scenario provided exactly the expected sequences and parameter values to the ASE software. Any inconsistencies or anomalies in execution were easily traced back as the scenario was well understood and used previously to generate command sequences during the model review process.

4.2.2 Stochastic Test Set

Clearly the baseline test set did not fully exercise the autonomous planning and reaction capabilities of the system. In order to test more nominal scenarios, and also gain coverage in the off-nominal parameter ranges, we devised a procedure for generating stochastic test sets based on parameter value distributions.

Parameters were given normal distributions around their nominal value, with standard deviations half the width of the off-nominal range (such that 95% of expected values will be either nominal or off-nominal). Nominal test sets were then generated assigning values to parameters based on the defined distributions. Furthermore, by modifying the construction of the parameter distribution, we were able to create off-nominal and extreme test sets that would stochastically favor some parameters to choose values outside of their nominal range.

4.2.3 Environmental Test Set

We further extended the stochastic test sets described above to include execution variations based on the parameter distributions. The spacecraft simulator was modified to allow as input variations to expected parameter values. During the execution of activities the simulator simulated changes to each parameter of the current activity, and then varied the value returned based on the provided parameter distributions. Again nominal, off-nominal, and extreme test sets were generated that instructed the simulator to vary parameter values within the corresponding value class.

Finally we needed a way to test how the system responded to unexpected or exogenous events within the environment. These events could be fault conditions in the spacecraft or events outside of the CASPER model. Unlike the initial-state and execution-based testing described above, these events could happen at any time, and do not necessarily correspond to any commanded action or modeled spacecraft event. To accomplish this we added to our spacecraft simulator the ability to change the value of any parameter, at either an absolute time or time relative to the execution of an activity, to a fixed value or a value based on the distributions described above. We added small-variation events (within appropriate off-nominal and nominal classes) to our nominal and off-nominal stochastic test sets.

4.3 Testing Procedure

The test cases generated using the procedure outlined above were used in unit testing the individual agent layers and integrated system testing. Unit testing verified primarily the first two decompositions of our test plan – that CASPER commanded within its model, and that SCL did not violate any spacecraft constraints. Integrated testing verified that these constraints hold within the full system, and that the commanded sequences safely achieve the mission objectives.

The vast majority of tests were run on the Solaris and Linux platforms - as they were the fastest and most readily available. However, these test the software under a different operating system and processor, and therefore are primarily useful for testing assumptions in the CASPER and SCL models. The operating system and timing differences are significant enough that many code behaviors occur only in the target operating system, compiler, and processor configuration. Therefore every effort was made to extensively validate the agent on higher fidelity testbeds.

Table 4. Testbeds available to validate EO-1 agent.

Type	Number	Fidelity
Solaris Sparc Ultra	5	Low – can test model but not timing
Linux 2.5 GHz	7	″
GESPAC PowerPC 100-450 MHz	10	Moderate – runs flight OS
JPL Flight Testbed RAD 3000	1	Moderate
EO-1 Flight Testbed Mongoose M5, 12 MHz	1	High – runs Flight Software
EO-1 Autonomy Testbed Mongoose M5, 12 MHz	2	High – runs Flight Software

On the Linux, Solaris, and GESPAC testbeds we used an automated test harness to setup, execute, and evaluate the results of each test run. Tests were run at accelerated speeds using the capabilities of our software simulator and the resources of the faster processors. The GESPAC and flight testbed configurations do not have similar acceleration capabilities, and therefore require tests to be run in real-time. The test harness ran over six years of autonomous operations during the first six months of our validation process.

To ensure stability, we implemented minimum requirements on the number of test cases that must execute without an identified failure before a build was cleared for flight. These requirements varied by platform as follows: 1 year of simulated operations on Linux/Solaris, 1 month on the GESPAC single board computers, and 1 week on the flight testbeds.

4.4 Success Criteria

To be considered successful a test run could not violate any spacecraft, operations, or safety constraints. On the Linux, Solaris, and GESPAC testbeds these constraints were checked by a software simulator that monitored activities committed by CASPER and executed by SCL. The simulator verified the timing, state, and resource constraints of the activities against those encoded in the CASPER model.

Recalling that our primary testing objective was to verify that our agent commanded EO-1 safely, we developed a separate "safety monitor" that watched only for violations of the safety and operations constraints. The safety monitor was developed with no knowledge of the CASPER or SCL models, and parsed the actual spacecraft commands issued by the autonomy software (isolated black-box testing). These commands were fed into state machines that monitored each of the safety and operations constraints – the same constraints that were derived from the safety and model review process. Any violations that were discovered were considered high-priority defects.

The flight testbeds used a higher-fidelity "Virtual Satellite (VSat)" simulator, developed independently from the autonomy software. The VSat simulator modeled the spacecraft at the subsystem level, including systems, states, and resources not modeled by CASPER or SCL.

5 Status & Deployment

The full ASE software has successfully commanded science observations onboard EO-1 since January 2004, As of April 2004, ASE has successfully collected target observations, analyzed science data onboard EO-1, and autonomously retargeted the spacecraft for subsequent observations. The ASE software has been so successful that it is currently being used to fly EO-1 in normal operations and is expected to be used as such until the end of mission (at least Fall 2005).

Test Description	Test Date
First test of onboard cloud detection (science analysis)	March 2003
Verification of ASE-EO-1 FSW commanding path	May 2003
Onboard execution of CASPER ground-generated command sequences	July 2003
Full ASE software upload	August 2003
First ASE autonomously-commanded dark calibration image and downlink	October 2003
First ASE autonomous science observation	January 2004
First autonomous science analysis and subsequent reaction observation.	April 2004
Expanded EO-1 science operations automation.	May 2004-Present

6 Conclusions

This paper described the design and validation of a safe agent for autonomous space science operations. First, we

described the challenges in developing a robust, safe, spacecraft control agent. Second, we described how we used a layered architecture to enhance redundant checks for agent safety. Third, we described our model development, validation, and review. Finally, we described our test plan, with an emphasis on verifying agent safety.

7 Acknowledgement

Portions of this work were performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

We would like to acknowledge the important contributions of Dan Mandl, Stuart Frye, and Stephen Ungar of NASA's Goddard Spaceflight Center, Jerry Hengemihle and Bruce Trout of Microtel LLC, Jeff D'Agostino of the Hammers Corp., Seth Shulman and Robert Bote of Honeywell Corp., and Jim Van Gaasbeck and Darrell Boyer of Interface and Control Systems.

8 References

[1] P. Bonasso, J. Firby, E. Gat, D. Kortenkamp, D. Miller, M. Slack, Experiences with an Architecture for Intelligent, Reactive Agents, Journal of Experimental and Theoretical Artificial Intelligence, 9:237-256, 1997.

[2] S. Chien, R. Sherwood, M. Burl, R. Knight, G. Rabideau, B. Engelhardt, A. Davies, P. Zetocha, R. Wainright, P. Klupar, P. Cappelaere, D. Surka, B. Williams, R. Greeley, V. Baker, J. Doan, "The TechSat 21 Autonomous Sciencecraft Constellation", *Proc i-SAIRAS 2001*, Montreal, Canada, June 2001.

[3] S. Chien, R. Knight, A. Stechert, R. Sherwood, and G. Rabideau, "Using Iterative Repair to Improve Responsiveness of Planning and Scheduling," *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*, Breckenridge, CO, April 2000. (see also casper.jpl.nasa.gov)

[4] Chien, S., R. Sherwood, D. Tran, B. Cichy, G. Rabideau, R. Castano, A. Davies, R. Lee, D. Mandl, S. Frye, B. Trout, J. Hengemihle, J. D'Agostino, S. Shulman, S. Ungar, T. Brakke, D. Boyer, J. VanGaasbeck, R. Greeley, T. Doggett, V. Baker, J. Dohm, F. Ip, The EO-1 Autonomous Science Agent, Proc. Of the 2004 Conference on Autonomous Agents and Multi-agent Systems, NY, NY, July 2004.

[5] S. Chien et al, EO 1 Autonomous Sciencecraft Experiment Safety Analysis Document, 2003.

[6] D. Cohen; Dalal, S.; Fredman, M.; and Patton, G.1997. The AETG system: An approach to testing based on combinatorial design. IEEE Transactions on Software Engineering 23(7):437-444.

[7] A.G. Davies, R. Greeley, K. Williams, V. Baker, J. Dohm, M. Burl, E. Mjolsness, R. Castano, T. Stough, J. Roden, S. Chien, R. Sherwood, "ASC Science Report," August 2001. (downloadable from ase.jpl.nasa.gov)

[8] E. Gat, Three layer architectures, in Mobile Robots and Artificial Intelligence, (Kortenkamp, Bonasso, and Murphy eds.), Menlo Park, CA: AAAI Press, pp. 195-210.

[9] Goddard Space Flight Center, EO-1 Mission page: eo1.gsfc.nasa.gov

[10] Interface and Control Systems, SCL Home Page, sclrules.com

[11] NASA Ames, http://ic.arc.nasa.gov/projects/remote-agent/, Remote Agent Experiment Home Page.

[12] G. Rabideau, R. Knight, S. Chien, A. Fukunaga, A. Govindjee, "Iterative Repair Planning for Spacecraft Operations in the ASPEN System," *International Symposium on Artificial Intelligence Robotics and Automation in Space*, Noordwijk, The Netherlands, June 1999.

Finding Optimal Plans for Domains with Restricted Continuous Effects with UPPAAL CORA

Henning Dierks
University of Oldenburg
Germany

Abstract

We present a translation of a variant of PDDL with restricted continuous effects into linearly priced timed automata. For the latter notion the model-checker UPPAAL CORA is able to find cost-optimal traces. We explain the PDDL variant and its translation into the syntax of UPPAAL CORA. A case study is used to explain the approach.

Introduction

An interesting trend in recent years in computer science research is the growing exchange of techniques in two areas which were rather disjoint previously. On the one side the planning community dealt with the problem to find valid plans in domains automatically. A main obstacle is to find informative and efficient heuristics to guide the search towards the goal. Usually the aspects of quantitative time and optimality of plans were neglected because finding just a valid plan in acceptable time was difficult enough in domains with huge state spaces.

On the other side the verification community worked hard to cope with the problem of state explosion when model-checking is applied. The standard technique is to find efficient data structures for symbolic representation of set of states. In case of discrete time this is rather successful and even for models with continuous time excellent tools are available, for example UPPAAL(Larsen, Petterson, & Wang Yi 1997; Behrmann, David, & Larsen 2004).

The common problem of both research areas are huge state spaces. Planning usually means to find a way through this space. Verification usually means to prove the absence of such a way. On the first sight it seems that the problems are contrary but there are situations where model-checking can benefit from heuristics. For example, when the model-checker

- tests a system that is known to be faulty or
- should check whether a system is able to execute a given abstract trace. This is rather often a problem when large systems are abstracted due to limited resources. When the model-checker finds an abstract trace the question is still open whether the abstraction was too coarse or the trace is feasible in the full model.

In both cases it make sense to guide the search of the model-checker. Examples of such approaches are (Kupferschmid *et al.* 2005; Dierks 2005).

However, the planning community can benefit from the achievements of the verification community as soon as quantitative time and the question for optimality comes into the play. This is topic of this paper. We introduce a variant of the standard specification language for planning problems PDDL. It is an extension of PDDL 2.1 at level 3 towards duration-dependent and continuous effects. The latter effects, however, are restricted to the costs. That means it is allowed to specify the costs of a durative action depending of the duration of the action. To solve the problem of finding cost-optimal plans in such domains we translate the planning problem to linearly priced timed automata for which the tool UPPAAL CORA exists. It is able to find cost-optimal traces which represent valid plans.

The paper is organised as follows. You are reading the introduction. The next section introduces briefly our variant of PDDL. Thereafter we explain (priced) timed automata and the capabilities of the model-checker UPPAAL. The translation from PDDL to timed automata is explained in the following section. We end with a case study and conclusions.

PDDL

PDDL (Planning Domain Definition Language) has been introduced in (McDermott & the AIPS-98 Planning Competition Committee 1998) as common problem-specification language for the AIPS-98 planning competition. The purpose of PDDL is to describe the nature of a domain by specifying its entities and actions that may have effects on the domain. These effects can change the state of the domain.

In order to handle domains with time and numbers PDDL has been extended hierarchically (Fox & Long 2001b; 2001a): The original PDDL is called PDDL level 1; the first extension by numeric effects represents level 2. That means in PDDL at level 2 it is possible to handle functional expressions and effects may change the values of those. The next extension (level 3) allows to specify *durations* for actions. Further extensions introduce duration-dependent effects for durative actions (level 4) and continuous effects for durative actions (level 5).

In this paper we will deal with PDDL at level 3 together with restricted continuous effects for durative actions (called PDDL$_{cora}$) for which we offer an *automatic* translation from PDDL$_{cora}$ domain specifications into networks of timed automata suited for the real-time verification tool UPPAAL CORA. Thus, the entire collection of data structures and heuristic search-algorithms developed within the framework of UPPAAL become available to any planning problem describable within PDDL$_{cora}$. An interesting aspect in which our approach differs from (Dierks, Behrmann, & Larsen 2002) is the fact that we leave the decidable set of timed automata due to our extensions. The reason is that by duration-dependent effects we can sum up durations. Thus, we are able to build "stopwatches" and it is known that timed automata with stopwatches are more expressive than timed automata (Cassez & Larsen 2000). Moreover, reachability is undecidable for stopwatch automata.

In Fig. 1 an example of a domain description in PDDL$_{cora}$ is given. It describes a variation of a planning problem in which landing planes have to be scheduled to runways. Each plane has three landing times given:

- `earliest` denotes the earliest time where the aircraft can land.
- `target` describes the desired time. When the aircraft misses this time it is late.
- `latest` defines the latest time at which the plane must land.

Moreover each aircraft is assigned two rates. The optimal time point for the plane is the target time. In this case it produces no costs at all. Landing earlier than the target time increases the costs by the `early-rate`. Landing after the target time produces immediate costs, a so-called `late-penalty`. The costs for landing late increase with the `late-rate`.

In PDDL$_{cora}$ we model this problem as follows. We introduce two types `plane` and `runway`. Each plane should be `landed` in the goal state and landing requires that a runway is not `occupied` and the plane is `scheduled` to this runway. The procedure to land an aircraft requires to schedule the aircraft to a runway first. This is done by the `schedule` action and it action takes 40 time units. After landing the aircraft has to leave the runway which takes 40 time units also.

The landing is modelled by two actions. The first durative action is `land-on-time` which covers the case that plane lands in-between the earliest and target time. It starts at the earliest landing time and finishes at the target at latest. This is achieved by specifying

```
(at start (= total-time (earliest ?p)))
```

and

```
(:duration
   (<= ?duration (- (target ?p)
                    (earliest ?p))))
```

The costs depend on the time when this action *ends*. We compute them by increasing the costs at the beginning and decrease them by the early rate during execution.

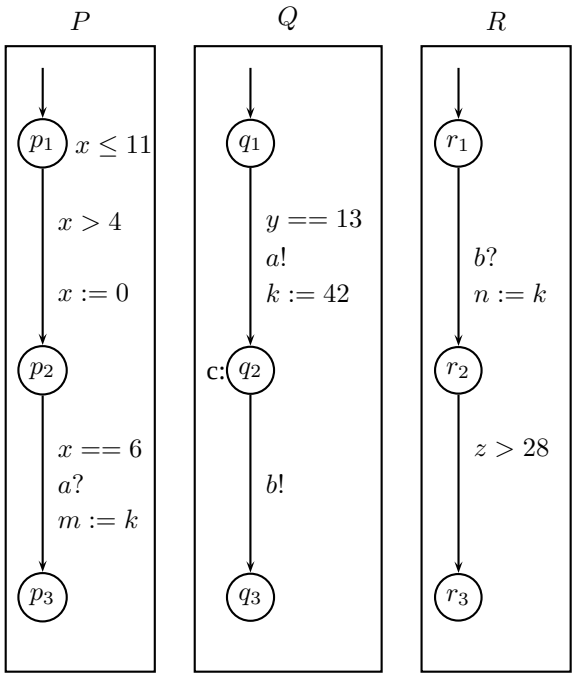


Figure 4: A simple UPPAAL model.

The second action models a delayed landing of the aircraft. In this case the action starts at the target time and takes at most the difference between latest and target time. The penalty for the delayed landing is paid at the beginning of the action and during the action the costs increase with the late-rate.

The problem specification contains concrete objects, initial values for the functions and predicates of the domain and the goal specification. In Fig. 2 a problem instance for the Landing domain is given. In contains a problem with three aircrafts and a single runway. The goal is to land all aircrafts with minimal costs.

A valid plan for this is sketched in Fig. 3. The costs of this plan consist only of the costs for the landing of aircraft KL101 because the other ones land exactly at target time. Since KL101 lands at 178 we get

$$500 + (178 - 155) \cdot 10 = 530$$

as costs for this example plan.

UPPAAL CORA

UPPAAL[1] is a modeling, simulation, and verification tool for real-time systems modeled as networks of timed automata (Alur & Dill 1990; 1994) extended with data types such as bounded integer variables, arrays etc. In this subsection we describe briefly how networks of timed automata are specified in the syntax of UPPAAL. For more details about this tool the reader is confered to (Larsen, Petterson, & Wang Yi 1997; Behrmann, David, & Larsen 2004).

[1] See the web site `http://www.uppaal.com/`.

```
(define (domain Planes)
  (:requirements :typing :fluents :negative-preconditions
   :conditional-effects :equality :duration-inequalities
   :durative-actions)
  (:types plane runway)
  (:predicates  (occupied ?r - runway)
                (landed   ?p - plane)
                (scheduled ?p - plane ?r - runway))
  (:functions   (earliest     ?p - plane)
                (target       ?p - plane)
                (latest       ?p - plane)
                (early-rate   ?p - plane)
                (late-rate    ?p - plane)
                (late-penalty ?p - plane)
                (cost))

(:durative-action schedule      :parameters (?p - plane ?r - runway)
 :duration (= ?duration 40)
 :condition (at start (and (not (occupied ?r))
                           (not (landed ?p))
                           (not (scheduled ?p ?r))))
 :effect (and (at start (occupied ?r))
              (at end (scheduled ?p ?r))))

(:durative-action clear         :parameters (?p - plane ?r - runway)
 :duration (= ?duration 40)
 :condition (and (at start (occupied ?r))
                 (at end   (occupied ?r))
                 (at start (scheduled ?p ?r))
                 (at start (landed ?p)))
 :effect (and (at end   (not (occupied ?r)))
              (at start (not (scheduled ?p ?r)))))

(:durative-action land-on-time  :parameters (?p - plane ?r - runway)
 :duration (<= ?duration (- (target ?p) (earliest ?p)))
 :condition (and        (at end   (occupied ?r))
                        (at end   (scheduled ?p ?r))
                        (at start (= total-time (earliest ?p))))
 :effect (and (at end (landed ?p))
              (at start (increase cost  (* (early-rate ?p)
                                           (- (target ?p) (earliest ?p)))))
              (decrease cost (early-rate ?p))))

(:durative-action land-late     :parameters (?p - plane ?r - runway)
 :duration (<= ?duration (- (latest ?p) (earliest ?p)))
 :condition (and        (at end   (occupied ?r))
                        (at end   (scheduled ?p ?r))
                        (at start (= total-time (target ?p))))
 :effect (and (at end (landed ?p))
              (at start (increase cost (late-penalty ?p)))
              (increase cost (late-rate ?p))))
)
```

Figure 1: The planes domain.

```
(define (problem plane1)
  (:domain Planes)
  (:objects     KL101,KL108,KL115 - plane
                r - runway)

  (:init (not (occupied r))
         (not (landed   KL101)) (not (scheduled KL101 r))
         (not (landed   KL108)) (not (scheduled KL108 r))
         (not (landed   KL115)) (not (scheduled KL115 r))

        (= (earliest KL101) 129) (= (early-rate   KL101) 10)
        (= (target   KL101) 155) (= (late-rate    KL101) 10)
        (= (latest   KL101) 559) (= (late-penalty KL101) 500)

        (= (earliest KL108) 195) (= (early-rate   KL108) 10)
        (= (target   KL108) 258) (= (late-rate    KL108) 10)
        (= (latest   KL108) 744) (= (late-penalty KL108) 500)

        (= (earliest KL115) 89)  (= (early-rate   KL115) 30)
        (= (target   KL115) 98)  (= (late-rate    KL115) 30)
        (= (latest   KL115) 510) (= (late-penalty KL115) 500)
  )

  (:goal (and   (landed KL101)
                (landed KL108)
                (landed KL115)
                (not (occupied r))))
  (:metric minimize cost)
)
```

Figure 2: The planes domain.

Figure 4 contains a simple system of three automata – called processes in UPPAAL – P, Q and R which work in parallel. The initial states p_1, q_1 and r_1 are marked by ingoing edges with no source. Initial all three clocks (x, y, z) are set to 0 and no transition is enabled: The transition from p_1 to p_2 is blocked by the guard $x > 4$. The transition from q_1 to q_2 is also blocked by the guard on clock y. Similarly, the transition from r_1 to r_2 is initially blocked because it requires a synchronisation on a channel b. Hence, the only legal behaviour of this system in the initial state is to let time pass. All clocks are incremented linearly with rate 1.

As soon as more than 4 time units have passed, the transition to p_2 can be fired. If this happens the clock x will be reset to 0. Note that the system is not forced to execute this transition immediately after it is enabled. It may even wait forever unless an invariant at the source state is given. In this case p_1 is equipped with the invariant $x \leq 11$ and therefore the process P is allowed to select a time point in $]4, 11]$ to fire the transition.

In state p_2 process P can fire the transition to p_3 only together with another transition since it requires a synchronisation on channel a. The synchronisation on a channel c is possible if there are two processes with transitions labelled with $c!$ resp. $c?$ and both guard are satisfied. Then both transitions are executed together whereas the actions of the $c!$ transition are executed before those of the $c?$ transition. Nevertheless the results of the actions are computed without any delay. In the example of Fig. 4 the system may switch from p_2 and q_1 to the states p_3 and q_2 simultaneously provided that $y == 13$ and $x == 6$ hold. When executed the variables k and m are *both* set to 42. It is easy to see that this can only happen when P has taken the transition to p_2 at the time point 7.

If Q has reached q_2 it has to leave this state at the same point in time because q_2 is marked as *committed* state. That requires the process to leave this state before time passes again. Although this behaviour can be expressed by setting the clock y to 0 and adding the invariant $y \leq 0$, this feature is useful since in can save clocks which is the most expensive entity when model-checking timed automata. Hence, Q is forced to synchronise on channel b and process R can serve as synchronisation partner. Hence at time point 13 but *after* the transition to q_2 it fires the transition to q_3 together with R which enters r_2. The action sets n to 42, too. Finally, R may change from state r_2 to r_3 provided that $z > 28$ holds.

This example highlights only some features of the extended timed automata which can specified in UPPAAL. Further additions are extensions of the data types (enumerations, bounded integers and Boolean variables and arrays of those). In the latest versions also broadcasting channels are allowed. If a broadcast $c!$ on channel c is executed all processes which have a enabled, $c?$-labelled transition will take part.

In Fig. 5 the model of Fig. 4 is given in the ASCII syntax of UPPAAL. Queries are given in a different file and the tool

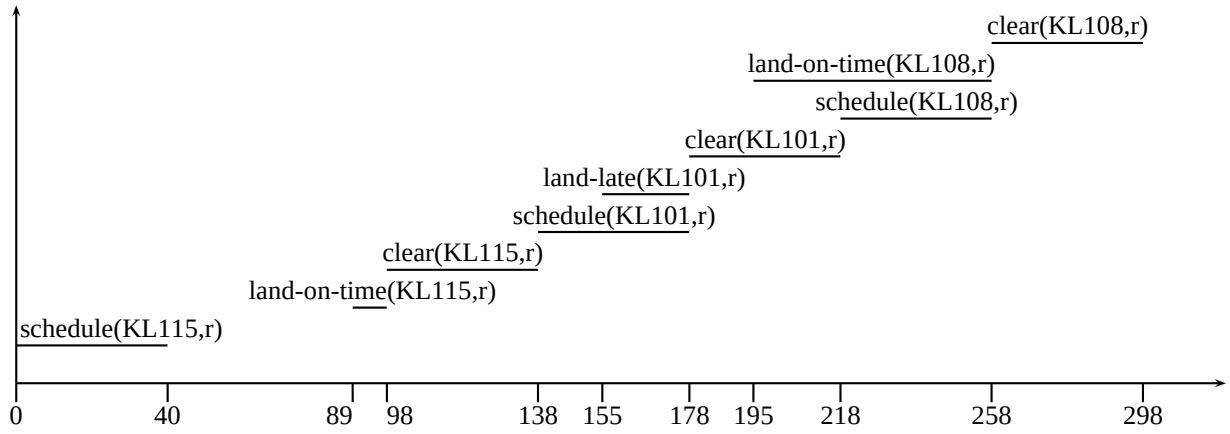


Figure 3: A valid plan for the problem of Fig. 2

is basically able to check for reachability properties.[2] An example suitable for Fig. 4 is:

```
E<> (P.p3 and Q.q3 and R.r3)
A[] (m==k)
```

Both queries are satisfied. Note that the last query states that m and k are always equal. Hence, the assignments of the a-synchronised transitions is executed atomically.

In more recent work (Behrmann *et al.* 2001; Larsen *et al.* 2001) the timed automata model as well as the underlying verification engine of UPPAAL have been extended to support computation of *optimal* reachability with respect to various cost criteria. The name of this tool variant is called UPPAAL CORA. In (Behrmann *et al.* 2001) the timed automata model has been extended with discrete costs on edges and the optimality criteria consist in minimising either the total accumulated time (for reaching a goal state) or the total accumulated discrete cost or the sum of these two. UPPAAL CORA (Behrmann *et al.* 2001; Larsen *et al.* 2001) offer various mechanisms for guiding and pruning the search for optimal reachability and has been applied successfully on a number of scheduling problems (e.g. job-shop scheduling, air-craft landing).

An example of a priced timed automaton is given in Fig. 6. It models a typical scheduling problem of a traveller from Paris to London. There are two choices for her: Flying directly with flight Fl.1 or via Amsterdam using Fl.2 and Fl.3. Depending on the optimality criterion the best choice is:

- Flying directly to optimise time consumption only. It takes 20 time units to fly directly and 30 time units via Amsterdam.
- Flying via Amsterdam optimises the costs. It costs 5 cost units via Amsterdam and 7 cost units to fly directly.
- If time is costly, i.e. one time unit costs one cost unit, the best choice is to fly directly (27 cost units vs. 35 cost units).

The example demonstrates how costs can be added as assignments for the discrete transitions. It is also possible to set the derivative of the variable $cost$ in order to specify whether time is costly or not. It is clear that UPPAAL CORA can prune the search space as soon as a trace to goal has been found. If the cost of a trace to the goal is c then all other traces for which the costs are known to be at least c are irrelevant. This pruning can be improved by adding information about the remaining costs into the model. To this end a special variable $remaining$ can be used that is expected to *underestimate* the remaining costs to reach the goal.

Consider again the example above in the setting where time costs. One could add the assignment $remaining := 12$ to the transition from state Fl.2 to state Amsterdam because it might be known that a flight from Amsterdam to London costs at least 12 time units. With this additional information the UPPAAL CORA would know that the costs via Amsterdam are *at least* $2 + 15 + 12 = 29$ as soon as it fires the transition from state Fl.2 to state Amsterdam. If it has found the direct flight with the total costs of 27 cost units earlier it could prune the search for cheaper flights via Amsterdam.

Besides the special variables $cost$ and $remaining$ another variable called $heur$ can be used to guide UPPAAL CORA finding the optimal solution. Internally the model-checker maintains a list of reachable states which are waiting for exploration ("waiting list"). It is possible to start UPPAAL CORA with a flag such that the waiting list is sorted by the corresponding values of $heur$. The effect is that the state which has the smallest $heur$ value is explored next. For example, if $heur$ is always equal to $cost + remaining$ then it is guaranteed that the first successful trace is also the cheapest one[3].

A further extension of priced timed automata is presented in (Rasmussen, Larsen, & Subramani 2004). It is possible to define the costrate depending on the current system

[2] Some extension in the expressiveness of the queries have been made but they are not used in our approach.

[3] We assume that $remaining$ represents an under-approximation.

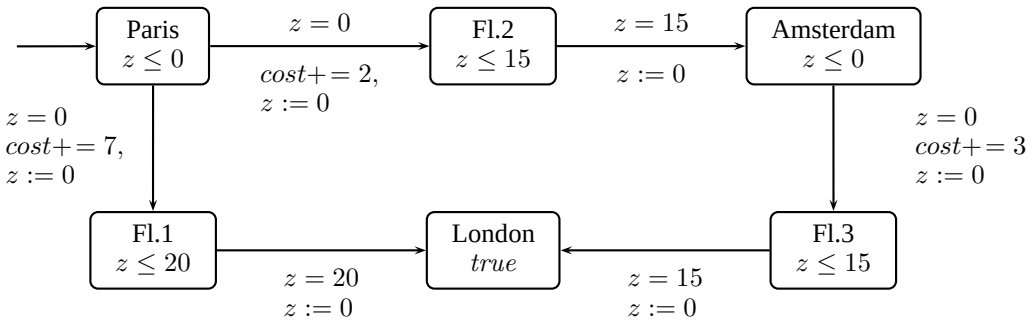


Figure 6: An example of a priced timed automaton.

```
clock x,y,z;
int   k,m,n;
chan  a,b;

process P {
  state  p1 { x <= 11 },
         p2,p3;
  init   p1;
trans
  p1 -> p2 { guard  x>4;
             assign x:=0; },
  p2 -> p3 { guard  x==6;
             sync   a?;
             assign m:=k; };
}

process Q {
  state  q1,q2,q3;
  commit q2;
  init   q1;
trans
  q1 -> q2 { guard  y==13;
             sync   a!;
             assign k:=42;},
  q2 -> q3 { sync   b!; };
}

process R {
  state r1,r2,r3;
  init  r1;
trans
  r1 -> r2 { sync   b?;
             assign n:=k; },
  r2 -> r3 { guard z>28; };
}

system P,Q,R;
```

Figure 5: The example of Fig. 4 in the syntax of UPPAAL.

state which consists of a vector of states of all components. Hence, the current costrate is a linear sum of costrates. Therefore, this extension is called *linearly priced timed automata* (LPTA).

Translation

In this section we explain how to translate the PDDL specifications into input for UPPAAL CORA. In contrast to (Dierks, Behrmann, & Larsen 2002) we can exploit additional expressive power in the syntax of the target and gain therefore readability. Instead of a formal treatment we explain the translation by the example of the Landing Domain.

Global Aspects

For the predicates and functions of the domain we add appropriate (global) variables with appropriate type. Since the type system of UPPAAL CORA supports arrays we can simply produce the following declarations of global variables for the Landing domain.

```
bool occupied[ALL_OF_runway];
bool landed[ALL_OF_plane];
bool scheduled[ALL_OF_plane][ALL_OF_runway];
meta int earliest[ALL_OF_plane];
meta int target[ALL_OF_plane];
meta int latest[ALL_OF_plane];
meta int early_rate[ALL_OF_plane];
meta int late_rate[ALL_OF_plane];
meta int late_penalty[ALL_OF_plane];
```

It is clear that predicates are translated to Boolean and functions to integers. The sizes of the arrays depend on the type of the parameters. For example, for `occupied` we need as many Boolean variables as runways are defined in the problem. `ALL_OF_runway` is a constant that is declared before and depends on the problem specification. In our example we get the following constants:

```
// type plane
const int ALL_OF_plane = 3;
// type runway
const int ALL_OF_runway = 1;
```

An interesting feature of UPPAAL CORA is the definition of `meta` variables. Whenever the model-checker computes

a new state it compare this new state with all previously seen states. However, some variables are considered to be auxiliary only. Hence, it can make sense to leave these variables out when checking a new state. This feature speeds up checks and reduces the stored state space since the tool does not distinguish states which differ only in these auxiliary variables. By the prefix `meta` we can specify such variables. In our translation we exploit this in the case of predicates and function in the domain which are never changed. These variables are only set once, namely by the problem specification. Because all functions of the Landing domain are never changed by the domain's actions we can declare them as `meta`.

Durative Actions

For the translation of durative actions we use process templates of UPPAAL CORA. That means that each durative action is translated into a corresponding template and in the final system declaration in UPPAAL CORA we instantiate some of these templates appropriately. Note that this differs completely from the way the translation was done in (Dierks, Behrmann, & Larsen 2002) because here we get a timed automaton for each instance of a template while in (Dierks, Behrmann, & Larsen 2002) integer variables where used. The advantage of this new approach is the direct correspondence between the PDDL specification and its translation into a template.

These templates can have parameters and the only parameter in our case is an unique identifier. We will explain our translation by the `land_late` action. We get the following process declaration

```
process land_late(const int id) {
int p, r;
clock duration;
int costrate=0;
int min_duration,max_duration;
```

The parameters of the durative action `?p` and `?r` occur here as local integer variables `p` and `r`. In order to measure the duration of the action we add a local clock and two variables `min_duration` and `max_duration` which are needed to record those duration constraints which are given at the begin of the durative action. We also add a variable `costrate` that defines the rate in which the costs are increased (or decreased resp.) while executing the action.

State Space: Next is the state space declaration of the template. The basic idea is that the process starts in a state `idle` where the action is not executed. In order to start the durative action the process guesses instances of the action's parameters `?p` and `?r`. This is done by transitions through two auxiliary states called `guess1` and `guess2`. The intention is that these states are left at same time point as they are entered. Finally, a state `work` is reached that means the durative actions is executed. To end the action a transition to `idle` is fired. In sum, we have the following state space declaration:

```
state idle,
      guess1 { duration <=0},
      guess2 { duration <=0},
      work   { duration<=max_duration
               && cost'==costrate };
commit guess1, guess2;
init idle;
```

This specifies that the automaton can stay in state `idle` arbitrarily long whereas the `guess`-states have to be left within 0 time units. The time it may stay in `work` is restricted by `max_duration` only, since here only upper bounds are legal in UPPAAL CORA. The lower bounds are implemented by transition guards. The cost rate is also specified here. It simply states that the cost rate is given by the variable `costrate`. Hence, we only have to manipulate this variable to change the current cost rate. In order to minimise the search space we also declare the `guess`-states as *committed*. Roughly, the meaning is that the system of all processes first fires transitions first which leave committed locations. That reduces the possible interleavings and saves search space. The declaration of `idle` as initial state is clear.

Action Parameters: Actions in PDDL may have parameters and our template for UPPAAL CORA implements this in the following way. Each action parameter is represented as local variable and for each parameter i we have an auxiliary state `guess`i that guesses the current instance of the parameter i, ie. for each legal value j of the ith parameter we get an unrestricted transition from `guess`$(i-1)$ to `guess`i where the ith parameter is set to j. For $i = 1$ we identify `idle` with `guess`0.

Duration Constraints: In PDDL the duration of durative actions are constrained by inequalities restricting the special variable `?duration`. It is possible to use functions in these inequalities and hence the evaluation may change during execution of the action due to interference by other actions. Therefore, duration constraints are either evaluated `at end` or `at start` where the latter is the default. To models this in UPPAAL CORA we have to store the most restrictive bounds in the local variables `min_duration` and `max_duration`. The check whether all duration constraints are satisfied can only happen when the state `work` is left. Hence, we get the following transitions. Note that only the parts of the transitions are shown which are relevant for the implementation of the duration constraints:

```
trans
  idle -> guess1 {
     guard  ...
     assign ...
            duration:=0,
            ...
  },
  guess1 -> guess2 {
            ...
  },
  ...
  guess2 -> work {
     guard  ...
     assign min_duration:=0,
```

```
            max_duration:=MAX_DURATION,
            max_duration:=
               min(max_duration,
               (latest[p]-earliest[p])),
            ...
            duration:=0;},
work -> idle {
   guard  duration>=min_duration
          && duration<=max_duration;
   assign ...
          min_duration:=0,
          max_duration:=0,
          ...
};
```

In our example action we only have an upper bound which has to be evaluated `at start`. Therefore it is computed in the transition from `guess2` to `work`. The preceding assignments `min_duration:=0, max_duration:=MAX_DURATION` make sure that both variables are initialised properly. `MAX_DURATION` is the maximal positive integer constant. When `work` is entered `duration` is reset to measure the time the action is executed currently. To stop this execution all constraints have to be satisfied. Hence, the guard of the transition from `work` to `idle` contains the check whether the duration is between `min_duration` and `max_duration`. This implements the `at start` constraints. The `land-late` actions has no `at end` duration constraints. But if it had such constraints they would appear here because they have to be evaluated exactly in the moment when the transition takes place.

Conditions and Non-Continuous Effects: Similar to duration constraints both conditions and effect come with a time specification in case of durative actions. The translation of conditions with `at start` and `at end` specification is straightforward. The syntax of guards for transition in UPPAAL CORA has been extended recently to simple `C` syntax. Hence it is basically an exchange of syntax because all PDDL operators have a representative in UPPAAL CORA syntax. The `at start`-conditions are checked when the transition to state `work` is executed. The `at end`-conditions are checked in the guard of the `work`-to-`idle` transition.[4]

An extension to standard PDDL is the possibility of adding *total-time conditions* in durative actions. The syntax is

```
(at start (comp total-time f_expr)
(at end   (comp total-time f_expr)
```

where `comp` is a comparison operator ($=, \geq, \leq, <, >$) and `f_expr` is a functional expression. The obvious meaning is that the total-time of the system and the value of the functional expression should be in the comparison relation. Both the check and the evaluation of the functional expression take place as specified `at start` or `at end` respectively. The translation is straightforward by appropriate comparison with a global clock measuring the total time.

In the case of non-continuous effects the translation has to cope with the simultaneous execution of all effects. In case of Boolean variables this is no problem since it is clear to which value a Boolean variable is set by an effect. However, in case of functions our translation has to translate

```
(and (assign a b) (assign b a))
```

into a sequence of non-simultaneous assignments that switch the values of the nullary functions `a` and `b`. A naive translation using auxiliary variables would add an unnecessary blow up of the search. However, the new feature of `meta` variables in UPPAAL CORA allows us to add for each function `f` (in PDDL) not only the integer variable `f` but also a `meta` variable `new_f` without any price to pay. That means that the example above would be translated to

```
new_a:=b, new_b:=a, a:=new_a, b:=new_b
```

in UPPAAL CORA. A special treatment is necessary for the variable `cost`. It has a special meaning in UPPAAL CORA and it must not occur in functional expressions. In our PDDL variant non-continuous assignments to `cost` are legal but must use the `increase` operator. The specification of the penalty that a late plane has to pay is

```
(at start
  (increase cost (late-penalty ?p)))
```

and it is translated into the assignment

```
cost+=late_penalty[p]
```

which is placed at the assignment of the transition to state `work`.

Continuous Effects for Costs: Our translator deals with a restricted set of continuous effects. It is possible to specify cost rates for a durative action in the following way

```
(increase cost f_expr)
(decrease cost f_expr)
```

Here, `f_expr` stands for an arbitrary functional expression. The meaning is that the costs for the durative action depend on the duration actually needed and (positive or negative resp.) rate is given by `f_expr`. Note that more than one durative action may be active at the same time. But we can exploit that UPPAAL CORA is able to cope with cost rates for each process. The translation of an effect `(increase cost f_expr)` is just an assignment to the local variable `costrate`. In our example we get

```
costrate=late_rate[p]
```

which has to be placed at the assignment of the transition to state `work`.

[4] In PDDL it is possible to specify `over all` for conditions. In this case the semantics requires that this condition has to be satisfied during the whole execution of the action. This could be modelled in UPPAAL CORA by introducing a transition from `work` to `idle` together with the *negated* condition and a synchronisation over an *urgent* channel. This forces UPPAAL CORA to take this transition as soon as the condition is not satisfied. This interrupt should also block all further attempts to reach the goal because interrupted actions are not part of valid plans.

The Problem Definition

Above we discussed the aspects of the domain translation but the current problem is given by a problem specification. The contents of the problem specification can be translated in a very canonical way. For the problem we add a process called `problem` with three states: `initial`, `work`, `goal`. The transition from `initial` to `work` initialises all predicates and functions as specified in the `(:init ...)` part of the problem definition. The transition from `work` to `goal` can only be fired if the property given in the `(:goal ...)` section is satisfied. Thus it suffices to ask UPPAAL CORA how to reach the state `problem.goal` with minimal costs.

The objects in the `(objects: ...)` part are used to instantiate the parameters of the actions. The names are not important and therefore the translation omits this information. It replaces all names by the number of the object with the same type.

In recent extensions of PDDL *timed initial effects* were introduced. It is very easy to add those to the translation of the problem by adding transitions with a guard that requires the correct total time and assignments that implement the desired effects.

Optimisations

The translation introduces further variables in order to avoid searching invalid plans. This happens when the guessing part of a process for a durative action selects instances of the parameters which do not satisfy the conditions given in the domain specification. In this case the only transition left for the process is a transition to idle where a special global Boolean flag `blocked` is set. As soon as this flag is set all actions cannot leave the `idle` state anymore. The effect is that the computation stops and UPPAAL CORA will not invest any time anymore in this branch.

Another important aspect is the number of process instances per template. It is clear that durative actions may overlap and the model-checker should consider such plans since they are often less time consuming than subsequent plans. Therefore it is reasonable to have several instances per template. However, increasing the number of processes increases the search space significantly. To minimise the price we have to pay for additional instances we added the following construction. If a durative action `da` is represented n times in the system for UPPAAL CORA, then we have an Boolean array of size n representing the information whether an instance is currently executing its action. When model-checker wants to start a new action it has to select the unique instance that is currently not active and has the minimal `id` of such inactive instances. This avoids that the tool considers plans which are equivalent up to matching to process instances.

Case Study

We made a series of experiments based on the planes domain.

# Planes	# instances	# clocks	plan	time	mem
3	2	9	0	1.5	9
3	3	13	0	28	57
4	2	9	860	8	32
4	3	13	860	249	479
5	2	9	1540	36	122
5	3	13	out of mem (165 s)		
6	2	9	180	1051	788
6	3	13	out of mem (152 s)		
7	2	9	out of mem (123 s)		
7	3	13	out of mem (139 s)		

We translated each problem specification (the number of planes is given in the first column of the table above) in two ways. The first variant contained two instances of each action template, the second variant had three instances. Consequently we get either 9 resp. 13 clocks in the input of UPPAAL CORA. Since the number of clocks is the most expensive entity w.r.t. complexity we only added information in the table. The result of the model-checking is given in the last three columns. In case of success they contain the *cost* of the optimal plan, the cpu time needed (on a Xeon with 2.66 GHz with memory limit set to 900MB) in seconds and the memory needed in MB.

It is an important aspect that UPPAAL CORA was not guided at all since our prototype does not construct any additional informative heuristics, ie. $remaining$ and $heur$ are not used. Hence, only pruning takes place as soon as a plan was found. It is obvious that appropriate planning techniques can be applied to improve these results. First attempts in this direction are made in the AVACS project[5].

All files of experiments are available at the website of the author:

http://csd.informatik.uni-oldenburg.de/dierks/

Conclusion

It was shown that due to the recent improvements of UPPAAL CORA it makes sense to consider this tool as planning tool for optimal planning problems provided that the continuous effects are restricted to costs only. A matter of future work is to find out to what extend this approach can be extended to less restricted continuous effects.

A key factor whether our approach should be considered for a certain domain is the relation between discrete state space and continuous state space. In many domains – for example those of IPC 2004 – the discrete state space is predominant even in domains where durative actions are used. From our perspective this is mainly due to the evolution of the planning community from the search in huge discrete state spaces. In case of purely discrete domains the planning tools are very powerful since much effort has been spent in the past on such domains. In contrast to that a real-time model-checker like UPPAAL has been tailored to deal with continuous state spaces and it is still an open research issue to find efficient symbolic representations of mixed states

[5] see `http://www.avacs.org` for more information about this project.

spaces. Hence, UPPAAL CORA is a good option for domains where the continuous state space is predominant. As soon as durations and continuous effects are missing or are a minor part of the domain our approach will not be competitive.

Acknowledgements:

The author thanks Gerd Behrmann, Kim G. Larsen and Jacob Illum Rasmussen from Aalborg for supporting this work by building UPPAAL CORA, providing case studies and several suggestions.

Moreover he thanks R. Howey, D. Long and M. Fox from Strathclyde for distributing the plan validator VAL as open source. The translator the author built is based on the parser for PDDL which is distributed together with VAL.

References

Alur, R., and Dill, D. 1990. Automata for modeling real-time systems. In Paterson, M., ed., *ICALP 90: Automata, Languages, and Programming*, volume 443 of *LNCS*, 322–335. Springer.

Alur, R., and Dill, D. 1994. A theory of timed automata. *TCS* 126:183–235.

Behrmann, G.; Fehnker, A.; Hune, T.; Larsen, K.; Pettersson, P.; and Romijn, J. 2001. Efficient Guiding Towards Cost-Optimality in Uppaal. In Margaria, T., and Wang Yi., eds., *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, 174–188. Springer.

Behrmann, G.; David, A.; and Larsen, K. 2004. A Tutorial on UPPAAL. In Bernardo, M., and Corradini, F., eds., *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, 200–236. Springer.

Cassez, F., and Larsen, K. 2000. The Impressive Power of Stopwatches. In *International Conference on Concurrency Theory (CONCUR)*, number 1877 in LNCS, 138–152. Springer.

Dierks, H.; Behrmann, G.; and Larsen, K. 2002. Solving Planning Problems Using Real-Time Model-Checking (Translating PDDL3 into Timed Automata). In Kabanza, F., and Thiebaux, S., eds., *AIPS-Workshop Planning via Model-Checking*, 30–39.

Dierks, H. 2005. Heuristic Guided Model-Checking of Real-Time Systems. submitted for publication.

Fox, M., and Long, D. 2001a. PDDL+ level 5: An Extension to PDDL2.1 for Modelling Planning Domains with Continuous Time-dependent Effects. Technical report, University of Durham.

Fox, M., and Long, D. 2001b. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. Technical report, University of Durham.

Kupferschmid, S.; Hoffmann, J.; Dierks, H.; and Behrmann, G. 2005. Enhancing UPPAAL with Automatically Generated Heuristic Functions: First Results. submitted for publication.

Larsen, K.; Behrmann, G.; Brinksma, E.; Fehnker, A.; Hune, T.; Pettersson, P.; and Romijn, J. 2001. As Cheap as Possible: Efficient Cost-Optimal Reachability for Priced Timed Automata. In Berry, G.; Comon, H.; and Finkel, A., eds., *Proceedings of CAV 2001*, volume 2102 of *LNCS*, 493–505. Springer.

Larsen, K.; Petterson, P.; and Wang Yi. 1997. Uppaal in a nutshell. *STTT* 1(1+2):134–152.

McDermott, D., and the AIPS-98 Planning Competition Committee. 1998. PDDL – The Planning Domain Definition Language. Technical report. Available at: www.cs.yale.edu/homes/dvm.

Rasmussen, J.; Larsen, K.; and Subramani, K. 2004. Resource-Optimal Scheduling Using Priced Timed Automata. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems '04 (TACAS)*, LNCS, 220–235. Springer.

Action Planning for Graph Transition Systems

Stefan Edelkamp[1] **and Shahid Jabbar**[2]
Department of Computer Science
Baroper Str. 301
University of Dortmund,
44221 Dortmund, Germany
[1]stefan.edelkamp@cs.uni-dortmund.de
[2]shahid.jabbar@cs.uni-dortmund.de

Alberto Lluch Lafuente
Dipartimento di Informatica
Università di Pisa, Pisa, Italy
lafuente@di.unipi.it

Abstract

Graphs are suitable modeling formalisms for software and hardware systems involving aspects such as communication, object orientation, concurrency, mobility and distribution. State spaces of such systems can be represented by graph transition systems, which are basically transition systems whose states and transitions represent graphs and graph morphisms. In this paper, we propose the modeling of graph transition systems in PDDL and the application of heuristic search planning for their analysis. We consider different heuristics and present experimental results.

Introduction

Graphs are a suitable modeling formalism for software and hardware systems involving issues such as communication, object orientation, concurrency, distribution and mobility. The graphical nature of such systems appears explicitly in approaches like graph transformation systems (Rozenberg 1997) and implicitly in other modeling formalisms like algebras for communicating processes (Milner 1989). The properties of such systems mainly regard aspects such as temporal behavior and structural properties. They can be expressed, for instance, by logics used as a basis for a formal verification method, like model checking (Clarke, Grumberg, & Peled 1999), which main success is due to the ability to find and report errors.

Finding and reporting errors in model checking and many other analysis problems can be reduced to state space exploration problems. In most cases the main drawback is the *state explosion problem*. In practice, the size of state spaces can be large enough (even infinite) to exhaust the available space and time resources. Heuristic search has been proposed as a solution in many fields, including model checking (Edelkamp, Leue, & Lafuente 2003), planning (Bonet & Geffner 2001) and games (Korf 1985). Basically, the idea is to apply algorithms that exploit the information about the problem being solved in order to guide the exploration process. The benefits are twofold: the search effort is reduced, i.e., errors are found faster and by consuming less memory, and the solution quality is improved, i.e., counterexamples are shorter and thus may be more useful. In some cases, like wide area networks with Quality of Service (QoS), one might not be interested in short paths, but in cheap or optimal ones based on some notion of cost. Therefore, we generalize our approach by considering an abstract notion of costs.

Our work is mainly inspired by approaches to directed model checking (Edelkamp, Leue, & Lafuente 2003), logics for graphs (like the *monadic second order logic* (Courcelle 1997)), spatial logics used to reason about the behavior and structure of processes calculi (Caires & Cardelli 2003) and graphs (Cardelli, Gardner, & Ghelli 2002), and approaches for the analysis of graph transformation systems (Baldan *et al.* 2004; Rensink 2003; Varrò 2003). At the theoretical front, our approach is very much inspired by cost-algebraic search algorithms (Sobrinho 2002; Edelkamp, Jabbar, & Lluch-Lafuente 2005a).

The work also relates to (Edelkamp 2003a) that compiled protocol software model checking domains in Promela to PDDL. Two of such domains have served as a benchmark for the 4th international planning competition in 2004 (Hoffmann *et al.* 2005). We extend the work of (Edelkamp, Jabbar, & Lluch-Lafuente 2005b) that applies heuristic search for graph transition systems in the context of the experimental model checker HSF-SPIN (Edelkamp, Leue, & Lafuente 2003). To the best of our knowledge this is the first work on action planning for the analysis of graphically described systems, probably with the exception of one currently running master's thesis (Golkov 2005).

The goal of our approach is to formalize structural properties of systems modeled by graph transition systems. We believe that our work additionally illustrates the benefits of applying heuristic search in state space exploration systems. Heuristic search is intended to reduce the analysis effort and, in addition, to deliver shorter or optimal solutions. We consider a notion of optimality with respect to a certain cost or weight associated to system transitions. For instance, the cost of a transition in network systems can be a certain QoS value associated to the transition.

The next section introduces the running example that is used along the paper to illustrate some of the concepts and methods. Next, we define our modeling formalism, namely graph transition systems. We then consider the kind of properties we are interested in verifying and discuss their PDDL model. We study two planning heuristics for the analysis of properties in graph transition systems and present experimental results obtained with a heuristic search planner. Finally, we conclude the paper and outline future research avenues. For the sake of readability the PDDL model for the arrow distributed protocol is included in an appendix that follows the bibliography.

The Arrow Distributed Directory Protocol

The *arrow distributed directory protocol* (Demmer & Herlihy 1998) is a solution to ensure exclusive access to mobile objects in a distributed system. The distributed system is given as an undirected graph G, where vertices and edges respectively represent nodes and communication links. Costs are associated with the links in the usual way, and a mechanism for optimal routing is assumed.

The protocol works with a minimal spanning tree T of G. Each node has an arrow which, roughly speaking, indicates the *direction* in which the object lies. If a node owns the object or is requesting it, the arrow points to itself; we say that the node is *terminal*. The directed graph induced by the arrows is called $\mathcal{L}$. Roughly speaking, the protocol works by propagating requests and updating arrows such that at any moment the paths induced by arrows, called *arrow paths*, either lead to a terminal owning the object or waiting for it.

More precisely, the protocol works as follows: Initially $\mathcal{L}$ is set such that every path leads to the node owning the object. When a node u wants to acquire the object, it sends a request message *find*(u) to $a(u)$, the target of the arrow starting at u, and sets $a(u)$ to u, i.e., it becomes a terminal node. When a node u whose arrow does not point to itself receives a *find*(w) message from a node v, it forwards the message to node $a(u)$ and sets $a(u)$ to v. On the other hand, if $a(u) = u$ (the object is not necessarily at u but will be received if not) the arrows are updated as in the previous case but this time the request is not forwarded but enqueued. If a node owns the object and its queue of requests is not empty, it sends the object to the (unique) node u of its queue sending a *move*(u) message to v. This message goes *optimally* through G. A formal definition of the protocol can be found in (Demmer & Herlihy 1998).

Figure 1 illustrates three states of a protocol instance with six nodes $v_0, \ldots, v_5$. The state on the left is the initial one: node v_0 has the object and all paths induced by the arrows lead to it. The state on the right of the figure is the result of two steps: node v_4 sends a request for the object through its arrow; and v_3 processes it by updating the arrows properly, i.e., the arrow points now to v_4 instead of v_2.

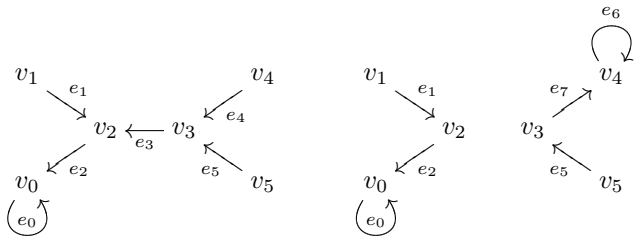


Figure 1: Three states of the directory.

One could be interested in properties like *Can node v_i be a terminal?* (Property 1), *Can node v_i be terminal and all arrow paths end at v_i?* (Property 2), *Can a node v be terminal?* (Property 3), *Can a node v be terminal and all arrow paths end at v?* (Property 4).

Graph Transition Systems

This section presents our modeling formalism. First, an algebraic notion of costs is defined. It shall be used as an abstraction of costs or weights associated to edges of graphs or transitions of transition systems. For a deeper treatment of the cost algebra we refer to (Edelkamp, Jabbar, & Lluch-Lafuente 2005a).

Definition 1 *A* cost algebra *is a 6-tuple $\langle A, \bigsqcup, \times, \preceq, \boldsymbol{0}, \boldsymbol{1}\rangle$, such that*

1. *$\langle A, \times\rangle$ is a monoid with $\boldsymbol{1}$ as identity element and $\boldsymbol{0}$ as its absorbing element, i.e., $a \times \boldsymbol{0} = \boldsymbol{0} \times a = \boldsymbol{0}$;*
2. *$\preceq \subseteq A \times A$ is a total ordering with $\boldsymbol{0} = \bigsqcap A$ and $\boldsymbol{1} = \bigsqcup A$;*
3. *A is* isotone*, i.e., $a \preceq b$ implies both $a \times c \preceq b \times c$ and $c \times a \preceq c \times b$ for all $a, b, c \in A$ (Sobrinho 2002).*

In the rest of the paper $a \prec b$ abbreviates $a \preceq b$ and $a \neq b$. Moreover, $a \succeq b$ abbreviates $b \preceq a$, and $a \succ b$ abbreviates $a \succeq b$ and $a \neq b$. The *least* element c in A is defined as $\bigsqcup A$, if $c \in S$ and $c \preceq a$ for all $a \in A$. The *greatest* element c in A is defined as $\bigsqcap A$, if $c \in A$ and $c \preceq a$ for all $a \in A$.

Intuitively, A is the domain set of cost values, $\times$ is the operation used to cumulate values and $+$ is the operation used to select the best (the least) amongst two values. Consider for example, the following instances of cost algebras, typically used as cost or QoS formalisms:

- $\langle \{true, false\}, \vee\wedge, \Rightarrow, false, true\rangle$ (Network and service availability)
- $\langle \mathbb{R}^+ \cup \{+\infty\}, \min, +, \leq, +\infty, 0\rangle$ (Price, propagation delay)
- $\langle \mathbb{R}^+ \cup \{+\infty\}, \max, \min, \geq, 0, +\infty\rangle$ (Bandwidth).

In the rest of the paper, we consider a fixed cost algebra $\langle A, \bigsqcup, \times, \preceq, \mathbf{0}, \mathbf{1}\rangle$.

Definition 2 *An (*edge-weighted graph*) G is a tuple $\langle V_G, E_G, src_G, tgt_G, \omega_G\rangle$ where V_G is a set of nodes, E_G*

is a set of edges, $src_G, tgt_G : E_G \to V_G$ are a source and target functions, and $\omega_G : E_G \to A$ is a weighting function.

Graphs usually have a distinguished start state which we denote with s_0^G, or just s_0 if G is clear from the context.

Definition 3 *A* path *in a graph G is an alternating sequence of nodes and edges represented as $u_0 \xrightarrow{e_0} u_1 \ldots$ such that for each $i \geq 0$ we have $u_i \in V_G$, $e_i \in E_G$, $src_G(e_i) = u_i$ and $tgt_G(e_i) = u_{i+1}$, or, shortly $u_i \xrightarrow{e_i} u_{i+1}$.*

An initial path is a path starting at s_0^G. Finite paths are required to end at states. The length of a finite path p is denoted by $|p|$. The concatenation of two paths p, q is denoted by pq, where we require p to be finite and end at the initial state of q. The cost of a path is the cumulative cost of its edges. Formally,

Definition 4 *Let $p = u_0 \xrightarrow{e_0} \ldots \xrightarrow{e_{k-1}} u_k$ be a finite path in a graph G. The* path cost *$\omega_G(p)$ is $\omega_G(e) \times \omega_G(q)$ if $p = (u \xrightarrow{e} v)q$ and $\mathbf{1}$ otherwise. If $|q| = 0$, $\omega_G(q) = \mathbf{1}$.*

Let $\gamma(u)$ denote the set of all paths starting at node u. In the sequel, we shall use $\omega_G^*(u, V)$ to denote the cost of the optimal path starting at a node u and reaching a node v in a set $V \subseteq V_G$. Formally, $\omega_G^*(u, V) = \bigsqcup_{p \in \gamma(s) | (p \cap V) \neq \emptyset} \omega_G(p)$. For ease of notation, we write $\omega_G^*(u, \{v\})$ as $\omega_G^*(u, v)$.

Graph transition systems are suitable representations for software and hardware systems and extend traditional transition systems by relating states with graphs and transitions with partial graph morphisms. Intuitively, a partial graph morphism associated to a transition represents the relation between the graphs associated to the source and the target state of a transition. More specifically, it models the merging, insertion, addition and renaming of graph items (nodes or edges). In case of a merge, the cost of merged edges is the least one amongst the edges involved in the merging.

Definition 5 *A* graph morphism *$\psi : G_1 \to G_2$ is a pair of mappings $\psi_V : V_{G_1} \to V_{G_2}$, $\psi_E : E_{G_1} \to E_{G_2}$ such that we have $\psi_V \circ src_{G_1} = src_{G_2} \circ \psi_E$, $\psi_V \circ tgt_{G_1} = tgt_{G_2} \circ \psi_E$,[1] and for each $e \in E_{G_2}$ we have, $\omega_{G_2}(e) = \bigsqcup\{\omega_{G_1}(e') \mid \psi_E(e') = e\}$.*
A graph morphism $\psi : G_1 \to G_2$ is called injective *if so are ψ_V and ψ_E;* identity *if both ψ_V and ψ_E are identities, and isomorphism if both ψ_E and ψ_V are bijective.*
A graph G' is a subgraph *of graph G, if $V_{G'} \subseteq V_G$ and $E_{G'} \subseteq E_G$, and the inclusions form a graph morphism.*
A partial graph morphism *$\psi : G_1 \to G_2$ is a pair $\langle G_1', \psi_m \rangle$, where G_1' is a subgraph of G_1, and $\psi_m : G_1' \to G_2$ is a graph morphism.*

The composition of (partial) graph morphisms results in (partial) graph morphisms. Now, we define a notion of transition system that enriches the usual ones with weights.

Definition 6 *A* transition system *is a graph $M = \langle S_M, T_M, in_M, out_M, \omega_M \rangle$ whose nodes and edges are respectively called* states *and* transitions*, with in_M, out_M representing the source and target of an edge respectively.*

Finally, we are ready to define graph transition systems, which are transition systems together with morphisms mapping states into graphs and transitions into partial graph morphisms.

Definition 7 *A* graph transition system *(GTS) is a pair $\langle M, g \rangle$, where M is a weighted transition system and $g : M \to \mathcal{U}(\boldsymbol{G}_p)$ is a graph morphism from M to the graph underlying $\boldsymbol{G}_p$, the category of graphs with partial graph morphisms. Therefore $g = \langle g^S, g^T \rangle$, and the component on states g^S maps each state $s \in S_M$ to a graph $g^S(s)$, while the component on transitions g^T maps each transitions $t \in T_M$ to a partial graph morphism $g^T(t) : g^S(in_M(t)) \Rightarrow g^S(out_M(t))$.*

In the rest of the paper we shall consider a GTS $\langle M, g \rangle$ modeling the state space of our running example, where g maps states to $\mathcal{L}$, i.e., the graph induced by the arrows, and transitions to the corresponding partial graph morphisms. Consider Figure 1, each of the three graphs depicted, say G_1, G_2 and G_3 corresponds to three states s_1,s_2,s_3, meaning that $g(s_1) = G_1$, $g(s_2) = G_2$ and $g(s_3) = G_3$. The figure illustrates a path $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3$, where $g(t_1)$ is the identity restricted to all items but edge e_4. Similarly, $g(t_2)$ is the identity restricted to all items but edge e_3. Thus, in both transitions all other items are preserved (with their identity) except the edges mentioned.

Properties of Graph Transition Systems

The properties of a graph transition system can be expressed using different formalisms. One can use, for instance, a temporal graph logic like the ones proposed in (Baldan *et al.* 2004; Rensink 2003), which combine temporal and graph logics. A similar alternative are spatial logics (Caires & Cardelli 2003), which combine temporal and structural aspects. In graph transformation systems (Corradini *et al.* 1997), one can use rules to find certain graphs: the goal might be to find a match for a certain transformation rule. For the sake of simplicity and generality, however, we consider that the problem of satisfying or falsifying a property is reduced to the problem of finding a set of *goal states* characterized by a goal graph and the existence of an injective morphism.

Definition 8 *Given a GTS $\langle M, g \rangle$ and a graph G, the* goal function $\text{goal}_G : S_M \to \{\text{true}, \text{false}\}$ *is defined such that* $\text{goal}_G(s) = \text{true}$ *iff there is an injective graph morphism $\psi : G \to g(s)$.*

Intuitively, *$goal_G$* maps a state s to *true* if and only if G can be injectively matched with a subgraph of $g(s)$.

[1] $\circ$ is the function composition operator. In other words $f \circ g = f(g(\cdot))$

Figure 2: Three graphs illustrating various goal criteria.

It is worth saying that most graph transformations approaches consider injective rules, for which a match is precisely given by injective graph morphisms, and that the most prominent graph logic, namely the Monadic Second-Order (MSO) logic by (Courcelle 1997) and its first-order fragment (FO) can be used to express injective graph morphisms. The graph G will be called *goal graph*. It is of practical interest identifying particular cases of goal functions as the following goal types:

1. ψ is an identity - the exact graph G is looked for. In our running example, this corresponds to Property 2 mentioned in Section . For instance, we look for the exact graph depicted in left of Figure 2.
2. ψ is a restricted identity - an exact subgraph of G is looked for. This is precisely Property 1. For instance, we look for a subgraph of the graph depicted in left of Figure 2. The graph in center of Figure 2 satisfies this.
3. ψ is an isomorphism - a graph isomorphic to G is looked for. This is precisely Property 4. For instance, we look for a graph isomorphic to the one depicted in left of Figure 2. The graph in the right of Figure 2 satisfies this.
4. ψ is any injective graph morphism - we have the general case. This is precisely Property 3. For instance, we look for an injective match of the graph depicted in center of Figure 2. The graph in the right of Figure 2 satisfies this.

Note that there is a type hierarchy, since goal type 1 is a subtype of goal types 2 and 4, which are of course subtypes of the most general goal type 4.

The computational complexity of the goal function varies according to the above cases. For goals of type 1 and 2, the computational efforts needed are just $O(|G|)$ and $O(|\psi(G)|)$, respectively. Unfortunately, for goal types 3 and 4, due to the search for isomorphisms, the complexity increase to a term exponential in $|G|$ for the graph isomorphism case and to a term exponential in $|\psi(G)|$ for the subgraph isomorphism case. The general problem of subgraph isomorphism (SI) can be reduced polynomially to graph isomorphism. Subgraph isomorphism is NP-complete, as CLIQUE $\leq_p$ SI. The general problem of graph isomorphism is not completely classified. It is expected not to be NP-complete (Wegener 2003).

Now we state the two analysis problems we consider. The first one consists on finding a goal state.

Definition 9 *Given a GTS $\langle M, g\rangle$ and a graph G (the goal graph), the* reachability problem *of our approach consists on finding a state $s \in S_M$ such that* $\text{goal}(s)$ *is* true.

The second problem aims at finding an optimal path to a goal state.

Definition 10 *Given a GTS (M, g) and a graph G (the goal graph), the* optimality problem *of our approach consists on finding a finite initial path p ending at a state $s \in S_M$ such that such that* $\text{goal}_G(s)$ *is* true *and* $\omega(p) = \omega^*_M(s^M_0, S')$*, where* $S' = \{s \in S_M \mid \text{goal}_G(s) = \text{true}\}$.

For the sake of brevity, in the following $\omega^*_M(s)$ abbreviates $\omega^*_M(s, S')$ with $S' = \{s \in S_M \mid goal_G(s) = true\}$, when $goal_G$ is clear from the context.

The two problems defined in the previous section can be solved with traditional graph exploration and shortest-path algorithms[2]. For the reachability problem, for instance, one can use, amongst others, depth-first search, hill climbing, best-first search, Dijkstra's algorithm (and its simplest version breadth-first search) or A*. For the optimality problem, only the last two are suited.

Nevertheless, Dijkstra's algorithm and A* are traditionally defined over a simple instance of our cost algebra A, namely algebra $\langle \mathbb{R}^+ \cup \{+\infty\}, \min, +, \leq, +\infty, 0\rangle$. Fortunately, the results that ensure the *admissibility* of Dijkstra's algorithm or A*, i.e., the fact that both algorithms correctly solve the optimality problem, have been generalized for the cost algebra (Edelkamp, Jabbar, & Lluch-Lafuente 2005a).

Encoding of the Arrow Distributed Directory Protocol

To simplify the discussion, we assume a uniform transition weight leading to pure propositional planning problems. But with the extensions that are available in current planning description languages such as PDDL2.1 (Fox & Long 2003), the current setting can be extended to numerical weights. Note that, the formal treatment of the problem presented earlier in this paper is capable of dealing with non-uniform weights.

In propositional planning, for each state we have atomic propositions that can either be true or false. Planning operators or actions change the truth values of atomic propositions AP. An action a in STRIPS consists of three lists: precondition, add, and delete lists, commonly denoted as $pre(a)$, $add(a)$, and $del(a)$, respectively (Fikes & Nilsson 1971). Each list consists of atomic propositions and the application of a to a state $S \subseteq 2^{AP}$ with $pre(a) \subseteq S$ yields the successor state $(S \setminus del(a)) \cup add(a)$.

To apply a planner to graph transition systems, we first need a propositional description of graph transition systems in PDDL. The graph is modeled with the help

[2]We refer here to a slight modification of the original algorithms, consisting of terminating the algorithm when a goal state is reached and returning the corresponding path.

of predicates defining the edges. We use `(link u v)` predicate to denote an edge between two nodes u and v. Since a node cannot exist on its own, we do not provide any predicate to declare a node. The predicate `(find-pending u v w)` is true if the node u receives a request from its neighbour v to find the object for the node w. Similarly, the predicate `(move-pending u v w)` is true, if the node u receives an object from the node w to be forwarded to the requesting node. The parameter v is actually not in use and its just for the sake of uniformity with the original model and with the `find-pending` predicate.

The predicate `(not-request-send u)` is used to control the requests generated by the nodes so that a node cannot request more than once. The contents of a queue attached with a particular node u are controlled through the `(queue u v)` predicate. The ownership of the object is determined through the `(owner u)` predicate.

Due to the parametric description facility provided by planning formalism, it is easier to define morphisms and partial morphisms as actions. For example, a morphism operation that inverses an edge can easily be defined as a very simple action as follows:

```
(:action morphism-inverse
:parameters(?u ?v - node)
:precondition
  (link ?u ?v)
:effect
  (and
    (not (link ?u ?v))
    (link ?v ?u)))
```

An example description for the Arrow Protocol is provided in Annex.

Problem description in PDDL

A GTS problem can be described with the help of predicates defining the graph in the initial state. The whole graph can be described by the use of `link` predicates defining the edges between different nodes of the graph. The owner node, i.e., the node that currently owns the object is define by the use of `owner` predicate.

A PDDL problem description for an instance of *star*-shaped network topology is shown in the appendix.

Goal Specification in PDDL

Fortunately, PDDL provides a very neat and elegant mechanism to formulate our goals' criteria. In the following we explain various methods to describe different types of goals.

Property 1 goal (subgraph): Perhaps the most simple to describe are the type 1 goals as we only search for a specific subgraph. As is evident from the PDDL specification of the domain, the subgraph can easily be declared by using the `(link u v)` predicates. If the subgraph to be searched for actually asks for an ownership predicate to be true for some node w, we simply declare the `(owner w)` predicate as our goal criteria. In Appendix, we see an example problem description in PDDL where a goal of type 1 is searched for.

Property 2 goal (exact graph): For a *Property 2* goal, we look for an exact matching of the goal graph in our state space. Just like for the previous type, we can describe the whole graph with `(link u v)` predicates. Note that it is true only for the current domain, since a spanning tree property of the graphs is preserved through out the search space, i.e., there cannot be a reachable state where the graph is a superset of the goal graph. This might not be the case in other GTS domains. In such cases we have to describe the non-existence of all the other edges too.

Property 3 goal (subgraph isomorphism): Given a goal graph G, the state space is searched for a state that contains a subgraph isomorphic to G. In such case goals are strictly more expressive and need an existential quantification over all the nodes to be described succinctly. Existential quantification can be incorporated in STRIPS through ADL (Pednault 1989) by the following construct:

```
(:goal <existential-expression> <goal-condition>)
```

A goal of type 3 can then be included in our problem specification as:

```
(:goal (exists (?n - node) (owner ?n))
```

Property 4 goal (isomorphism): Given a goal graph G, the state space is searched for a node that contains a graph isomorphic to G. Having the existential quantifier in our hands, we can describe G using `(link u v)` predicates. For our example in Figure 2, a type 4 goal will have the form:

```
(:goal (exists ?v0 ?v1 ?v2 ?v3 ?v4 ?v5 - node)
 (and (link ?v0 ?v0)  (link ?v1 ?v0)
      (link ?v2 ?v0)  (link ?v3 ?v1)
      (link ?v4 ?v0)  (link ?v5 ?v4)
      (owner ?v3)))
```

Given actions with ADL expressivity, it is not difficult to transform an existential goal description to a non-existential one by adding the following special operator to the domain description:

```
(:action goal-achieving-action
 :precondition <old-goal-condition>
 :effects (and (goal-achieved)))
```

The modified goal condition then simplifies to

```
(:goal (goal-achieved))
```

With the extended expressivity of PDDL2.2 (Edelkamp & Hoffmann 2004) goal achievement is best introduced in form of domain axioms, so-called derived predicates. They are inferred in form of a fix-point computation with rules that do not belong to a plan. For this case we include

```
(:derived (goal-achieved) <old-goal-condition>)
```

to the domain description.

Planning Heuristics for Graph Transition Systems

Heuristic search algorithms use heuristic functions to guide the state space exploration as apposed to blind search algorithms that do not utilize any information about the search space. Two of the most famous heuristic search algorithms are A* and IDA*. A* utilizes a heuristic estimate for the distance from a state to the goal, to prioritize states' expansion. The result is a reduced search space; consequently, less consumption of memory with gain in speed. A* is guaranteed to produce optimal results in case of admissible and consistent heuristic.

Most of the modern planners (for example, FF (Hoffmann & Nebel 2001) or MIPS (Edelkamp 2003b)) utilize various heuristics to guide the planner. Two of such heuristics that have performed very good in planning domains are *relaxed planning heuristic* and *planning pattern databases*.

Relaxed Planning Heuristic

A *relaxed planning heuristic* (Hoffmann & Nebel 2001) is computed by solving a *relaxed version* of a planing problem. The *relaxation* a^+ of a STRIPS action $a = (pre(a), add(a), del(a))$ is defined as $a^+ = (pre(a), add(a), \emptyset)$. The relaxation of a planning problem is the one in which all actions are substituted by their relaxed counterparts. Any solution that solves the original plan also solves the relaxed one; and all preconditions and goals can be achieved if and only if they can be in the relaxed task. Value h^+ is defined as the length of the shortest plan that solves the relaxed problem.

Solving relaxed plans optimally is still computationally hard (Bylander 1994), but the decision problem to determine, if a relaxed planning problem has at least one solution, is computationally tractable. The optimization task can efficiently be approximated by counting the number of operators in a *parallel plan* that solves the relaxed problem. Note that optimal parallel and optimal sequential plans may have a different sets of operators, but good parallel plans are at least informative for sequential plan solving, and can, therefore, be used for the design of a heuristic estimator function.

The extension to the *numerical relaxed planning heuristic* is a polynomial-time state evaluation function for mixed integer domain-independent planning problems (Hoffmann 2003). It has been extended to nonlinear tasks (Edelkamp 2004).

Planning Pattern Databases

Abstraction is one of the most important issues to cope with large and infinite state spaces, and to reduce the exploration efforts. Abstracted systems should be significantly smaller than the original one while preserving some properties of concrete systems. The study of abstraction formalisms for graph transition systems is, however, out of the scope of this paper. We refer to (Baldan *et al.* 2004) for an example of such a formalism. Assuming that abstractions are available, we state the properties necessary for abstractions to preserve our two problems (reachability and optimization) and propose how to use abstraction to define informed heuristics.

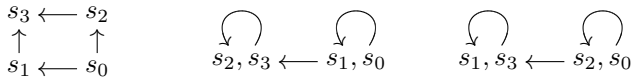


Figure 3: A transition system (leftmost) with two different abstractions.

The preservation of the reachability problem means that the existence of an initial goal path in the concrete system must entail the existence of a corresponding initial goal path in the abstract system. Note that this does not mean the existence of *spurious* initial goal paths in the abstract system, i.e., abstract paths that do not correspond to any concrete path. Similarly, the preservation of the optimization problem means that the cost of the optimal initial goal path in the concrete system should be greater or equal to the cost of the optimal initial goal path in the abstract system.

Abstractions have been applied in combination with heuristic search in single-agent games (Culberson & Schaeffer 1998; Korf 1997), in model checking (Edelkamp & Lluch-Lafuente 2004) and planning (Edelkamp 2001) approaches. The main idea is that the abstract system is explored in order to create a database that stores the exact distances from abstract states to the set of abstract goal states. The exact distance between abstract states is an admissible and consistent estimate of the distance between the corresponding concrete states. The distance database is thus used as heuristics for analyzing the concrete system.

When different abstractions are avaliable, we can combine the different databases in various ways to obtain better heuristics. The first way is to trivially select the best value delivered by two heuristic databases, which trivially results in a consistent and admissible heuristic. Figure 3 depicts a concrete transition system (left) with three abstractions (given by node mergings). The two abstractions are mutually disjoint.

Experimental Results

We validate our approach by presenting initial experimental results obtained with the heuristic search planning system FF. We have implemented the *arrow distributed directory protocol* in PDDL2.1, Level 1, i.e. in the specification language STRIPS/ADL. We performed our experiments on a Pentium IV 3.2 GHz. machine with Linux operating system and 2 gigabytes of internal memory. In all our experiments we set a memory bound of 2 GB.

When running the planner on the instances, we obtain the results as shown in Table 1 in comparison with

	HSF-SPIN		FF
star	DFS	BFS_{h_f}	$EHC + RPH$
Stored nodes	6,253	30	6
Sol. length	134	58	5
chain	DFS	BFS_{h_f}	$EHC + RPH$
Stored nodes	78,112	38	6
Sol. length	118	74	5
tree	DFS	BFS_{h_f}	$EHC + RPH$
Stored nodes	24,875	34	6
Sol. length	126	66	5

Table 1: Comparison of results between HSF-SPIN and FF.

	# Nodes	*Stored Nodes*	*Sol. Length*
star	10	6	5
	25	7	6
	50	7	6
	70	7	6
chain	10	6	5
	25	33	28
	50	100	73
	70	138	101
tree	10	6	5
	25	22	16
	50	47	25
	70	61	31

Table 2: Scaling behaviour of the model.

the results that we have obtained in the model checking domain through our experimental model checker HSF-SPIN. The goal searched for is of type 2. Column DFS shows the results while running HSF-SPIN with depth-first search as the exploration algorithm. The gain in HSF-SPIN by employing a heuristic guided exploration as apposed to DFS is noticeable in column BFS_{h_f}. The heuristic estimate used here is based on original *formula-based heuristic* (Edelkamp, Leue, & Lafuente 2003) that exploits the length of the specification of goal states to guide the search algorithm. A discussion on this heuristic is out of the scope of this paper and we refer the reader to (Edelkamp, Leue, & Lafuente 2003) for a detailed treatment.

For all three topologies, namely, *star*, *tree*, and *chain*, the planner resulted in much lesser expansions of nodes. Note that, though the results through the use of planner seem by far better than the one by model checker, we cannot actually compare the two approaches with each other for several reasons. A crucial difference is the dynamic creation of nodes during exploration. PDDL specifications currently do not support such kind of dynamism in models. For a limited case, we can utilize the visibility paradigm of domain specification by providing a pool of invisible nodes to the planner along with the model. These nodes can be made visible whenever a new node is required to be created. The other crucial difference is the modeling of finite and bounded channels - one of the main component of a concurrent system. Such channels can be defined in a model checker but not in PDDL.

In Table 2, we depict the scaling behaviour of the problem for different topologies. We generated random graphs with random `owner`s and with random goals. The second column shows the number of nodes that composed the graph. Column 3, *Stored Nodes*, shows the number of nodes stored during the serach. The length of the solution obtained is shown in the fourth column. For *star* topology, the problem was quite simple. But a major shift in space and time requirement was noted when we switched to the *chain* topology. The longest running example in *chain* topology was with 70 nodes that took about 139 secs to be solved. The scaling factor of memory usage turned out to be very sharp. A 50 nodes problem required about 0.5 GB that jumped to 1.9 GB for 70 nodes in all the topologies. Unfortunately this was also the capacity of our machine - the reason that we are unable to show the results for bigger models.

Conclusion

We have presented an abstract approach for the analysis of graph transitions systems, which are traditional transition systems where states and transitions respectively represent graphs and partial graph morphisms. It is a useful formalism to represent the state space of systems involving graphs, like communication protocols, graph transformations, and visually described systems.

The analysis of such systems is reduced to exploration problems consisting on finding certain states reachable from the initial one. We analyze two problems: finding just one path and finding the optimal one, according to a certain notion of optimality. As specification formalism, we propose the use of ADL. It is capable of expressing all four types of goals that we have suggested. In addition, we have proposed the use of abstraction-based heuristics which exploit abstraction techniques in order to obtain informed heuristics.

We have illustrated our approach with a scenario in which one is interested in analyzing structural properties of communication protocols. As a concrete example we used the *arrow distributed directory protocol* (Demmer & Herlihy 1998) which ensures exclusive access to a mobile service in a distributed system. We implemented our approach in a heuristic search planning system, and presented experiments validating our approach. The PDDL specification presented in this paper is one of the first steps towards modeling *arrow distributed directory protocol* and still has some of the specifications unmodeled such as a bounded queue to prioritize the requests.

In the 2004 International Planning Competition (IPC-4)[3], a Promela domain was used for the first time

[3] `http://ipc.icaps-conference.org/`

as an AI planning problem. This opened new horizons to bridge model checking with AI planning. This paper is one of the first efforts to model the systems represented by Graph Transition Systems as an AI planning problem. There is still a lot of room for expansion for the ideas presented in this paper.

In future work we would like to investigate further scenarios for the analysis of graph transformation systems to planning problems. One such direction is to model other more complicated protocols than the Arrow Distributed Directory protocol. With more challenging problem instances, we expect that graph transition system can serve as a challenging benchmark for upcoming planning competitions.

References

Baldan, P.; Corradini, A.; König, B.; and König, B. 2004. Verifying a behavioural logic for graph transformation systems. In *CoMeta'03*, ENTCS.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1–2):5–33.

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artificial Intelligence* 165–204.

Caires, L., and Cardelli, L. 2003. A spatial logic for concurrency (part I). *Inf. Comput.* 186(2):194–235.

Cardelli, L.; Gardner, P.; and Ghelli, G. 2002. A spatial logic for querying graphs. In *ICALP'2002*, volume 2380 of *Lecture Notes in Computer Science*, 597–610. Springer.

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

Corradini, A.; Montanari, U.; Rossi, F.; Ehrig, H.; Heckel, R.; and Löwe, M. 1997. *Algebraic approaches to graph transformation*, volume 1. World Scientific. chapter Basic concepts and double push-out approach.

Courcelle, B. 1997. *Handbook of graph grammars and computing by graph transformations*, volume 1 : Foundations. World Scientific. chapter 5: The expression of graph properties and graph transformations in monadic second-order logic, 313–400.

Culberson, J. C., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(4):318–334.

Demmer, M. J., and Herlihy, M. 1998. The arrow distributed directory protocol. In *International Symposium on Distributed Computing (DISC 98)*, 119 –133.

Edelkamp, S., and Hoffmann, J. 2004. PDDL2.2: The language for the classical part of the 4th international planning competition. Technical Report 195, University of Freiburg.

Edelkamp, S., and Lluch-Lafuente, A. 2004. Abstraction databases in theory and model checking practice. In *ICAPS Workshop on Connecting Planning Theory with Practice*.

Edelkamp, S.; Jabbar, S.; and Lluch-Lafuente, A. 2005a. Cost-algebraic heuristic search. In *Proceedings of Nineteenth National Conference on Artificial Intelligence (AAAI'05)*. To appear.

Edelkamp, S.; Jabbar, S.; and Lluch-Lafuente, A. 2005b. Heuristic search for the analysis of graph transition systems. draft.

Edelkamp, S.; Leue, S.; and Lafuente, A. L. 2003. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer (STTT)* 5(2-3):247–267.

Edelkamp, S. 2001. Planning with pattern databases. In *European Conference on Planning (ECP)*, Lecture Notes in Computer Science. Springer. 13-24.

Edelkamp, S. 2003a. Promela planning. In *Model Checking Software (SPIN)*, 197–212.

Edelkamp, S. 2003b. Taming numbers and durations in the model checking integrated planning system. *Journal of Artificial Research (JAIR)* 20:195–238.

Edelkamp, S. 2004. Generalizing the relaxed planning heuristic to non-linear tasks. In *German Conference on Artificial Intelligence (KI)*. 198–212.

Fikes, R., and Nilsson, N. 1971. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Research*. Special issue on the 3rd International Planning Competition.

Golkov, E. 2005. Graphtransformation als Planungsproblem. Master's thesis, University of Dortmnund. draft.

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

Hoffmann, J.; Englert, R.; Liporace, F.; Thiebaux, S.; and Trüg, S. 2005. Towards realistic benchmarks for planning: the domains used in the classical part of IPC-4. Submitted.

Hoffmann, J. 2003. The Metric FF planning system: Translating "Ignoring the delete list" to numerical state variables. *Journal of Artificial Intelligence Research* 20:291–341.

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

Korf, R. E. 1997. Finding optimal solutions to Rubik's Cube using pattern databases. In *National Conference on Artificial Intelligence (AAAI)*, 700–705.

Milner, R. 1989. *Communication and Concurrency*. Prentice Hall.

Pednault, E. P. D. 1989. ADL: Exploring the middleground between STRIPS and situation calculus. In *Knowledge Representation and Reasoning (KR)*, 324–332. Morgan Kaufman.

Rensink, A. 2003. Towards model checking graph grammars. In *3rd Workshop on Automated Verification of Critical Systems*, Tech. Report DSSE-TR-2003, 150–160.

Rozenberg, G., ed. 1997. *Handbook of graph grammars and computing by graph transformations*. World Scientific.

Sobrinho, J. 2002. Algebra and algorithms for QoS path computation and hop-by-hop routing in the internet. *IEEE/ACM Trans. Netw.* 10(4):541–550.

Varrò, D. 2003. Automated formal verification of visual modeling languages by model checking. *Journal on Software and Systems Modeling*.

Wegener, I. 2003. *Komplexitätstheorie*. Springer. (in German).

Appendix

PDDL model of the arrow distributed directory protocol as described in *The Arrow Distributed Directory Protocol* by M. J. Demmer and M. P. Herlihy.

Domain Description

```
(define (domain arrow-domain)
(:requirements :typing :strips)
(:types node - object)
(:predicates
  (link ?n1 ?n2 - node)
  (queue ?n1 ?n2 - node)
  (owner ?n1 - node)
  (not-request-send ?n1 - node)
  (find-pending ?n1 ?n2 ?n3 - node)
  (move-pending ?n1 ?n2 ?n3 - node))

(:action request-object
:parameters (?u - node)
:precondition
  (and (not-request-send ?u))
:effect
  (and
     (not (not-request-send ?u))
     (find-pending ?u ?u ?u)))

(:action accept-request
:parameters (?u ?v ?w ?z - node)
:precondition
   (and
     (link ?u ?z)
     (not (= ?u ?z))
     (find-pending ?u ?w ?v))
:effect
   (and
     (not (find-pending ?u ?w ?v))
     (find-pending ?z ?w ?u)
     (link ?u ?v)
     (not (link ?u ?z))))

(:action accept-request
:parameters (?u ?v ?w - node)
:precondition
   (and
     (link ?u ?u)
     (find-pending ?u ?w ?v))
:effect
   (and
     (not (find-pending ?u ?w ?v))
     (link ?u ?v)
     (not (link ?u ?u))
     (queue ?u ?w)))

(:action satisfy-request
:parameters (?u ?x - node)
:precondition
  (and
    (owner ?u)
    (queue ?u ?x))
:effect
  (and
    (move-pending ?x ?x ?u)
    (not (owner ?u))
    (not (queue ?u ?x))))

(:action receive-object
:parameters (?u ?w ?v - node)
:precondition
  (and (move-pending ?u ?w ?v))
:effect
  (and
    (not (move-pending ?u ?w ?v))
    (owner ?u)))
```

Problem Instance

```
(define (problem tree)
(:domain arrow-domain)
(:objects v0 v1 v2 v3 v4 v5 - node)
(:init
  (link v0 v0)  (link v1 v0)
  (link v2 v0)  (link v3 v0)
  (link v4 v0)  (link v5 v0)
  (owner v0))

(:goal (owner v1)))
```

Exploration of the Robustness of Plans

M. Fox and **R. Howey** and **D. Long**
Department of Computer and Information Sciences
University of Strathclyde, Glasgow, UK

Abstract

This paper considers the problem of stochastic robustness testing for plans. Although plan generation systems might be proven sound the resulting plans are valid only with respect to the abstract domain model. It is well-understood that unforseen execution-time variations, both in the effects of actions and in the times at which they occur, can result in a valid plan failing to execute correctly. Other authors have investigated the stochastic validity of plans with non-deterministic action outcomes. In this paper we focus on the uncertainty that arises as a result of inaccuracies in the measurement of time and other numeric quantities. We describe a probing strategy that produces a stochastic estimate of the robustness of a temporal plan. This strategy is based on Gupta, Henzinger and Jagadeesan's (Gupta, Henzinger, & Jagadeesan 1997) notion of the "fuzzy" robustness of traces through timed hybrid automata.

1 Introduction

Classical planning has traditionally been concerned with construction of plans as either sequences of actions, or as partially ordered sets of actions. Researchers have explored beyond the constraints of classical planning and PDDL2.1 (Fox & Long 2003) represents a formalisation of the representation of temporal planning domains, in which plans become collections of time-stamped *durative* actions. We have shown a close relationship between PDDL2.1 and models of real-time systems based on timed hybrid automata (Fox & Long 2002). An important limitation of PDDL2.1 is that it is concerned entirely with deterministic domains in which no uncertainty is captured. Others have considered the consequences of extending PDDL2.1 to allow actions to have non-deterministic effects (Younes & Littman 2004), but we are concerned with a different form of uncertainty. In this paper we argue that there is an important reason to relax certainty about precise execution times of actions (as others have also argued — for example, see (Muscettola 1994)) even if one adopts a deterministic model of actions. We then proceed to explore the relationship between temporal uncertainty and work in timed hybrid automata (Gupta, Henzinger, & Jagadeesan 1997). We discuss the work we have done in extending our plan validation system to handle plan validation in the face of temporal uncertainty, including the implications of temporal uncertainty on plan correctness. We conclude with a discussion of the further problems of managing metric uncertainty and our progress in handling them.

2 Temporal Uncertainty in Planning

Although the introduction of metric time into planning makes it possible to represent and reason about far more realistic domains than with classical planning models, it introduces new problems in the relationship between planning and execution. Unlike the classical model, in which time is measured only in a relative sense, in the ordering of actions, once one has metric time, with actions assigned precise execution times, it is possible for the correctness of a plan to rely on the precise synchronisation of actions as they are performed by the executive. This is unreasonable, since no executive, even under the control of highly accurate microcontrollers, can achieve arbitrary levels of accuracy in the synchronisation of actions. This problem is only compounded when one considers that in translating plans into actions, it is inevitable that the abstractions in the domain model will fail to match precisely the reality of the world.

In the planning literature, this problem has been handled by the introduction of *temporal flexibility* in which intervals of uncertainty surround times of execution (Muscettola 1994; Vidal & Ghallab 1996). This is an attractive solution, although there has been some ambiguity about the precise semantics of the intervals: it is not always clear whether the interval indicates freedom in the choice of an executive of precisely when to execute an action or whether it indicates uncontrollable uncertainty about precisely when an action will execute. This matters a great deal, since the former intervals may be subjected to constraints to reduce their size, while the latter are presumably outside the control of the executive. Determining the dynamic controllability of a set of temporal constraints has been explored and efficient algorithms have been proposed (Morris, Muscettola, & Vidal 2001).

3 Robust Automata

In (Gupta, Henzinger, & Jagadeesan 1997), Gupta *et al.* also identify the difficulties that arise when trajectories through hybrid automata are interpreted as defining the timing of events with arbitrary precision. Again, the problem that is discussed is that physical interpretations of the execution of

the trajectories rely on executives that cannot meet the demand for arbitrary precision. The authors observe that a trajectory in a hybrid automaton can be technically valid, according to the formal definition of validity of a trace, but can pass arbitrarily close to trajectories that are *in*valid. In such situations, the theoretical validity of the trace is of little practical value if a physical system cannot achieve the precision of execution that would avoid the invalid trajectories. The solution to the problem proposed by the authors is to identify *robust traces*. A trajectory defines a robust trace, τ, through a hybrid automaton if there is a dense subset of the trajectories lying within some open tube around τ that contains only acceptable traces. The authors define various alternative metrics that can be used in determining the open tube around a trajectory and also indicate that others could be considered. Amongst these is the metric defining the distance between two traces to be the maximum of the distances between pairs of corresponding events in the two traces. We will call this metric the *max*-metric.

Although the definition of robust acceptance is a useful and intuitively appealing one, the authors do not offer any proposals for how a trajectory might be tested for this property in practice. The work described here proposes a practical strategy for the stochastic determination of plan validity based on the theoretical foundations established by Gupta *et al* (Gupta, Henzinger, & Jagadeesan 1997). In considering robustness on a stochastic basic, we are forced to consider the distribution of the trajectories that might be pursued, around the original planned trajectory. This is in contrast to the work of Gupta *et al*, which, by requiring that a dense subset of trajectories should be valid, is unconcerned with how unlikely are the possible failing trajectories around the original planned trajectory.

4 Robust Plan Validation

We have developed a system based on our plan validation tool, VAL (Howey, Long, & Fox 2004), which allows us to test the robustness of plans. The approach we adopt is to *probe* the plan space in the tube around the plan to be tested, using Gupta *et al.*'s *max*-metric to determine the tube we sample. The samples are identified by introducing random perturbations into the timings of the execution points of individual actions. We call this *juddering* the plan. Each such perturbation determines a new plan that can be tested using the precise deterministic testing implemented in VAL. We perform a large number of tests (a configurable value, defaulting to 1000) and then measure the proportion of successful plans. In order for the plan to be robust in an analogous way to the robust trajectories of Gupta *et al.*, the successful plans in the plan space we probe should form a dense subset. This cannot be tested empirically, so instead we report the proportion under the assumption that a plan can be considered robust if a sufficiently high percentage of the plans in the tube are valid. Although we use the *max*-metric to define the tube in which we sample, the samples are selected by applying an approximately normal distribution in generating perturbations of the times of the actions. This use of probing plan space has also been adapted to support planning under uncertainty in the work of Younes (Younes 2004). In that work, the probing allows exploration of the space generated by non-deterministic effects of actions, rather than of the space of plans in the tube around a specific plan, so the author explores a rather different direction to the one explored here.

There are some interesting questions raised in applying the probing strategy we have described. The time points that are relevant to a plan include both the times of execution of actions and also the times at which durative actions complete execution. In plans for domains that include exogenous events (as defined in PDDL+ (Fox & Long 2002)), the timing of events and the timing of actions could both be perturbed. However, we consider that the perturbations represent the inability of an executive to apply arbitrary precision in determining when to execute actions. In contrast, events model reaction of the world to the actions of the executive, and their timing is not subject to the constraints of physical limitations of an executive. For example, the event of a ball bouncing, after the executive executes the action of releasing it, will occur at a certain time after the release action without any need for a conscious reaction. One might argue that there will be slight variations in the time of flight of the ball, caused by slight variations in the air pressure, in the level of the surface the ball strikes and so on. We consider that these fluctuations are at orders of magnitude less than the accuracy of timing for most feasible executives, so they can be ignored.

5 Varying the Timestamps of Actions

In the family of languages based on PDDL the representation of a plan is as a list of timestamped actions. However when the plan is executed in a real world situation it is unlikely that the actions within it will be executed at *exactly* these times. Therefore we consider the possibility that these timestamps are not fixed when validating the plan, and use our probing strategy to investigate by how much the timestamps may be displaced. When a juddered plan is executed each action is executed at a time that is slightly different from the time in the original plan. Juddering ensures that, on each execution of the plan, the times of the actions will be (independently) different and we can identify the robustness of the original plan with respect to the times at which the actions are specified to occur. This approach introduces just enough temporal flexibility into the plan to guarantee a desired level of confidence in its robustness.

For each action, a, at time t_a, we execute the action in the interval $[t_a - \delta, t_a + \delta]$ for some $\delta > 0$. The chosen times of execution are random and follow a normal distribution about t_a. The exact nature of how the action timestamps are chosen in this interval is independent of the investigation of plan robustness. In our initial experiments into plan robustness we have used both uniformly distributed times and approximately normally distributed times within the intervals.

If a plan is not robust then it would be very useful to know where the plan is most likely to fail. This is also a consideration we are investigating. When a plan is not robust VAL reports where and when a plan is failing. See section 7 for some examples.

5.1 ε Separation and Robust Plans

Previously, as defined in PDDL2.1, see (Fox & Long 2003), it was required that actions must be separated by a minimum value, namely ε, or the *tolerance value*. This was a solution to the problem of actions being so close together that the executive of the plan may not be reliable enough to ensure that these actions are executed in the correct order. Certain orderings may invalidate the plan, so ensuring that the end points of interfering actions do not coincide is very important. The solution adopted in the semantics of PDDL2.1 was the following: if two actions are within this tolerance value then they are assumed to be executed at the same time. To check that this results in a valid plan the actions are then checked, using VAL, to ensure that they are pairwise non-mutex at the coinciding end points. However there is a slight difficulty: suppose that three actions are timestamped t_1, t_2 and t_3 such that $t_1 < t_2 < t_3$, $t_2 - t_1 < \varepsilon$, $t_3 - t_2 < \varepsilon$ and $t_3 - t_1 > \varepsilon$ then it is unclear how to handle the interactions between the actions. Currently in VAL the first two actions are executed together and the third action escapes any mutex checks with the second action, which is clearly unsatisfactory.

With the newly proposed approach of executing many plans with varied timestamps there is no need consider any such ε separation. When the timestamps of coinciding actions are juddered it can be determined whether the possible reordering of actions that occurs as a consequence invalidates the plan or not. If juddering the actions invalidates the plan then the separation between them should be increased. The size of the gap between actions will depend on how robust the plan is required to be.

The ε separation approach for ensuring the robustness of a plan is inadequate when a plan contains *continuous effects*. Suppose we have a plan where all actions are separated by at least ε, so that when the plan is executed the actions cannot switch their order of execution. This does not guarantee the robustness of the plan. On executing the plan the time at which each action executes may differ by up to $\frac{\varepsilon}{2}$. These small changes may in turn affect the continuous effects, which may be very sensitive to the times at which actions are executed, since their effects may interact with one another. The change in values of continuous effects changes the values of PNEs for given times which may, of course, invalidate preconditions, invariants and the plan itself. The new approach of varying the timestamps of actions takes this complication into account. In fact, since continuous effects may be arbitrarily complex this is the only feasible way to ensure plans with continuous effects are robust.

5.2 Mutex Conditions and Robust Plans

The semantics of PDDL2.1(Fox & Long 2003) relies on mutex-checking to ensure that two actions that are executed at the same time are non-interfering so that the order of in which actions are actually applied does not change the outcome. However, with our approach of executing many plans with varied timestamps we are, indeed, checking that the order of execution of actions that are close to one another does not change the outcome of the plan. This has the same effect as checking that coinciding action end points are non-mutex. In fact, mutex conditions are effectively rendered redundant when we vary the timestamps. The timestamps of actions are varied to within certain bounds and the chances of two actions occurring at exactly the same time is very remote.

The strong mutex constraint in PDDL2.1 *guarantees* that there can be no order in which actions at a single time point might be executed in practice and interfere with one another. It is in order to support a guarantee of correctness that the mutex condition is so strict. In the sampling approach we consider here we cannot offer a guarantee of plan correctness, but only a stochastic assessment, which can, of course, be made arbitrarily close to certainty. This is significant, because, for example, it is possible to construct a family of planning domains in which a set of n actions may be executed at the same time point and generate the same resulting state in every possible sequential execution of the actions but one. This means that there would be only one chance in $n!$ of the actions producing a failure, and this might well be an acceptable risk for sufficiently high n, even though the plan would be rejected by VAL under the mutex rules of PDDL2.1 because it is not guaranteed to execute successfully.

6 Statistical Analysis

Our goal is to judder the times associated with a plan and check the validity of the resulting juddered plan as often as is necessary to give an acceptable level of confidence in the robustness of the original plan. If the plan juddered and executed one thousand times we can claim to have strong evidence for the extent of its robustness. In the following we report the results we have obtained from a *binomial experiment* investigating the robustness of plans. A binomial experiment satisfies the four following conditions:

1. There must be a fixed number of trials.
2. Trials must be independent (one trial's outcome cannot affect the probabilities of other trials).
3. All outcomes of trials must be in one of two categories.
4. Probabilities must remain constant for each trial.

The number of times that a plan will succeed out of N runs of the experiment (each run consisting of juddering the timestamps and executing the plan), is given by a *binomial distribution*, denoted by $B(N, p)$, where p is the probability of a plan succeeding. The value of p is unknown, and we wish to determine its value. It is not possible to calculate this value precisely, but we can calculate it to within certain limits. Firstly we calculate a *confidence interval*, calculated using the following formula, for the number of valid plans obtained from N runs of the experiment.

$$\bar{x} \pm \frac{t_{(\frac{\alpha}{2}, N-1)} s}{\sqrt{N}}.$$

The mean of the sample is denoted by $\bar{x}$, in this case the number of valid plans, a valid plan counts 1 and an invalid plan counts 0. The value s is the standard deviation of the population. This is unknown, but (due to the central limit theorem) the sample standard deviation may be used given that $N > 30$. Finally $t_{(\frac{\alpha}{2}, N-1)}$ is the upper critical value of the t student distribution with $N - 1$ degrees of freedom (a

number to be retrieved from a table). We set α equal to 0.05 so that the level of confidence is 95% which is considered to be significant. There is a 95% chance of the mean lying in this interval. The more often the experiment is run the smaller the size of the interval. Simply dividing the confidence interval by N gives a confidence interval for the value of p. Sometimes a value of $p < 1$ may be acceptable: for example, if the plan has high rewards. However, most often we will be looking for plans that never fail. For this case we can perform a different statistical test.

We want to know how sure we can be that the plan is robust if it always executes successfully when the timestamps are juddered. We can perform a hypothesis test to determine this. Suppose that we have successfully executed N juddered plans. We wish to be 99% certain that the plan will be valid with a probability greater than 99%. If that the probability of executing a plan successfully is less than or equal to 99% then the probability of the result being a fluke is at most 0.99^N. To be 99% certain that the result is not a fluke we need this value to be less than 0.01. Therefore we require, $0.99^N < 0.01$, which implies that $N \geq 459$. Similarly it can be shown that to be 99% certain that the plan will be valid with a probability of at least 95% then $N \geq 90$. Also, to be 95% certain of a valid execution with probability of at least 99% and 95%, we require $N \geq 299$ and $N \geq 59$ respectively. When VAL executes N plans with their timestamps juddered and all are valid then it reports that you can be 99% certain that the plan will have a valid execution with probability of at least some percentage. For example 1000 successful runs grants 99% certainty that the plan will execute with a probability of at least 99.77%.

6.1 Calculating the Robustness of a Plan

So far we have only considered juddering action timestamps by a random amount no larger than some bound, and how likely the plan is to succeed in these circumstances. It would be useful to know, for a given plan, what is the maximum possible judder that results in valid plan execution every time. Let v be the judder value. The largest value for v can be calculated by searching amongst its possible values. For each given value of v we check that the varied plans will always be valid with a probability of at least 95% with a confidence level of 95%. This requires successful execution of 59 plans. In this way the value of v is calculated to within a small interval, see section 7 for some examples.

7 Examples

7.1 Thermostat

Consider the temperature of a machine that is controlled by a thermostat which fluctuates over time as given in figure 1. The temperature is modelled using events and processes as specified in the description of PDDL+ (Fox & Long 2002). the details of how these are modelled in PDDL+ are omitted as it is not relevant to the current discussion. It should be noted that the juddering of action timestamps does not vary the temperature model in any way. We define the problem so that a valid plan must place a discrete action at every local maximum and minimum of the temperature curve within

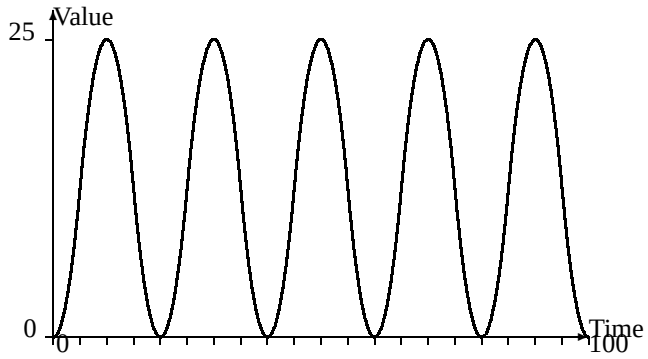


Figure 1: Graph of (temp unit).

a limited time. This is achieved by forcing these actions to be executed for values above or below certain temperatures, and by ensuring that the actions must alternate between maxima and minima. The best plan to solve this problem is given below:

Time	**Action**
10:	(upper unit)
20:	(lower unit)
30:	(upper unit)
40:	(lower unit)
50:	(upper unit)
60:	(lower unit)
70:	(upper unit)
80:	(lower unit)
90:	(upper unit)

Firstly, suppose we wish to test how the plan performs when the action timestamps can vary by up to 4 time units. When VAL executes 100 randomly altered plans the following results are reported:

- 12 plans are valid from 100 plans for each action timestamp ± 4.
- There is a 95% chance that the plan has a valid execution with probability in the range 12 ± 6.44724.

The plan failures are reported as follows:

Failures	**Time**	**Action**
22	**10:**	(upper unit)
19	**20:**	(lower unit)
6	**30:**	(upper unit)
7	**40:**	(lower unit)
12	**50:**	(upper unit)
5	**60:**	(lower unit)
13	**70:**	(upper unit)
2	**80:**	(lower unit)
2	**90:**	(upper unit)

As the results show, the plan is not highly robust. Each action has the same probability of failure as the other actions.

However, when a plan has failed at some point the execution stops, so the plan failures listed show the first point at which the plan fails. As a consequence the actions later in the plan are less likely to invalidate the plan because they depend on the other actions not failing first. Figure 2 shows the percentage of plans that invalidate the plan at certain times. (These points are joined by lines.) Figure 3 shows the cumulative percentage of plans that fail at certain times. The confidence interval is quite large, so to reduce the size we can perform the same test again but this time with 1000 varied plans.

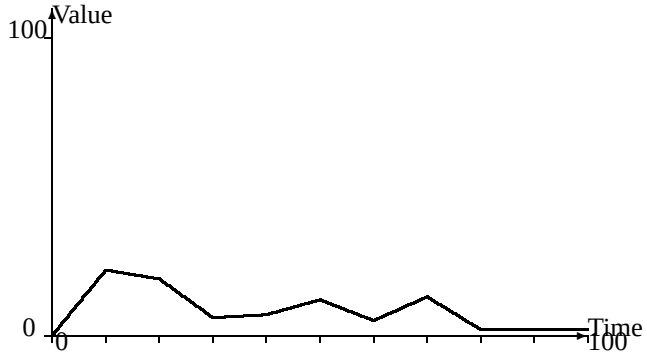


Figure 2: Percentage of plans failing at different times for 100 plans.

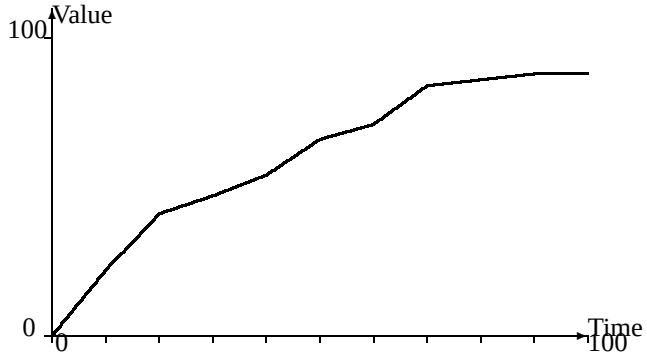


Figure 3: Cumulative percentage of plans failing by different times for 100 plans.

- 122 plans are valid from 1000 plans for each action timestamp ± 4.
- There is a 95% chance that the plan has a valid execution with probability in the range 12.2 ± 2.03061.

Because of the larger sample size we can be more confident that the probability of success is about 12.2%. The graphs showing when the plans fail, figures 4 and 5, show a smoother appearance as we would expect. The probability of a given action failing is $(1-p)^n p$, where p is the probability of one of the actions failing and n is the number of actions before the action in question. The graphs confirm that the plans fail following these probabilities. In more complex plans the probability of each action failing will be different and their interaction with other actions will need to be taken into account. For any plan an action can only invalidate a plan if the preceding plan has been successful, which will have a given probability.

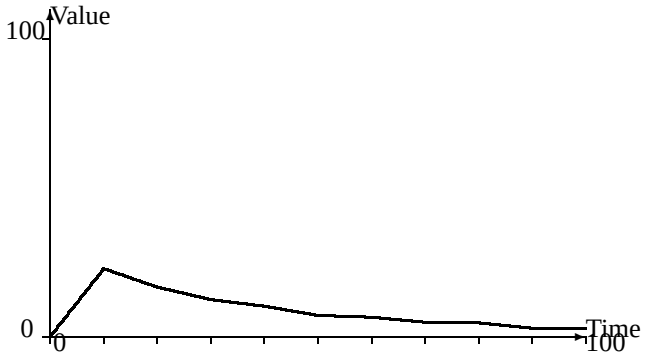


Figure 4: Percentage of plans failing at different times for 1000 plans.

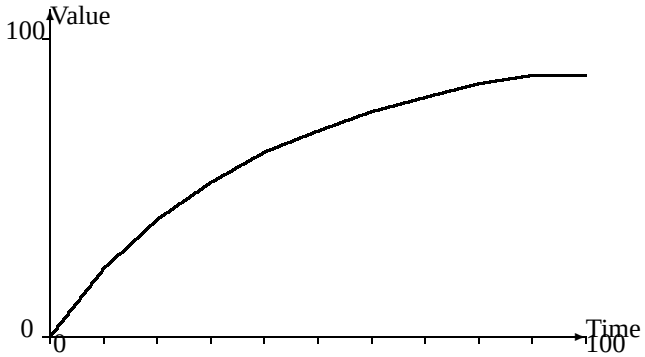


Figure 5: Cumulative percentage of plans failing by different times for 1000 plans.

If we use VAL to calculate how robust this plan is we get the following report:

- The plan has a robustness in the range 3.15918 ± 0.00488281.

This shows that provided that the actions do not vary by more than 3.154 (taking the most conservative bound) then the plan will be execute successfully. This value can be considered as the robustness measurement of the plan. For this example we can, in fact, calculate its robustness exactly, giving $\sqrt{10} = 3.162277...$, which is in the range calculated by VAL. In general it is not possible or feasible to calculate the robustness measurement of a plan exactly. However, using VAL, it is easy to calculate this measurement to within a small interval.

7.2 The Generator

As another example of calculating the robustness of a plan consider the generator example. Suppose that a generator must run continuously for 100 time units. In order to achieve this it must be refuelled whilst it is generating using two tanks of fuel. Refuelling starts quickly, but slows down to a trickle as the tank empties. If refuelling is initiated too early then the generator fuel tank will overflow. If it is initiated too late the tank will run dry. Therefore we need we refuel the generator somewhere near the mid point of the generating activity. However, the two refuelling actions must not be too close as they cannot overlap. The graph in figure 6 shows the fuel level of the generator in a robust plan for this problem. VAL reports:

- The plan has a robustness in the range 6.26465 ± 0.00488281.

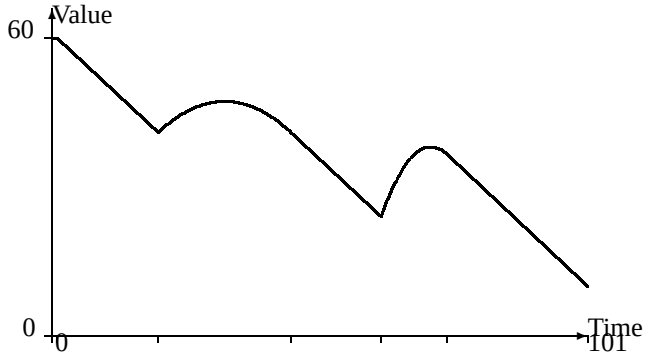


Figure 6: Graph of (fuel-level generator).

7.3 Robustness to Duration Variation

As well as juddering the start point of an action we can judder the action duration. This reflects the fact that actions sometimes take slightly less or more time than expected. However, the impact of this is that end points of actions can be displaced by up to twice the judder value. This can have significant impact on plan validity as illustrated in the following example.

Plans generated in the IPC3 competition, using the ε tolerance value, are often not robust to variations in the action durations because they have been constructed to be as tightly packed as possible with respect to ε. As an example we consider a plan from the 2002 IPC produced by LPG for the zeno travel domain with time and numerics. VAL calculates the robustness of this plan as $0.000457764 \pm 0.000152588$. The plan was calculated using $\varepsilon = 0.001$, which is the minimum distance by which interfering actions should be separated. Therefore we would expect the plan to have a robustness of at least 0.0005 (since two actions may move toward one another). However, testing the plan for a variation of ± 0.0004 on 1000 plans yields the result that there is a 95% chance that the plan has a valid execution with probability in the range 98.6 ± 0.728956. This loss of robustness is directly due to the double judder phenomenon described above.

Now suppose that we wish to use this plan with a variation of ± 0.001. The robustness measure is smaller than this so we do not expect the plan to always work. We wish to identify how likely the plan is to succeed and where the plan is most likely to be invalidated. If we test the plan with this variation on 1000 runs then VAL reports that 'there is a 95% chance that the plan has a valid execution with probability in the range 43.1 ± 3.07251.' The plan failures are reported as below:

Failures	Time	Action
157	**0.002:**	(board person1 plane1 city0) [0.3]
0	**0.303:**	(fly plane1 city0 city1) [4.870]
164	**5.174:**	(board person3 plane1 city1) [0.3]
10	**5.175:**	(debark person1 plane1 city1) [0.6]
115	**7.196:**	(fly plane1 city1 city0) [4.87]
123	**12.067:**	(debark person3 plane1 city0) [0.6]

Figure 7 shows a graph produced by VAL of the number of actions that fail at certain times. There is also a list of why each action failed, together with sample plan repair advice. The plan repair advice is for only one failed instance, since in general when numerical values are involved every single failure could be unique. For example the advice for the first action is:

1. 157 failures for **0.002:** (board person1 plane1 city0) [0.3]

   (a) 157 failures: The invariant condition is unsatisfied. Sample plan repair advice:

      i. *Invariant for* (board person1 plane1 city0) has its condition unsatisfied between times 0.302713 and 0.3029. The condition is satisfied on the empty set. Set (at plane1 city0) to true.

Failure of the execution of a durative action can be caused by violation of its invariant condition or failure to satisfy its precondition. This action has failed because its invariant condition has been violated. Looking at the plan it can be seen that the *(fly plane1 city0 city1)* action starts almost exactly when the boarding operation has finished. It is clear that the *board* action fails because, as a consequence of juddering, the plane has taken off before the passenger has finished boarding.

8 Robustness with Metric Fluents

Any plan that is intended to interact with physical processes will be subject to other sources of uncertainty than simply the times at which actions are executed. In particular, processes generate continuous change in the world that will only ever be modelled at some level of abstraction. Thus, a bath filling with water that is flowing at a constant rate can be modelled as having a volume that increases linearly. This model abstracts phenomena such as small quantities of water splashing out of the bath, minor fluctuations in the rate of flow due to unpredictable and uncontrollable additional demands on the water supply and so on. Figure 8 illustrates

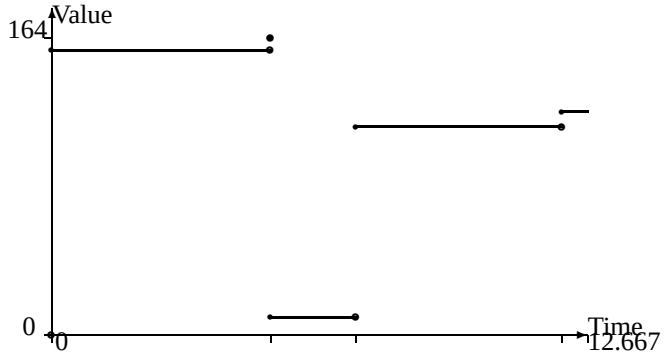


Figure 7: Number of plans failing at these times.

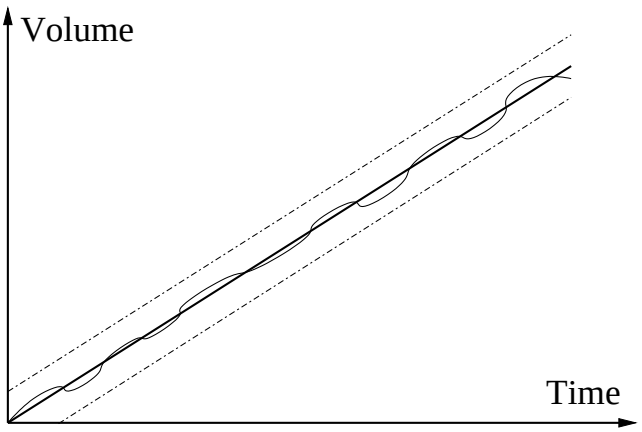


Figure 8: Graph showing water flowing into a bath.

how the volume of a bath might fluctuate from its linearly increasing estimate, as suggested by the fluctuating curve. In using the model to predict the volume of the water in the bath it would be accepted that the predicted volume would not exactly match reality to arbitrary levels of precision (nor could the real volume even be measured to arbitrary precision in order to compare it with the model). The implication of this for robust planning is that no plan can be considered robust if its correctness depends on the values predicted by its models being accurate to arbitrary degrees of accuracy. Thus, just as the times of execution of actions should be expected to judder, so also should measurements of metric fluents evolving under the influence of continuous processes.

We distinguish values that are influenced by continuous processes from values that are affected by discrete change alone. Where values increase or decrease by discrete quantities then the abstraction of the quantity into these discrete units is sufficient to ensure that the uncertainty in execution can be eliminated. Essentially, the uncertainty about the execution of actions that depend on these values is abstracted into the question of how accurately the discrete units can be measured and how appropriate these units are for the execution of actions that consume them. We may assume that continuous processes are only modelled explicitly in domains where there is a potentially significant sensitivity to threshold values and it is precisely in these cases that we want our plans to be robust to minor fluctuations in the physical processes that drive them.

8.1 Change, Chaos and the Butterfly Effect

In some cases, as is well known, small changes in initial conditions can lead to dramatically different evolutions of a physical system. These systems are often said to exhibit chaotic or highly non-linear behaviour. The so called "butterfly effect" is apparent in a wide range of physical phenomena. It is readily apparent that plans are extremely unlikely to be able to interact with metric fluents with this kind of behaviour in any way that is highly sensitive to the actual values of the fluents. For this reason, it will make more sense to model systems with these behaviours as abstractions that can only be managed at a coarse level. For example, we know that weather patterns have this kind of chaotic behaviour and it is therefore not reasonable to construct plans that depend on predicting precise temperatures, cloud cover or precipitation at precise times of day. Instead, we can manage abstractions that use ranges of temperatures across intervals of time, so that we can, for instance, plan what clothes to take on holiday.

If we assume that our planning models do not contain explicit models of physical processes that are highly non-linear or chaotic, then we can simplify our management of the uncertainty that can arise in handling the metric fluents that are affected by the processes. In particular, we can assume that, over time, the model is an accurate prediction of the evolution of a process, subject only to a local fluctuation in the value measured at any given instant.

8.2 Robust Plans with Metric Uncertainty

To test the robustness of plans to uncertainty caused by fluctuations in the behaviours of physical processes we consider only metric fluents that are subject to continuous change at points where they occur in comparison conditions. Wherever such comparisons are made as preconditions for execution of actions we apply a small judder to the value of the appropriate metric fluents before checking the condition. This process is no more complicated in the case of invariants, since the judder is treated as a constant shift in the curve governing the process for the purposes of testing the invariant across its appropriate interval. We do not propagate the effects of the judder into the use of the corresponding metric fluents for updating values in the effects of actions, which is the consequence of our assumption that all processes are sufficiently accurately modelled and sufficiently predictable to be adequately handled by the model. We also do not use judder to adjust the preconditions of events. This decision is based on the view that events represent consequences of changing processes in the world and there is no imperfection in the reaction of the world to those consequences. Of course, in some models events might be intended to represent the reactions of external agents to processes initiated by the planning agent and, in that case, it might be appropriate to apply a judder to those reactions. The question of precisely what is an appropriate way to handle events and whether to handle some events differently, remains an area for future work.

One implication of this approach is that any preconditions that require *strict equality* tests between continuously changing metric fluents and some other values will fail. We consider this to be realistic: it will only ever be possible to achieve strict equality at the level of abstraction used in measuring discrete units. Otherwise the best that can be achieved is to obtain a value lying within a particular interval.

Even though we do not judder the effects of metric updates, simply juddering the times at which actions occur is sufficient to have an impact on the behaviour of metric update processes. It is possible for these effects to be non-linear and for their propagation into the plan to have dramatic consequences for the execution. This is an aspect of considerable interest and knowing where a plan is vulnerable to non-linear effects caused by apparently minor changes in the structure of a plan would be of value in determining whether a plan is useful and how to protect it during execution.

9 Future Work

In future work we intend to address both an increased level of non-determinism in the metric components of the plan and the integration of our approach with the probing strategy of Younes (Younes 2004) which considers non-deterministic outcomes of actions. In terms of extending the level of metric non-determinism that we consider, we wish to address the fact that uncertainty about the time of execution of specific actions and the uncertainty in the processes that govern metric fluents are not uniform. Our current treatment assumes that they are. Our current model for introducing judder into the behaviour of durative actions assumes that the time of the final point is governed by when the action starts and our ability to measure the accuracy of its duration. In some cases the duration will be governed by a process so that a better model of the uncertainty would be to judder the end point independently of the start point.

We currently judder the plan and then validate the resulting plan, so that events are triggered according to the times at which the juddered actions occur. In cases where events are triggered by metric fluents crossing critical thresholds under the influence of continuous processes our current approach will judder the value of the metric fluents leading to a corresponding impact on the timing of events. In some cases this can lead to significant changes in the behaviour of the plan and even apparently chaotic outcomes. We are still considering how best to deal with this.

10 Conclusion

We have presented a stochastic strategy for determining the robustness of temporal plans to the possible variations in the timings of actions at execution. We have proposed a probing strategy which uses a juddering mechanism to sample plans within a *tube* around the original plan. The width of the tube is determined by the judder value. Using this strategy we can determine whether a plan is robust to the given judder value, and we can also determine the judder value that gives robustness to a required confidence level. We consider temporal and metric constraints to constitute a form of non-determinism because of the inaccuracy inherent in measuring properties of the physical world. We want to increase the amount of non-determinism that can be handled by our approach and to integrate our strategy with those that consider non-deterministic outcomes of actions.

References

Fox, M., and Long, D. 2002. PDDL+ : Planning with time and metric resources. Technical report, University of Strathclyde, UK. Available at: `planning.cis.ac.uk/competition/`.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of AI Research* 20.

Gupta, V.; Henzinger, T.; and Jagadeesan, R. 1997. Robust timed automata. In *HART'97: Hybrid and Real-time Systems, LNCS 1201*, 331–345. Springer-Verlag.

Howey, R.; Long, D.; and Fox, M. 2004. VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *Proceedings of 16th IEEE International Conference on Tools with Artificial Intelligence*.

Morris, P.; Muscettola, N.; and Vidal, T. 2001. Dynamic control of plans with temporal uncertainty. In *Proceedings of IJCAI*, 494–502.

Muscettola, N. 1994. HSTS: Integrating planning and scheduling. In Zweben, M., and Fox, M., eds., *Intelligent Scheduling*. San Mateo, CA: Morgan Kaufmann. 169–212.

Vidal, T., and Ghallab, M. 1996. Constraint-based temporal management in planning: the IxTeT approach. In *Proc. of 12th European Conference on AI*.

Younes, H., and Littman, M. 2004. PPDDL1.0: An extension to PDDL for expressing planning domains with probabilistic effects. Technical report, `www.cs.cmu.edu/~lorens/papers/ppddl.pdf`.

Younes, H. 2004. Planning and verification for stochastic processes with asynchronous events. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, 10011002.

Lifecycle Verification of the NASA Ames K9 Rover Executive

Dimitra Giannakopoulou[1, 3] **Corina S. Pasareanu**[2, 3] **Michael Lowry**[3] **and Rich Washington**[4]

(1) USRA/RIACS
(2) Kestrel Technology LLC
(3) NASA Ames Research Center, Moffett Field, CA 94035-1000, USA
{dimitra, pcorina, lowry}@email.arc.nasa.gov
(4) Google Inc., 1600 Amphitheatre Parkway, Mountain View, CA 94043, USA
rwashington@google.com

Abstract

Autonomy software enables complex, robust behavior in reaction to external stimuli without human intervention. It is typically based on planning and execution technology. Extensive verification is a pre-requisite for autonomy technology to be adopted in high-risk domains. This verification is challenging precisely because of the multitude of behaviors enabled by autonomy technology.

This paper describes the application of advanced verification techniques for the analysis of the Executive subsystem of the NASA Ames K9 Rover. Existing verification tools were extended in order to handle a system the size of the Executive. A divide and conquer approach was critical for scaling. Moreover, verification was performed in close collaboration with the system developers, and was applied during both design and implementation. Our study demonstrates that advanced verification techniques are useful for real-world planning and execution systems. Moreover, it shows that when verification proceeds hand-in-hand with software development throughout the lifecycle, it can significantly improve the design decisions and the quality of the resulting plan execution system.

Introduction

Verification is essential for planning and execution technology to be adopted in high-risk domains. This paper is a demonstration of how advanced verification techniques were used on a plan execution system in the domain of Mars rovers.

The work presented here has been performed as part of a project at NASA Ames. The objective of the project is to develop and demonstrate the use of advanced verification techniques for detecting integration problems in the design and implementation of NASA autonomy software. Traditional testing is hard for autonomous systems due to high complexity and unpredictable environments. Moreover, integration problems are difficult to detect, and are typically checked during integration testing, i.e. *after* the entire system has been implemented. At that stage, fixing such problems may require significant time and effort since they may involve major changes in the architecture of the system, and possible re-implementation of a large part of it. Therefore, we believe that the verification of a safety critical system should be addressed *as early as possible during its design*, and should go hand-in-hand with later phases of software development.

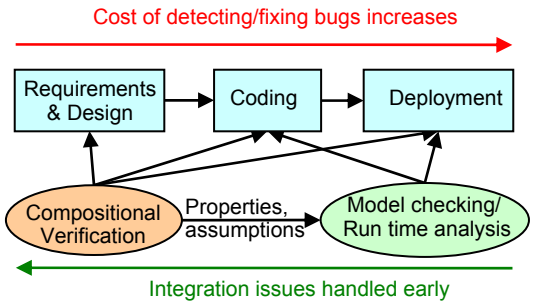


Figure 1. Compositional verification throughout the software lifecycle

Our work advocates the use of a combination of formal analysis techniques (i.e. model checking [11]) and testing to analyze autonomous systems throughout their lifecycle. The size of such systems is beyond the capabilities of existing (formal) verification technologies. Moreover, due to the combinatorics of the possible behavior paths, testing alone cannot provide the desired degree of confidence. To address these issues, our work has the following goals (see Figure 1):

- Apply, extend, and integrate verification tools at different phases of software development, i.e. at design and implementation phases of the software lifecycle.
- Use divide and conquer techniques that decompose the verification of a software system into manageable verification of its components, to achieve scalability in software verification. The verification of the components can then be composed to verify the entire system, hence the name 'compositional verification'.

• Use design level artifacts to subsequently guide the implementation of the system and to enable more efficient verification at the source-code level.

The main contribution of the work discussed here is the development of *compositional* verification and validation techniques for autonomy software, and their integration with other verification techniques for an integrated lifecycle approach to verification. This approach was applied to a significant autonomy software system: the Executive of the K9 experimental Martian Rover developed at NASA Ames, a concurrent software system of 35 K lines of C++ code.

The verification effort was performed in close collaboration with the designers and developers of the system throughout its lifecycle, and consisted of the following steps:

• **Design level modeling**. Detailed design level models were created. The models describe the overall concurrent architecture of the executive (coordinating and monitoring components) and advanced features that allow for increased autonomy (e.g. alternate plan execution, support for concurrent activities, separation of start and end constraints). A comprehensive set of requirements were also created (both English and formal descriptions). The requirements capture key concurrency and plan execution properties. We believe that both the models and the requirements could be successfully re-used for the design and analysis of future advanced executives.

• **Design level analysis**. Model checking techniques were used for the *exhaustive verification* of design models against requirements. We developed automated compositional reasoning techniques to increase the scalability of model checking. These techniques were applied to the analysis of the design models, achieving a 10x improvement – in terms of time and memory consumed - over monolithic (non-compositional) model checking.

• **Code level analysis**. Although design level verification is important, subsequent code-level verification is needed to guarantee that the implemented system indeed satisfies the properties. To this aim, the design level artifacts were used to perform compositional verification of the actual source code, as advocated in [4]. For code-level analysis of individual components, we applied software model checking, where we obtained a 3x improvement in terms of consumed memory. We also investigated automated testing technologies (run-time verification, to monitor the execution of the system, and automated test input generation, to systematically generate test inputs up to a given size).

As a result of design level analysis, we discovered several integration problems. Based on these results, the developer *changed the design* of the Executive, resulting in a simplified architecture with increased modularity. We analyzed both versions of the Executive. While for the first version, we created the models after coding (partly by reverse engineering), for the second version, we created the design models before coding. During this process, we re-used component models from the previous version.

This experiment convinced developers that there is considerable benefit in using verification techniques at the design level. Models were used to quickly experiment with design decisions. Moreover, several integration issues were identified and corrected. It was acknowledged that the later in the lifecycle design errors are identified, the more costly it is to fix them, especially if such errors require major design changes in the system. Our techniques are directly applicable to the analysis of other complex autonomous systems. This is particularly so for systems that make the notions of components explicit (e.g. the Mission Data Systems [7]), since our techniques take advantage of the modular architecture of the system.

Our work builds on a previous effort [15] that compared the performance of tools based on formal methods to traditional testing for the code-level analysis of the K9 Executive. Some timed aspects of the Executive were also analyzed in [19-20]. What differentiates the work presented here is 1) the integrated application of techniques *throughout* the lifecycle and 2) the development and application of novel *compositional* techniques as a way of addressing scalability issues.

The rest of the paper is organized as follows. In the next section we describe the architecture of the K9 Rover Executive and the design changes made as a result of our analysis. We then describe the compositional technologies that were used for design- and code-level verification. We follow with a discussion on the design-level modeling of the Executive; we also describe the properties that were checked and the results obtained from the lifecycle verification of the Executive. Finally, we close the paper with conclusions and some plans for future work.

K9 Rover Executive

The NASA Ames K9 Rover is an experimental platform for autonomous wheeled vehicles called rovers, targeted for the exploration of a planetary surface such as Mars. K9 is specifically used to test out new autonomy software, such as the Rover Executive [16]. Previous to the development of autonomy software, planetary rovers were controlled through sequences of detailed, low level commands uploaded from Earth. The Rover Executive provides a more flexible means of commanding the rover through the use of high-level plans, which the Executive interprets and executes in the context of the execution environment. The Executive monitors execution of

primitive actions, and performs appropriate responses and cleanup when execution fails. The Rover executive is a software prototype written in C++ by researchers at NASA Ames (approximately 35K lines of C++ code).

Plans are programs written in a language that specifies actions and constraints on the movement, experimental apparatus, and other resources of the Rover. The operational semantics of the language takes into account the possibility of failure of atomic-level command actions. The structure of a plan is a hierarchy of actions that the Rover must perform: each plan is a *node*; a node is either a *task*, corresponding to a primitive action, a (possibly concurrent) *block*, corresponding to a logical group of nodes, or a *branch*, representing a conditional branch within the plan. The plan language allows the association of each action with a number of state or temporal start, maintenance, and end conditions, which must hold before, during, and on completion of the action execution, respectively. A *continue-on-failure* flag is associated with each node. The flag being set signifies that the plan should continue execution even if the current node fails. In addition, *floating branches*, which are plan fragments triggered dynamically, may be inserted into the plan, allowing a limited form of run-time plan modification.

In contrast to programming language interpreters, the executive is expected to be robust under many plan primitive execution failures. The operational semantics for recovery from primitive failures are extensive.

Architecture of the K9 Executive

Figure 2 illustrates the architecture of the executive, prior to the design changes that the developer made partly as a result of our analysis. The executive has been implemented as a multi-threaded system, made up of a main coordinating component named *Executive*, components for monitoring the state conditions *ExecCondChecker*, and temporal conditions *ExecTimerChecker* - each further decomposed into two threads - and finally an *ActionExecution* component that is responsible for issuing the commands to the Rover. Synchronization between components is performed through mutexes and condition variables (implemented using the Posix libraries).

During the design level analysis of the executive, we discovered several concurrency problems with the inter-thread communication between different components of the executive. To eliminate these problems, the developer changed the architecture of the system, as illustrated in Figure 3. The main change is the use of an *Event Queue* as a communication mechanism between the *Executive* and the rest of the components. As a result, the communication between different components became much simpler and less prone to errors. E.g., our analysis of the first version revealed a concurrency problem (i.e. race condition) with a variable shared between the *ExecCondChecker* and the *Executive*. This shared variable was eliminated in the second version, its role being replaced by the *Event Queue*.

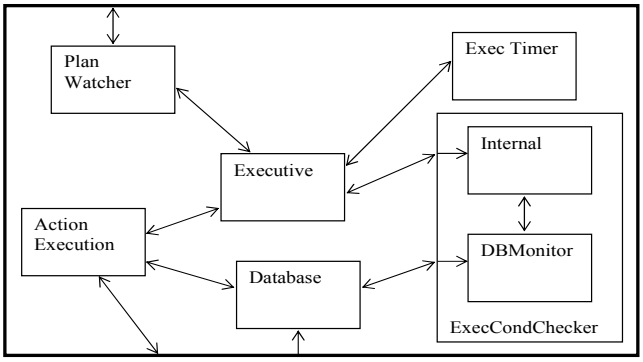


Figure 2. Original architecture of the executive

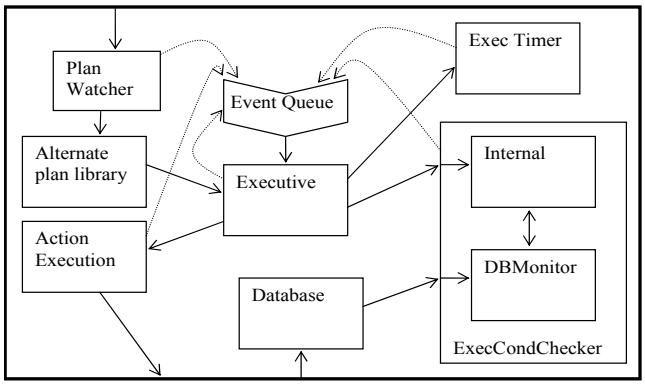


Figure 3. Updated architecture of the executive

Besides the changed architecture, the "new" executive presents several functional changes from the original one: added support for concurrent activities and "floating branches" (dynamically inserted branches and plan fragments), separation of temporal constraints from other pre- and post-conditions, and addition of relative temporal constraints to arbitrary actions within the plan. The high-level changes are summarized below:

- The Executive was changed to be event-based. An *Event Queue* was added. Both *ExecTimer* and *ExecCondChecker* were simplified to return all events, leaving the task of processing and ignoring events to the Executive. The Executive acts on an "execution context" - a data structure representing the current state of execution. This execution context was augmented to support concurrent activities and floating branches. The design documents were changed from state diagrams into event-processing loops.

- The *ActionExecution* was changed to support parallel execution threads.

- Simpler usage and removal of condition variables: there were a number of places in the first version where condition variables were used to coordinate information passing between modules (such as the *ExecCondChecker* and the *Executive*). By simplifying the information flow

(via the event queue), tight coordination is no longer necessary.

• Design simplicity over code re-use: there were a few places in the first version where code, locks, or features were re-used for conciseness. However, in some cases this made the design much more convoluted. For example, the return value of *ActionExecution* was routed through the Database and then the *ExecCondChecker* for uniformity with other conditions being checked during execution. However, this makes the information flow in the system circuitous, unclear, and error-prone.

Design-Level Verification

At design level, we use verification techniques that *exhaustively* explore all the possible executions of a system. Although exhaustive exploration is typically intractable at the code level, designs tend to be more abstract, making them more amenable to efficient verification. Specifically, we use model checking: given some formal description of a system and of a required property, model checking [11] *automatically* determines whether the property is satisfied by the system. Since scalability can also be an issue at the design-level, we enhance model checking with compositional techniques.

In this section, we describe the LTSA verification tool for design-level software analysis. We also summarize the compositional techniques [1,2] with which we extended the LTSA.

The Labeled Transition System Analyzer (LTSA)

The LTSA [8] is an automated tool that supports Compositional Reachability Analysis (CRA) [9] of a software system based on its architecture. In general, the architecture of a concurrent system has a hierarchical structure. CRA incrementally computes and abstracts the behavior of composite components based on the behavior of their immediate children in the hierarchy.

The input language FSP (Finite State Processes) of the tool is a process-algebra style notation with Labeled Transition Systems (LTS) semantics. An LTS is a finite-state machine whose transitions are labeled by *actions*, representing the internal and communication events in which a component may engage. LTSs are composed by synchronization of common actions and interleaving of local, internal actions. Safety properties are expressed as LTSs with extended semantics, and are treated as ordinary components during composition. Properties are combined with the components to which they refer. They do not interfere with system behavior, unless they are violated. In the presence of violations, the properties introduced may reduce the state space of the (sub) systems analyzed.

The LTSA framework treats components as open systems that may only satisfy some requirements in specific contexts. By composing components with their properties, it postpones analysis until the system is closed, meaning that all contextual behavior that is applicable has been provided. The LTSA tool also features graphical display of LTSs, interactive simulation and graphical animation of behavior models, all helpful aids in both design and verification of system models.

Compositional Analysis

We extended the LTSA model-checking tool with the compositional verification techniques presented in [1, 2]. Compositional verification decomposes the properties of a system into properties of its components, so that if each component satisfies its respective property, then so does the entire system. Components are thus model checked separately. It is often the case, however, that components only satisfy properties in specific contexts (also called environments). This has given rise to the assume-guarantee style of reasoning.

Assume-guarantee reasoning [12,13,14] first checks whether a component M guarantees a property P, when it is part of a system that satisfies an assumption A. Intuitively, A characterizes all contexts in which the component is expected to operate correctly. To complete the proof, it must also be shown that the remaining components in the system (M's environment) satisfy A. This style of reasoning is captured by the following assume-guarantee rule.

$$\frac{\langle A \rangle\ M_1\ \langle P \rangle \quad \text{(Premise 1)} \qquad \langle \mathit{True} \rangle\ M_2\ \langle A \rangle \quad \text{(Premise 2)}}{\langle \mathit{True} \rangle\ M_1 \,||\, M_2\ \langle P \rangle}$$

Several frameworks have been proposed to support this style of reasoning. However, their practical impact has been limited because they require extensive human input in defining assumptions that are strong enough to eliminate false violations, but that also reflect appropriately the remaining system.

In previous work, we developed several techniques that automate assume-guarantee reasoning. We implemented these techniques in the LTSA tool and used them in the analysis of the design models of the Rover Executive. We should note that our techniques are general; they rely on standard features of model checkers and could therefore easily be introduced in any model checking tool.

In [2], we present an approach to synthesizing the assumption that a component needs to make about its environment for a given property to be satisfied. The assumption produced is the weakest, that is, it restricts the environment no more and no less than is necessary for the component to satisfy the property. The automatic

generation of weakest assumptions has direct application to the assume-guarantee proof. More specifically, it removes the burden of specifying assumptions manually thus automating this type of reasoning.

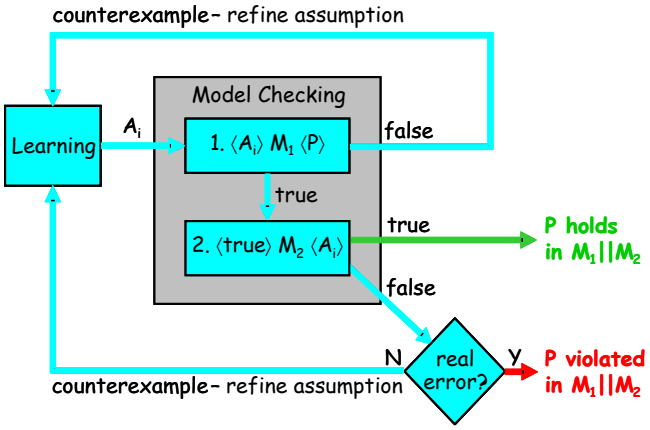


Figure 4. Framework for assume-guarantee reasoning

The technique presented in [2] does not compute partial results, meaning no assumption is obtained if the computation runs out of memory, which may happen if the state-space of the component is too large.

We address this problem in [1], where we present a model checking framework for performing assume-guarantee reasoning using the above rule in an incremental and fully automatic fashion. The framework is illustrated in Figure 4. To check that a system made up of two components M_1 and M_2 satisfies a property P, our framework automatically learns and refines assumptions Ai for component M_1 to satisfy the property, which it then tries to discharge on component M_2. The framework uses an automata learning algorithm [17] to construct the assumptions for the compositional analysis of the models.

At each iteration i, the learning algorithm is used to build an approximate assumption A_i, based on querying the system and on the results of the previous iteration. The two premises of the assume-guarantee rule are then checked. Premise 1 is checked to determine whether M_1 guarantees P in environments that satisfy A_i. If the result is false, it means that this assumption is too weak, and therefore needs to be refined with the help of the counterexample produced by checking premise 1. If premise 1 holds, premise 2 is checked to discharge A_i on M_2. If premise 2 holds, then according to the assume-guarantee rule P holds in $M_1||M_2$. If it doesn't hold, further analysis is required to identify whether A_i is indeed violated in $M_1||M_2$ or whether A_i is stronger than necessary, in which case it needs to be refined. The new assumption may of course be too weak, and therefore the entire process must be repeated. For finite state systems, this process is guaranteed to terminate. In fact, it converges to an assumption that is necessary and sufficient for the property to hold in the specific system.

A useful characteristic of our framework is that the generated assumptions are minimal; they strictly increase in size as the learning algorithm progresses, and grow no larger than the weakest assumption for M_1 to satisfy P. Moreover, in our experience, the interfaces between components are small for well designed software. Therefore, assumptions are expected to be significantly smaller than the environment that they represent in the compositional rules, and the cost of assume-guarantee reasoning will be significantly smaller than monolithic (non-modular) model checking, both in terms of time and consumed memory. Recently, we have extended our frameworks to handle more assume-guarantee rules and more than two components.

Code Level Verification

In this section, we describe our methodology [4] for using the artifacts of the design level analysis to decompose the verification of the implementations (see Figure 5).

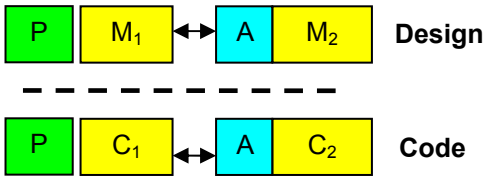


Figure 5. Using design level assumptions for source code verification

At the design level, the architecture of a system is described in terms of components and their behavioral interfaces modeled as LTSs. Design models are intended to capture the design intentions of developers, and allow early verification of key integration properties. For example, consider a system that consists of two design level components M_1 and M_2, and a property P, describing the sequence of events that the system is allowed to produce, or equivalently the bad behaviors that the system must avoid.

To check in a scalable way that the composition of M_1 and M_2 satisfies P, we use the assume-guarantee frameworks described in the previous section. We expect that, with the feedback obtained by our verification tools, the developers of the system will correct their design models until the property is achieved at the design level. At that stage, our frameworks will have automatically generated an assumption A that is strong enough for M_1 to satisfy P but weak enough to be discharged by M_2.

To then establish that the property is preserved by the implementation, our approach uses the automatically generated assumption A, to perform assume-guarantee

reasoning at the source code level. The implementation is decomposed as specified by the architecture at the design level (i.e. components C_1 and C_2 implementing M_1 and M_2, respectively), and we establish that C_1 composed with C_2 satisfies P by checking that C_1 satisfies P under assumption A, and by discharging A on C_2. If both these checks return true then the property is preserved by the implementation. Otherwise, the counterexample(s) obtained expose some incompatibility between the models and the implementations, and are used to guide the developers in correcting the implementation, the model, or both. For the actual verification of source code, we investigated two technologies: software model checking and run time verification, which are described below.

Software Model Checking

We used the Java Pathfinder software model checker (JPF) [18] developed at NASA Ames. JPF is an explicit state model checker that analyzes programs written in Java (an implementation for C++ analysis is currently being developed). JPF checks for deadlocks and assertion violations. JPF is built around a special purpose Java Virtual Machine (JVM) that allows Java programs to be analyzed. JPF supports depth-first, breadth-first and several heuristic search strategies to search systematically explore the state spaces of the analyzed programs.

Run Time Verification and Automated Test Input Generation

For the first version of the Executive we focused on checking implementations using compositional reasoning and software model checking tools. In the second version we experimented with methods that provide more scalability at the price of exhaustiveness. Specifically we investigated the use of lighter-weight analysis techniques, i.e. run time verification, for the compositional analysis of the second version of the executive.

Run time verification is an advanced testing technique that provides a means for constructing oracles that examine not just the output and interfaces of a system, but the internal computational status of the system. In run time verification, a program is instrumented to emit events which are then monitored to check for conformance to formalize requirements, either stated as temporal logic assertions, or as specialized algorithms looking for common errors, such as deadlocks and data races.

For the analysis of the Rover Executive, we used the Eagle temporal logic runtime verification framework [5]. In order to generate different executions for thorough testing, we used automated test input generation techniques to create all (non-isomorphic) input plans up to a pre-defined size [5]. Given a formal description of the inputs to a system, the test input generation techniques combine symbolic execution, model checking and heuristic search to systematically search and generate the input state space and to achieve full coverage of the input specification.

Modeling and Analysis of the Rover Executive

Initial Modeling

We produced abstract models of the Rover Executive that contained enough information – but at a higher level – to allow us to study architectural properties of the system and detect potential integration problems. The developer of the executive initially described the architecture of the system as a hierarchy of threads as illustrated in Figure 2. Moreover, he provided some design documents in his own, ad-hoc flowchart-style notation, describing the main functionality of the threads in the Rover Executive.

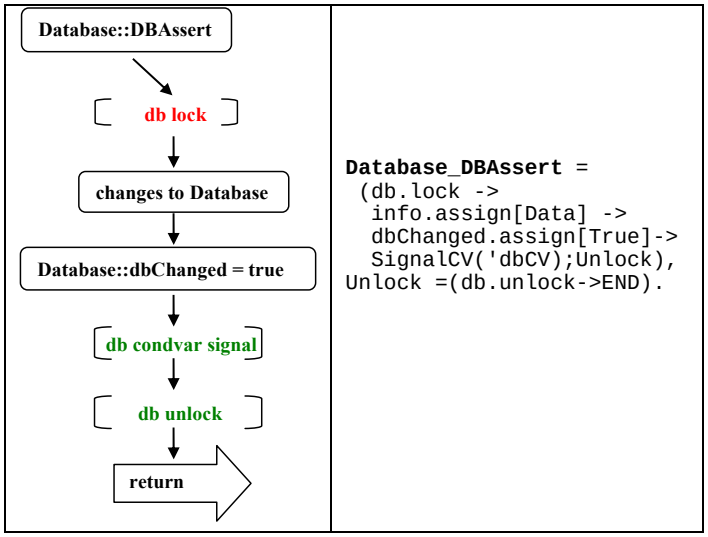


Figure 6. Original design (left) and corresponding FSP model (right) produced for a method in the database

These documents were produced "after the fact", meaning after a first implementation of the Rover was available. It took the developer *only a few hours* to produce these documents. Moreover, he found the diagrams that we produced of the architecture of the system helpful, and subsequently maintained it for communicating the structure of the system to his collaborators.

Figure 6 illustrates the original design provided to us and the corresponding (FSP) model that we produced. In the model, *Data* is the domain of values for variable info. *SignalCV* is the method that needs to be called to signal a specific condition variable, in this case *dbCV*. *Unlock* is simply a state alias – mutex *db* must get unlocked after signaling *dbCV* and before returning.

We made a systematic effort to keep the architecture explicit in the model. Each thread has a unique instance name – the name of the thread in the architecture – which prefixes all the actions in its behavior, thus clearly differentiating its behavior from that of other threads in the system. This was achieved by the instantiation operator that the LTSA tool provides. Moreover, communication points were modeled by binding the associated actions, captured by the renaming operator of the LTSA. The resulting model was approximately 600 lines of FSP code that had a very close correspondence to the design documents provided by the developer.

Modeling the New Executive

As mentioned, the developer changed the design of the Executive, partly as a result of our analysis. We created two new models for the design level analysis of the new executive. Model 1 (~800 lines of FSP code) captures the new architecture of the executive, the queuing mechanism and the detailed event handling for block and task nodes. Model 2 (~900 lines of FSP code), which captures synchronous and asynchronous execution of floating branches.

Model 1: Queuing and Event Handling
We reused from our previous model the FSP encoding of the functionality of mutexes and condition variables. We added a model for the FIFO *Queue* and the event handling mechanism in the *Executive* and we updated the *ExecCondChecker* and *ActionExecution* models according to the new design, as illustrated in Figure 3.

Model 2: Floating Branch Execution
We extended Model 1 to handle the execution of floating branches. This execution is triggered by pre-defined conditions. Floating branches can be synchronous (i.e. triggered at action transitions within the plan) or asynchronous (i.e. monitored continuously in parallel with execution). The execution of floating branches involves suspending execution of the principal plan, executing the floating branch, and resuming execution in the principal plan. In the case of asynchronous floating branches, the currently executing action is suspended, and then it resumes after completion of the floating branch. We extended the Executive's main loop to deal with new events (e.g. *Task Suspension/Suspended*, *Task Resume*, *Floating Branch Expand* and *Floating Branch Terminate*). We also extended the event handling mechanism in node (i.e. block or task) execution, to deal with suspension/resuming of the execution of the current node when a floating plan is activated. We added functionality for event handling in nodes of synchronous and asynchronous floating branches.

Properties

Our analysis focused on properties related to the correct execution of the plans, according to the plan semantics, and to the synchronization issues between threads. Specifically, we analyzed the following properties:

P1: *Absence of local and global deadlocks.*

P2: *No irrelevant action execution events can happen.*

P3: *No irrelevant condition checker events can happen.*

P4: *If the last task in the plan terminates successfully, then the only possible outcome for the plan is successful termination.*

P5: *When a task fails, the continue-on-failure flag on the block will always be checked before any outcome is produced; moreover, if continue-on-failure is true, the outcome is success, otherwise it is failed.*

P6: *The Executive only receives ExecCondChecker events if it has registered for them.*

P7: *The ExecCondChecker only puts events in the queue if the Executive registered for them.*

P8: *When a task fails, it will always check its continue-on-failure flag; moreover, if the continue-on-failure flag is false, no subsequent task in the block will be started; new tasks can be started after the parent block reports the results (i.e. other block is expanded).*

P9: If the Executive thread reads the value of the shared variable savedWakeupStruct, then the ExexcCondChecker thread should not read it until the Executive clears it first.

P10: *(Race condition) All accesses to shared structure conditionSetChanged by the Executive and the ExecCondChecker threads will be protected by locks.*

P11: *(Race condition) All accesses to shared structure existConditions by the Executive and the ExecCondChecker threads will be protected by locks.*

P12: *Floating branches and principal plans cannot execute concurrently.*

Design Level Verification

Our initial analysis uncovered a number of synchronization problems such as deadlocks and data races. Moreover, the design models were used for quick experimentation with alternative solutions to exiting defects, leading eventually to the re-design of the software.

As mentioned, safety properties are expressed as LTSs. For example, Figure 7 illustrates property P9 that was formulated by the developer. The property is represented as two states, corresponding to the shared variable *savedWakeupStruct* being cleared or not cleared, and with a third state representing the error state. The developer

expected the property to be satisfied. We applied assume-guarantee reasoning as supported by our tools, were assumptions were generated for the *ExecCondChecker* thread (module M_1) and discharged on the *Executive* thread (module M_2).

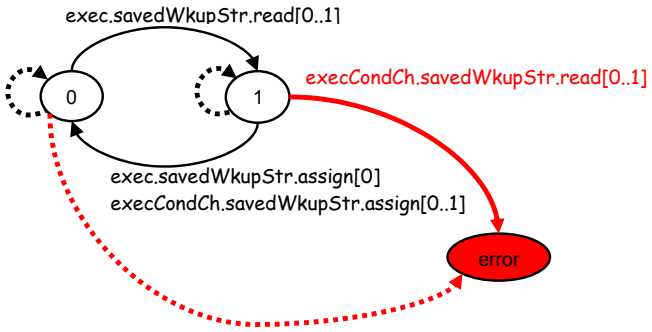


Figure 7. Example property

The results obtained from the design-level analysis are summarized in the first row of Table 1. The design level model is an order of magnitude smaller than the corresponding Java implementation. The largest state space that our modular verification techniques compute consists of 541 states, as opposed to 4672 states computed by monolithic model checking. We therefore achieved an order of magnitude savings in terms of space.

Table 1. Analysis results at design & code level

Analysis	Tool	LOC	Monolithic model checking	Modular verification
Design level	LTSA	700 FSP	4672 states	541 states
Code level	JPF	7.2K Java	183K states	53K states

The generated assumption consists of 5 states. It describes an environment where the *Executive* thread reads the *savedWakeupStruct* variable after acquiring the *exec* mutex and holds the mutex until it clears (assigns value 0) the variable. The assumption is illustrated below (in FSP).

Assumption = Q0,
Q0 = (**executive.exec.lock** -> Q2),
Q2 = (executive.exec.unlock -> Q0
| **executive.savedWakeupStruct.read[1]** -> Q3
| executive.savedWakeupStruct.assign[0] -> Q4
| **executive.savedWakeupStruct.read[0]** -> Q5),
Q3 = (executive.savedWakeupStruct.read[1] -> Q3
| **executive.savedWakeupStruct.assign[0]** -> Q4),
Q4 = (**executive.exec.unlock** -> Q0
| executive.savedWakeupStruct.assign[0] -> Q4
| executive.savedWakeupStruct.read[0] -> Q5),
Q5 = (**executive.savedWakeupStruct.assign[0]** -> Q4
| executive.savedWakeupStruct.read[0] -> Q5).

This assumption could not be discharged on the *Executive* thread. The counterexample obtained describes a scenario where the *Executive* thread reads *savedWakeUpStruct* and then it performs *wait* on a condition variable associated with the *exec* lock (a *wait* operation automatically releases the lock). The problem was temporarily fixed by adding to the *Executive* thread a statement that clears the shared variable. Note that the variable *savedWakeupStruct* was eliminated altogether when the Executive was re-designed.

Stage I In the first stage, we checked several simple properties (P1, P2, P3). To do this, we decomposed the system into two modules, M_1 that consists of the *Executive*, the *ActionExecution* and the *EventQueue*, and M_2 that consists of the *ExecCondChecker* and the remaining threads in the system. The results of our analysis are summarized in Tables 2a-2c.

Table 2a. Analysis results - stage I

Property	Subsystem	#States, #Trans	|A|	Result
P1	M_1	3805, 10450	n/a	false
P2	M_1	8478, 22875	n/a	true
P3	M_1	8478, 22875	374	false

Table 2b. Analysis results – property P3

Subsystem	#States
M_1	8478
M_2 (discharge)	18080
$M_1 \| \| M_2$ (CRA)	74649
$M_1 \| \| M_2$ (monolithic)	84690

Table 2c. Reachable state space computation

Subsystem	#States
M_1	8478
M_2	14448
$M_1 \| \| M_2$ (monolithic)	> 10 Million

We first checked local and global deadlocks (P1) by incrementally putting components of M_1 and M_2 together. Note that, in the LTSA, the assumption is that environment inputs are always available. This is a significant benefit for modeling partially specified systems (or verification of modules of systems), because one does not need to explicitly model drivers for the component. Moreover, uninteresting cases where the Executive is deadlocked because no plans are available at the input are ignored.

A local deadlock was detected in M_1. Two threads, *Executive* and *ActionExecution*, synchronize on shared transitions (in order to start and stop the execution of actions) and they also synchronize via the *EventQueue* (i.e., the *ActionExecution* sends events when the execution of the action is completed). The counterexample represents

a behavior where the *Executive* tries to stop the current action, without knowing that the current action was completed (i.e., before processing the respective event), while the *ActionExecution* is waiting to start a new action. This was a problem in our design, which we fixed (by adding self-loops for "unconsumed" stops from the previous actions).

Property P2 was checked on M_1. This property holds in any environment. Property P3 was checked on the same subsystem. This property does not hold in any environment, since it depends on the behavior of the *ExecCondChecker*, which is in M_2. We generated automatically the assumption that M_1 needs to make about the *ExecCondChecker* for the property to hold. We obtained an assumption of 374 states. By minimizing M_1 using compositional reachability analysis as supported by the LTSA, we obtain a subsystem of 1493 states; the assumption is therefore more concise to use for analysis.

When we tried to discharge this assumption on the *ExecCondChecker*, after exploring 18080 states we obtained a counterexample describing the following scenario: the *ExecCondChecker* detects the fact that a maintenance condition has been broken, sends an event to the *EventQueue*, but the action terminates before this event gets handled. As a result, the event remains unconsumed in the *EventQueue* and gets handled in the context of the next node, at which time it is irrelevant. The counterexample exhibited the fact that the system is highly asynchronous, as a result of which it is possible for the *EventQueue* to hold "obsolete" events that are no longer relevant to the execution of the current node.

As illustrated in Table 2b, our assume-guarantee framework enables a significant reduction in the state space that needs to be explored (18080 states) as compared both to CRA (74649 states) and to monolithic model checking. Note that, as illustrated in Table 2c, if we disable error detection and simply compute the reachable state space of the model, monolithic model checking runs out of memory after exploring 10 million states.

Stage II After we enriched our models with advanced autonomy features (i.e. detailed task and block execution, floating branch execution, etc.) we checked the remaining properties (P4, P12 – except P9 which was no longer applicable). We should first note that we could not compute the reachable states of the whole system (the computation runs out of memory when using 1GB of memory); this means that checking any property on the whole system would not complete.

We therefore used compositional techniques. Again, we decomposed the system into components: M_1 (the *Executive* thread, *the Event Queue* and the *ActionExecution* thread) and M_2 (the *ExecCondChecker* thread). M_1 has 47906 states, M_2 has 14496 states. We analyzed the properties using assume guarantee reasoning. Checking properties P6, P7, P10, P11 required small assumptions (the largest obtained assumption has 7 states). Interestingly, properties P4, P5, and P8 were checked locally (no environment assumption was necessary). This reflects the modularized architecture of the new executive.

During our analysis we discovered a problem with the design (reflected in the implementation) due to the asynchronous communication between components through the queue. Specifically, property P4 did not hold because of the order of events arriving in the queue: if a task terminates successfully and at the same time a time-out occurs or a condition fails for the parent block, then, the outcome for the parent block can be non-deterministically success or failure, depending on the order in which the corresponding events are put in the queue. Similar problems were found in relation to the execution of synchronous floating branches. The problems were corrected according to the developer's suggestions, by adding an extra test for cases when events signaling time-outs or failed conditions are received by the *Executive* thread.

It is interesting to note that compositional reachability analysis fails for this large case study. E.g. for M_1 composed with property P6 which has 47918 states, compositional reachability analysis runs out of memory, while the generated assumption has only 5 states computed in 16.165 seconds.

Code Level Verification

Model Checking We used JPF for the analysis of the software components of the first version of the Executive code (which was manually translated in Java). To check each component in isolation, we used the assumptions that were generated during design level analysis to build appropriate environments. Techniques for automated generation of environments from user supplied assumptions are presented in [6]. These environments provide stubs for the methods called by the component that are implemented by other components, or test drivers that execute a component by calling methods that the component provides to its environment. Moreover, these environments are constrained by the design level assumptions.

Some of the results of this analysis are reported in the second row of Table 1. Compositional verification yields a 3x improvement (in terms of memory used) over monolithic verification. E.g., when we checked property P9 on the corrected system, monolithic (non-compositional) model checking explored 183K states and it consumed 952 Mb of memory in 12 minutes and 12 seconds. In contrast, compositional verification explored at most 60K states, and it consumed 315 Mb in 6 minutes and 55 seconds.

Run Time Verification We used Eagle for the run time verification of the C++ code of the Executive. The design-level artifacts (properties and assumptions) were automatically translated into Eagle monitors. We instrumented (by hand) the code of the Executive, to emit events that appear in these assumptions and properties. To generate test input plans, we encoded the plan language grammar as a non-deterministic input specification. Running model checking on this specification generates hundreds of input plans in a few seconds.

We developed a tool that integrates run time verification and test input generation to perform assume guarantee style reasoning about the run-time behavior of the executive. The tool generates a set of test input plans. A script runs the Executive on each plan and it calls Eagle to monitor the generated run-time traces. The user can choose to perform a whole program (monolithic) analysis or to perform assume-guarantee reasoning. In the latter case, the Executive is broken in two parts: M_1 consists of the *Executive* thread, the *Event Queue* and the *ActionExecution* thread, and M_2 consists of the *ExecCondChecker* thread and the remaining threads.

We ran several experiments for different input plan configurations. For Property P6, we found a discrepancy between the implementation and the models, due to the fact that nodes can send *null* conditions. Instead of putting these in the condition list (and altering the values of variables *conditionSetChanged* and *existConditions*), the *ExecCondChecker* code immediately pushes an event to the queue. We corrected this in the model.

One benefit of our approach is that the use of design-level assumptions in the verification of software implementations enables the detection of costly integration problems well prior to system integration. In fact, assume-guarantee verification can detect such problems as soon as one component of a software system becomes "code complete" (while the remaining software components may not be even implemented yet). Whenever the complete implementation of one component becomes available, we can check it against the required properties, under environments that are suitable restricted by the design-level assumptions. When the rest of the components become available, we check the assumptions on these components. As a result, we guarantee that the whole system behaves correctly, without being necessary to perform verification/testing on the integrated components.

Also note that assume-guarantee verification provides better unit testing. We only test components (i.e. units) in the environments in which it can be expected that the units will be integrated. Moreover, assume-guarantee reasoning provides increased behavioral coverage of the integrated system. We have found that, in some cases, assume-guarantee verification uncovers errors that escape integration testing. The reason is that by generating and analyzing traces for each component in isolation, we can predict, by mathematical inference, properties about all the possible inter-leavings of these traces, while at system integration, one could generate only a subset of these interleavings.

Conclusions and Future Work

We described the development and application of compositional verification techniques to a significant autonomous system throughout its lifecycle. Subtle errors in the design and implementation of a rover executive have been detected Compared to testing by itself, these techniques are aimed at two challenging aspects of autonomy verification:

1) Assuring correct execution of plans. Robust execution of plans, especially plans with contingencies, is a significant advantage of autonomy software compared to traditional sequence execution. Automatic verification of such extended features is a prerequisite for their introduction in missions.

2) Concurrency is an inherent feature of autonomy. First, responding robustly to asynchronous environmental changes introduces concurrency. Second, in contrast to sequence execution or simple sequential plans, plans for rovers consist of multiple concurrent tasks overlapping in time. Third, modern programming practices for complex software favors encapsulating control into multiple threads, introducing concurrency at the implementation level. However, concurrency is difficult to debug with testing alone: concurrency errors are typically manifested only as transient faults in black-box testing, and are often masked by thread scheduling and computational resource profiles that differ subtly and uncontrollably between the testing environment and actual field conditions. Our techniques provide high assurance that autonomy software is free of concurrency errors.

The techniques presented have several benefits. First, they can be applied early in the software development life cycle, when it is cheaper to detect and fix bugs. Second, our compositional techniques provide a way to automatically decompose global (system-level) requirements into local properties, which are cheaper – in terms of time and consumed memory – to check, with an increased level of confidence. Third, assumptions allow checking global properties (that are usually checked at integration testing) at unit testing level, thereby increasing the chances of detecting costly integration errors early. Fourth, our results show that compositional reasoning can enhance integration testing at the source-code level (by exploring multiple inter-leavings in concurrent programs).

In our work we consider a top-down software development process where design models are first created and debugged, and are subsequently used to guide the implementation. It goes without saying that it is not a straightforward task to obtain a correct model. However, verification tools provide several features such as interactive simulation, which facilitate the debugging of models. Moreover, as our results show, it is essential to make connections between verification performed at the design level with the actual implemented system. Note that we are currently investigating a complementary approach that uses abstraction techniques to automatically extract models from source code [3].

In the future, we plan to leverage our work for the analysis of other executives (and autonomy software), with minimal modifications (e.g. for plan generation, we could simply modify the plan language grammar; for properties and assumptions, we expect to build upon the existing specifications). For example, we plan to participate in the development and analysis of next generation executives, built within the CLARAty decision layer distribution [10].

References

[1] J. M. Cobleigh, D. Giannakopoulou, C. S. Pasareanu, Learning Assumptions for Compositional Verification, in Proc. 9th International Conf. on Tools and Algorithms for the Construction and Analysis of Systems, 2003.

[2] D. Giannakopoulou, C. S. Pasareanu, H. Barringer, Component Verification with Automatically Generated Assumptions, J. of Automated Software Engineering, 2005.

[3] S. Chaki, E. Clarke, D. Giannakopoulou, and C. Pasareanu. Abstraction and assume-guarantee reasoning for automated software verification. *RIACS TR 05.02*, October 2004.

[4] D. Giannakopoulou, C. S. Pasareanu, J. M. Cobleigh, Assume-guarantee Verification of Source Code with Design-level Assumptions, Proc. of the 26th International Conf. on Software Engineering, 2004.

[5] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. S. Pasareanu, G. Rosu, K. Sen, W. Visser, R. Washington, Combining Test Case Generation and Runtime Verification, in Theoretical Computer Science, 2004.

[6] O. Tkachuk, M. Dwyer, C. S. Pasareanu, Automated Environment Generation for Software Model Checking, in Proc. of the 18th IEEE International Conf. on Automated Software Engineering, 2003.

[7] D. Dvorak, R. Rasmussen, G. Reeves, A. Sacks. Software Architecture Themes in JPL's Mission Data System. In Proceedings 2000 IEEE Aerospace Conference.

[8] J. Magee, and J. Kramer, Concurrency: State Models & Java Programs: John Wiley & Sons, 1999.

[9] S. C. Cheung, J. Kramer, Checking Safety Properties using Compositional Reachability Analysis, ACM Transactions on Software Engineering and Methodology, 8(1):49-78, 1999.

[10] I.A. Nesnas, A. Wright, M. Bajracharya, R. Simmons, T. Estlin, W.S. Kim, CLARAty: An Architecture for Reusable Robotic Software, SPIE Aerosense Conf., 2003.

[11] E. M. Clarke, O. Grumberg, and D. A. Peled, Model Checking: The MIT press, 1999.

[12] T. A. Henzinger, S. Qadeer, and S. K. Rajamani, You assume, we guarantee: methodology and case studies, in Proc. of the International Conf. on Computer-Aided Verification (CAV'98). LNCS 1427, pp. 440-451.

[13] C. B. Jones, Specification and design of parallel programs. Information Processing 83: Proceedings of the IFIP 9th World Congress, 1983: pp. 321--332.

[14] A. Pnueli, In Transition for Global to Modular Temporal Reasoning about Programs, in Procedings of the Logic and Models of Concurrent Systems. 1985.

[15] G. Brat, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. S. Pasareanu, A. Venet, W. Visser, R. Washington, Experimental Evaluation of verification and validation Tools on Martian Rover Software, in International Journal on Formal Methods in System Design, September – Nov. 2004, 25(2-3), 167-198.

[16] R. Washington, K. Golden, J. Bresina. Plan Execution, Monitoring, and Adaptation for Planetary Rovers. Electronic Transactions on AI, 4(A):3-21,2000.

[17] D. Angluin, Learning Regular Sets from Queries and Counterexamples, Information and Computation, 75(2).

[18] W. Visser, K. Havelund, G. Brat, S. Park, F. Lerda. Model Checking Programs. Automated Software Engineering Journal. Volume 10, Number 2, April 2003.

[19] S. Bensalem, M. Bozga, M. Krichen, S. Tripakis. Testing conformance of real-time software by automatic generation of observers. In Proc. of Runtime Verification Workshop, RV'04 April 2004.

[20] A. Akhavan, S. Bensalem, M. Bozga, E. Orfanidou. Experiment on Verification of a Planetary Rover Controller. In Proc. of the 4th Intl. Workshop on Planning and Scheduling for Space, IWPSS'04, June 2004.

B vs OCL: Comparing specification languages for Planning Domains

D. E. Kitchin, T. L. McCluskey and **M. M. West**
School of Computing and Engineering
The University of Huddersfield, Huddersfield HD1 3DH, UK
D.Kitchin,T.L.McCluskey,M.M.West@hud.ac.uk

Abstract

In this paper we examine the specification and validation of Artificial Intelligence Planning domain models using the B Abstract Machine Notation and its associated tool support. We compare this to the use of *OCL* (object-centred language) within its tool-supported environment, GIPO. We present encodings of two well-known AI planning domain models, the Blocks world and the Tyres world, with the aim of finding a correspondence between the B and the OCL languages. We also compare the tool-supported validation offered by their respective environments.

Introduction

As AI Planning and Scheduling systems become mature enough to be deployed in safety-related and safety critical systems, the reliability of the systems themselves, and the accuracy of the knowledge models that underlie the systems, need to be certified to a high level. Planning systems typically contain planning engines, plan execution architectures, plan generation heuristics and application domain models. In this paper we focus on techniques for the rigorous construction and validation of the application domain model. This typically contains a structural model of the objects and constraints in the planning world, and a model of the actions/events that affect objects in that world.

It is likely that any reasonably sized realistic domain model will continue to contain errors and inconsistencies for some time. A planner may manage to produce a solution despite the fact that the domain model is flawed. Alternatively no plan will be produced because of inconsistencies in the domain model. Whatever the case it is desirable to be able to validate the domain model *before* an attempt is made to generate a plan. One approach to this is to use model checking for validation, as in (Penix, Pecheur, & Havelund 1998), but this is limited by potential state space explosion. Another approach could be to assume **a priori** that the domain model will be incomplete as in the *SiN* algorithm (Munoz-Avila *et al.* 2001). SiN can generate plans given an incomplete domain theory by using cases to extend that domain theory, and can also reason with imperfect world-state information. This is a fruitful assumption in many ways, as philosophically no model can ever be 'proved' complete and correct. However, this approach neglects the issue of correctness - the incomplete parts must still be validated and bugs identified and eliminated.

In this paper we investigate the use of a formal method from the area of software specification to capture planning domain models. These mathematically based methods are chiefly for use in applications where safety-critical software has to be produced, and where validation (of the specification) and verification (of software derived from the specification) are important considerations. The method we chose is B, in conjunction with its tool support the B-Toolkit (B-Core (UK) Ltd). B is an industrial-strength method which has been used in a wide variety of software applications.

To facilitate this, we compare it to a language and method specifically aimed at capturing planning domain models: the planning language OCL (object-centred language) and its platform GIPO (Graphical Interface for Planning with Objects) (Simpson *et al.* 2001)). GIPO is a GUI and tools environment for building AI planning domain models in OCL which supports some validation activities such as consistency and animation. GIPO provides both a graphical means of defining a planning domain model and a range of validation tools to perform syntactic and semantic checks of emerging domain models.

Superficially at least, formal specification languages and planning domain description languages are similar in that they share

1. the concept of a 'state'
2. the technique of using pre and post conditions in state transformation, via operations to specify state dynamics
3. the assumptions of closed world, default persistence and instantaneous operator execution
4. the presence of state invariants for validity and documentation purposes. State invariants are also used in OCL.

In the paper we use this correspondence to help in the comparison. In the next sections we apply both B and OCL methods to the acquisition of the two domains, and finish by making a comparison of their performance with respect to validation and consistency checking,

The Blocks World in OCL

As a case study that all planning researchers are aware of, we use a version of the well known 'Blocks world', show-

ing how this can be modelled in both languages, and what opportunities there are for validation. The version we use will consist of a table, on which there are a number of blocks. There are robot arms capable of gripping individual blocks and moving them from one location to another. The complete specification is in the *Resource* section at http://scom.hud.ac.uk/planform/.

The object-centred family of languages (OCL) and their associated development method (embedded in the GIPO tool) forms a rigorous approach to capture the functional requirements of classical planning domains. OCL derives from the work in reference (McCluskey & Porteous 1997). Originally designed for classical goal achievement planning, *OCLh* has been developed for HTN models and PDDL+ – like models (Simpson & McCluskey 2003). We will use the basic version of OCL and the tools available in GIPO to support this.

A specification of the model $\mathcal{M}$ of a domain of interest $\mathcal{D}$, is composed of sets of:

- sort names and object names: A sort (or object class) is a set of object identifiers representing objects that share a common set of characteristics and behaviours. A sort is *primitive* if it is not defined in terms of other sorts. *Sorts* in the Blocks world are *block* and *gripper* and are both primitive.
- predicate definitions: (*Prds*) A predicate from *Prds* represents a functional property of an object. Predicates can be static or dynamic - static predicates include built-in ones such as 'ne' (not equal). The *Prds* of the Blocks world are in Figure 1.
- invariant expressions on individual sorts *Exps*: this is a set of invariants which define all the possible "states" that an object of each sort can inhabit. These are called substates to distinguish them from a world state. An object description is specified by a tuple (s, i, ss), where s is a sort identifier, i is an object identifier, and ss an object's substate. For example, $(gripper, G, [free(G)])$ is an object description meaning that some object G of sort *gripper* is free. Substates operate under a *closed world* assumption local to this restricted set - thus in Figure 1 a block can either be gripped, stacked on another block and clear, stacked on another block and not clear, on the table and clear, or on the table and not clear. For object *block B*, substate $gripped(B, G)$ means that other predicates relating to *block* are *not* true: it is not on the table or clear or on another block.
- general domain invariants: Within OCL general constraints linking sorts can be stated and used in tools. A typical example in the Blocks world is the assertion "for any blocks B, B1, gripper G, $gripped(B, G), on\_block(B, B1)$ is inconsistent".
- operator schema: An action in a domain is represented by an *operator schema*. Actions or events change objects substates. An operator shows the set of object transitions for each object affected by the action. It is specified by a *name*, a set of *prevail* conditions, a set of *necessary* changes and a set of *conditional* changes.

```
predicates:
  on_block(block,block)
  on_table(block)
  clear(block)
  gripped(block,gripper)
  busy(gripper)
  free(gripper)
invariants of 'block': an object B must be
described by exactly one of the following
expressions:
   gripped(B,G)
   on_block(B,B1),clear(B),ne(B,B1)
   on_block(B,B1),ne(B,B1)
   on_table(B),clear(B)
   on_table(B)
```

Figure 1: The Predicates and Substates of Blocks World

```
name: grip_from_table
parameters: B - block, G - gripper
prevail - none
necessary transitions -
block, B:  [on_table(B),clear(B)]
             => [gripped(B,G)]
gripper,G: [free(G)]
             => [busy(G)])
conditional - none
```

Figure 2: Operator for Gripping a Block from a Table

Operators in the Blocks world contain only necessary transitions. These show the conditions on objects that must be true before an action can take place, and specify the new state of an object after the action has been executed. For example, the *grip_from_table* operator has a necessary transition:

```
For any block B
[on_table(B),clear(B)] => [gripped(B,G)]
```

meaning that block B has to be clear and on the table as a precondition and that its state after the transition is that it is gripped. An example *OCL* operator, *grip_from_table*, shows the state changes for two objects, a block and a gripper (see Figure 2 and Figure 3).

Transitions are only shown for objects which are changed. By default all other objects are assumed to remain unchanged. If an object is required to be in a particular state before the transition but does not itself change, it is included as a *prevail* condition. However it is not used in any of the actions of the Blocks world. The meaning of *conditional* change is that **if** a condition on an object is true before an action takes place, **then** the object changes to a new specified state. There are no conditional changes in the Blocks world domain model.

Validation and debugging in OCL

There are several built-in security checks in OCL. Firstly, the user has to capture the space of possible descriptions (substates) of an object of each sort within the sort invariants. Thus world states are formally defined as being a set of legal substates - one for each object declared. This gives an

grip_from_blocks(B, G):

```
[on_block(B,B1),clear(B),ne(B,B1)]
            => [gripped(B,G)]
[on_block(B1,B2),ne(B1,B2)]
            => [on_block(B1,B2),clear(B1)]
```

grip_from_one_block(B, G):

```
[on_block(B,B1),clear(B),ne(B,B1)]
            => [gripped(B,G)])
[on_table(B1)]
            =>[on_table(B1),clear(B1)]
```

Figure 3: Transitions for Blocks when being Gripped from a Block

explicit specification of all possible world states, allowing bugs in state expressions to be prevented. It also restricts the set of goal expressions to those that are feasible in the domain. Secondly, the object transitions in operator schema must conform to the invariants, and hence the transitions are restricted so that operator schema make objects transform to a legal state according to the invariant.

These checks are embedded in GIPO, the GUI and tools environment for building domain models. GIPO prevents errors being introduced (by restricting values in menus etc) and in some cases GIPO's validation checks reveals errors. Other kinds of validation supported by GIPO include: a *Stepper* which aids the user to interactively build up a plan, selecting and applying operator schema for a chosen task; and a *PDDL interface*, allowing third party planners to be bolted on, and their output returned back into a GIPO animator, so that the user can step through a complete plan.

The Blocks World in B

A specification in B will be constructed from one or more abstract machines, with the components of a machine being its variables, invariant, initialisation and operations. A typical abstract machine state comprises several variables which are constrained by the machine invariant and initialised. Operations on the state contain explicit preconditions; the post-conditions are expressed as 'generalised substitutions'. Further information describing B can be found in (Schneider 2001).

Some of the sets and logic notation of B is 'standard'. There follows a brief explanation of other parts which may not be familiar to the reader.
If R is a relation from S to T and $A \subseteq S, B \subseteq T$:
$A \lhd R$ means 'restricting the domain of R to set A';
$A \lhd\!\!\!- R$ means 'restricting the domain of R to set $S - A$';
$R \rhd B$ means 'restricting the range of R to set B';
$R \rhd\!\!\!- B$ means 'restricting the range of R to set $S - B$';
If R_1, R_2 are relations from S to T:
$R_1 <\!\!+ R_2$ means 'domain overriding of R_1 by R_2 '. Hence on the domain of R_2, the value is given by R_2. Outside $\mathrm{dom}(R_2)$, the value is given by R_1.

In applying B to AI Planning, our strategy was to find a correspondence between B specifications of planning worlds and planning-specific languages, hence we used reverse engineering on the OCL model. This gives the correspondence in Table 1.

OCL	B
primitive sorts	sets
predicate names	variable names
operator schema	operations
properties	boolean-valued functions
predicates(x,y)	relations between x and y

Table 1: OCL - B correspondence

Sets in B translate to 'primitive sorts' in OCL. For example *Block*, *Gripper* in B map to corresponding primitive sorts in OCL;

Boolean valued functions in B map to OCL predicates of arity one. For example function *On_Table*$(block)$ maps to predicate *on_table*$(block)$ in OCL;

Relations in B whose *domain* is the type of x and whose *range* is the type of y ($\in X \leftrightarrow Y$) map to OCL predicates of arity 2, *pred*(x, y). For example *On_Block* $\in$ *Block* $\rightarrowtail$ *Block* becomes the predicate *on_block*$(block, block)$ in OCL. (Note that we have restricted ourselves to domains capable of being modelled via predicates of arity two.)

The following comprises the B machine header, sets and variables clauses:

MACHINE *Blocks_World*
SETS *Block*; *Gripper*
VARIABLES
On_Block, *On_Table*, *Clear*, *Gripped*, *Free*

Note that the predicate 'Busy' from OCL is not represented for it is simply the negation of 'Free'. This exception was made to avoid unnecessary replication in the B model. A fragment is presented in the next subsection. The complete specification and that of the Tyres World is in
http://scom.hud.ac.uk/scommmw/PlanningDomains/

Invariant and Initialisation of Blocks World in B

The types of the variables were reverse engineered - OCL predicates of arity one, *pred*(x) were modelled by B total functions whose *domain* is the type of x and whose *range* is the booleans. OCL predicates of arity 2, *pred*(x, y), were modelled by B relations whose *domain* is the type of x and whose *range* is the type of y. Modelling in this manner allowed us to take advantage of the fact that in the Blocks world all the relations were functions and some were 1-1.

A simple initialisation condition (below) was specified - that each block is on the table, clear and not gripped. The '||' stands for *parallel substitution* where all variable substitutions are assumed to take place in parallel rather than in sequence. The idea of a fixed initialisation differs from OCL where the domain is initialised at the start of each plan.

INITIALISATION
On_Block $:= \{\}$ ||

INVARIANT

$$
\begin{array}{ll}
On\_Block \in Block \rightarrowtail\!\!\!\!\!\!\rightarrow Block \wedge & (1)\\
On\_Table \in Block \rightarrow BOOL \wedge & (2)\\
Clear \in Block \rightarrow BOOL \wedge & (3)\\
Gripped \in Block \rightarrowtail\!\!\!\!\!\!\rightarrow Gripper \wedge & (4)\\
Free \in Gripper \rightarrow BOOL \wedge & (5)\\
\forall\, blk \,.\, (\, blk \in \mathsf{dom}\ On\_Block \Rightarrow On\_Block\,(\, blk\,) \neq blk\,) \wedge & (6)\\
\mathsf{ran}\ On\_Block \cup \mathsf{dom}\ Gripped = \mathsf{dom}\ (\, Clear \rhd \{\, FALSE\,\}\,) \wedge & (7)\\
\mathsf{ran}\ Gripped \cap \mathsf{dom}\ (\, Free \rhd \{\, TRUE\,\}\,) = \{\} \wedge & (8)\\
\mathsf{ran}\ Gripped \cup \mathsf{dom}\ (\, Free \rhd \{\, TRUE\,\}\,) = Gripper \wedge & (9)\\
\mathsf{dom}\ On\_Block \cap \mathsf{dom}\ Gripped = \{\} \wedge & (10)\\
\mathsf{dom}\ Gripped \cap \mathsf{dom}\ (\, On\_Table \rhd \{\, TRUE\,\}\,) = \{\} \wedge & (11)\\
\mathsf{dom}\ On\_Block \cap \mathsf{dom}\,(\, On\_Table \rhd \{\, TRUE\,\} = \{\} \wedge & (12)\\
\mathsf{dom}\ Gripped \cap \mathsf{dom}\ (\, Clear \rhd \{\, TRUE\,\}\,) = \{\} \wedge & (13)\\
\mathsf{dom}\ Gripped \cap \mathsf{dom}\ On\_Block = \{\} \wedge & (14)\\
\mathsf{dom}\ Gripped \cap \mathsf{ran}\ On\_Block = \{\} \wedge & (15)\\
\mathsf{dom}\ Gripped \cup \mathsf{dom}\ (\, On\_Table \rhd \{\, TRUE\,\}\,) & \\
\quad \cup\, \mathsf{dom}\ On\_Block = Block & (16)
\end{array}
$$

Figure 4: B Invariant for the Blocks World

$On\_Table := Block \times \{\, TRUE\,\} \;||$
$Clear := Block \times \{\, TRUE\,\} \;||$
$Gripped := \{\} \;||$
$Free := Gripper \times \{\, TRUE\,\}$

Blocks World Operations in B

It is only necessary to have one operation for gripping a block in B. However, *Grip_Block_On_Table* was represented plus one operation *Grip_Block_On_Block* to represent the two actions required by OCL. (See Figure 5). This was so that we could compare the representations more closely. For a similar reason two operations were also specified to release a block: *Put_Block_On_Table*, *Put_Block_On_Block*.

Validation

Reasoning about a formal specification and animation of a formal specification are both activities concerned with *validation*, and these are complementary activities.

A way of reasoning about a formal specification is via the generation and discharge of 'proof obligations'. A set of proof obligations involving *consistency properties* of a system can be automatically generated by the BTool. An example of two of these is (1) Consistency of initialisation: the initialisation *must* establish the invariant. (2) Consistency of operation: each operation must *preserve* the invariant. Other consistency properties involve the static parts of the machine (sets, constants, properties etc.). It is also possible to check the machine invariant during animation.

The B-Toolkit includes an animator to 'execute' operations, and a proof tool to check that proof obligations are met. The Blocks world in B was validated using the B-Toolkit. Each new version was animated to check for errors. The version was run for each operation with the invariant displayed and this provided a quick method for rooting out errors *before* the prover was used. The proof obligation generator and prover were then run - in all 72 proof obligations were generated with 40 undischarged by the prover - these were subsequently hand-checked, which was a laborious process. During this procedure the invariant was frequently scanned and it was discovered that an unnecessary conjunct was present in the original versions:
$\mathsf{ran}\,(\, On\_Block\,) \cap \mathsf{dom}\,(\, Clear \rhd \{\, TRUE\,\}\,) = \{\}$
was found to be already covered by (7) and (13).

As part of the checking process of the two models, we ran one of the planners in GIPO on a particular task, and then simulated this in the B-Toolkit using the animator. We used the well-known task (in AI planning literature), the Sussman Anomaly, as shown in Figure 6. This solution was obtained

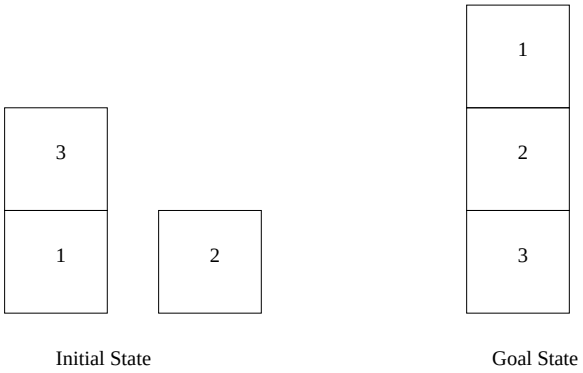


Figure 6: Initial and Goal States

from one of the planners:

```
grip_from_one_block(block3,block1,tom)
put_on_table(block3,tom)
grip_from_table(block2,tom)
put_on_one_block(block2,tom,block3)
grip_from_table(block1,tom)
put_on_blocks(block1,tom,block2,block3)
```

It was obviously not possible to generate an automatic sequence of operations using the B-Toolkit - as in the case of

Grip_Block_On_Table (*blk* , *grp*) $\widehat{=}$
PRE
$grp \in \mathsf{dom}\ (\ Free \rhd \{\ TRUE\ \}\ ) \wedge$
$blk \in \mathsf{dom}\ (\ Clear \rhd \{\ TRUE\ \}\ ) \wedge$
$On\_Table\ (\ blk\ ) = TRUE$
THEN
$Free\ (\ grp\ ) := FALSE\ ||$
$Clear\ (\ blk\ ) := FALSE\ ||$
$On\_Table\ (\ blk\ ) := FALSE\ ||$
$Gripped\ (\ blk\ ) := grp$
END

Grip_Block_On_Block (*grp* , *blk*) $\widehat{=}$
PRE
$grp \in \mathsf{dom}\ (\ Free \rhd \{\ TRUE\ \}\ ) \wedge$
$blk \in \mathsf{dom}\ (\ On\_Block\ ) \wedge$
$blk \in \mathsf{dom}\ (\ Clear \rhd \{\ TRUE\ \}\ )$
THEN
$On\_Block := \{\ blk\ \} \lhd On\_Block\ ||$
$Free\ (\ grp\ ) := FALSE\ ||$
$Clear := Clear \Leftarrow \{\ blk \mapsto FALSE\ ,\ On\_Block\ (\ blk\ ) \mapsto TRUE\ \}\ ||$
$Gripped\ (\ blk\ ) := grp$
END **Put_Block_On_Block** (*blk1* , *grp*) $\widehat{=}$
PRE
$grp = Tom \wedge$
$blk1 \in Block \wedge$
$Gripped\ (\ blk1\ ) = grp \wedge$
$Free\ (\ grp\ ) = FALSE$
THEN
$Free\ (\ grp\ ) := TRUE\ ||$
$Gripped := \{\}\ ||$

ANY *blk2*
WHERE
blk2 $ *in Block* $\wedge$
$Clear\ (\ blk2\ ) = TRUE$
THEN
$On\_Block\ (\ blk1\ ) := blk2\ ||$
$Clear := Clear \Leftarrow \{\ blk1 \mapsto TRUE\ \} \Leftarrow \{\ blk2 \mapsto FALSE\ \}$
END
END

Figure 5: Operations in B

the planner. However the equivalent of the 'Sussman Anomaly' configurations was achieved by commencing from the initial state and placing block 3 on block 1, as shown in the following animation (where * means the variable has changed):

```
*On_Block {block3 |-> block1}
 On_Table {block3 |-> FALSE ,
           block1 |-> TRUE ,
           block2 |-> TRUE ,

           block4 |-> TRUE ,
           block5 |-> TRUE ,
           block6 |-> TRUE ,
           block7 |-> TRUE}
*Clear    {block1 |-> FALSE ,
           block3 |-> TRUE ,
           block2 |-> TRUE ,

           block4 |-> TRUE ,
           block5 |-> TRUE ,
           block6 |-> TRUE ,
           block7 |-> TRUE}
*Gripped  {}
*Free     {Tom |-> TRUE}
```

The desired final state was achieved using the operations

```
Grip_Block_On_Block ( block3, Tom )
Put_Block_On_Table ( block3 )
Grip_Block_On_Table ( block2 , Tom )
Put_Block_On_Block ( block2 )
   (Local Variable blk2 in 'ANY' set to block3)
Grip_Block_On_Table ( block1 , Tom )
Put_Block_On_Block ( block1 )
```

(see Figure 5) with end state:

```
*On_Block {block2 |-> block3 ,
           block1 |-> block2}
 On_Table {block1 |-> FALSE ,
           block2 |-> FALSE ,
           block3 |-> TRUE, ..  }
*Clear    {block2 |-> FALSE ,
           block1 |-> TRUE ,
           block3 |-> FALSE, ..  }
*Gripped  {}
*Free     {Tom |-> TRUE}
```

Tyres World Case Study

We used a similar strategy (i.e. reverse engineering in modelling variables) when we modelled the 'Tyres World' in B. The Tyres world involves the changing of a faulty wheel using a wrench and a jack, both of which are (usually) initially in the car boot. Wheel changing involves loosening and removing wheel nuts, then changing the wheel. The wrench, jack and spare wheel must be available when required and the actions must take place in the correct order. The objective of the case study was (first) as a preliminary investigation into the relationship between B and OCL and (second) to test the adequacy of the validation tools (fully described in (West & Kitchin 2003)). The 'Tyres World' domain (Russell 1992) was chosen because, in the field of Planning, it is a well-known and well-used model that is unlikely to have any hidden errors. In the case of OCL, two wheels, hubs and their attached nuts were modelled plus a spare wheel in the car boot. The additional wheel (as compared with the 'usual' model in (Russell 1992)) was introduced so that extra validation checks could be introduced. In contrast, in the B model *four* wheels plus hubs and nuts were modelled, although as it turned out, two would have been sufficient. Actions in the OCL model include 'opening the car boot', 'loosening the nuts', 'removing the wheel' etc. and the B specification contained operations equivalent to these. Some exceptions were made where simplifications were possible in B; an example is the use of a single operation for 'fetching a tool'.

The approach was the deliberate introduction of equivalent errors into both the B and OCL models to see if the use of the stepper/animator and validation checks and proof tool, would identify these faults. Various tasks were tried out in GIPO using both the stepper and planning engines to see if errors and inconsistencies in the domain model were detected, and to compare its performance with that of the B-Toolkit. Validation checks in GIPO include checks on operators and checks on tasks. Thus operators must consist of legal expressions with respect to invariant expressions on the individual sorts; and initial states and goal expressions must likewise consist of a legal substate expressions.

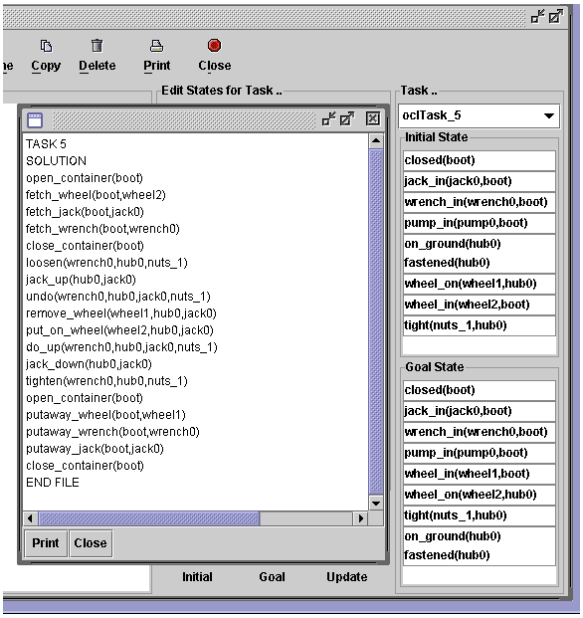


Figure 7: Task and Plan for changing a wheel

A screen-shot of GIPO (Figure 7) shows a plan generation for a task: initially all the tools and spare wheel (wheel2) are in the car boot and the goal is a change of wheel (wheel1). The next section describe the errors introduced and the results of validation for each of the two tools.

Challenging the Models

Errors in the initial state We introduced errors such as the obviously incorrect state of two wheels on a single hub. The result for the OCL model was that GIPO did not object to this inconsistent initial state - no errors were found

by the validation checks. When tried with a planner (FF (Hoffmann 2000)), it reported that the goal was impossible. Another initial state was created in which the nuts were on one of the hubs but no wheel was on that hub. The other hubs were in a correct state. This inconsistency is not found by the validation checks in GIPO. The user can find it by using the stepper or using a planner that incorporates such checks: FF for example reports that the goal is impossible. This uncovered a simple insecurity in the GIPO tool itself - although individual sort invariants were actively used in its tools, the general domain invariants were not being used to check state validity.

The errors were introduced in the B model by altering the initial state. Both of these errors introduced inconsistency in the B specification - of the initial state with respect to the invariant. They were discovered by the B-Toolkit most simply by using the animator. However the proof tool also provided a check.

Errors in setting tasks Errors were introduced in the OCL task description within GIPO. These involved an attempt to reach an illegal state (i.e. one which was inconsistent with the domain model). One example involved jacking up two hubs using the same jack. Again the validation checks of GIPO don't report an impossible task. When tested, the FF planner very quickly says the problem is proven unsolvable. It was subsequently discovered that the domain model in OCL did not contain the 'inconsistency constraint' in the prohibition of the jack on two different hubs.

Of course there is no equivalent function of 'setting a task' in B so in order to correspond with the GIPO task, a single operation, of attempting to jack up a wheel where the jack is already in use on a different wheel was tried using the animator. However the invariant of the Tyres World B specification was such that there is a 1-1 relationship between a wheel hub and a jack - and the operation 'JackUpCar' supported this in its precondition that the jack should not be in use already and an error was generated.

Errors in pre- and postconditions Prevail conditions in OCL are pre-conditions that persist - that is, the object concerned does not change state during the operation. An example of this would be the prevail condition *have_wrench* of the *loosen* operator: in order to loosen the nuts we must have the wrench - and we will still have the wrench after the nuts have been loosened. We removed the *have_wrench* prevail condition from the *loosen* operator. This error, as expected, was not detected by validation checks, but became apparent when using GIPO's stepper. The operator was not able to be applied because the wrench was not available. This type of error does not affect overall consistency of the domain model, but is just concerned with a specific object being part of a particular operation.

In the case of the B model there was no problem with contravention of the pre-condition and no problem with the invariant. (Note that the OCL prevail condition can be modelled as a pre-condition in B as, by default, any variable for which there is no substitution does not change.) The error is only demonstrated by the showing of a 'silly' result in that the wrench is still in the car boot. This is a 'domain-specific' error which could only have been demonstrated by animation.

Because of the manner in which actions are described in OCL - by the change in state of individual objects - it was similarly not possible for a 'post' condition to be removed on its own. For example: for the operator *fetch_jack* the jack object changes state from being in the car boot (precondition for the transition) to being available for use (post-condition for the transition). In an attempt to introduce this kind of error, we removed the transition for the 'jack' object from the 'fetch jack' action. The error became apparent when using the stepper.

The experiments also uncovered a previously unknown omission in the B model - a missing precondition for putting away the tyre.

Comparison of Operations

Two operations in B are compared with the equivalent actions in the OCL model.

Gripping a Block from the Table If we compare the operation *Grip_Block_On_ Table* in B with its equivalent in OCL (Figures 5 and 2) we see the same pre and postconditions. However they are structured differently in OCL where the precondition for each object is an appropriate substate from the substate classes:

1. The precondition that the block is on the table and clear becomes the left hand side of the necessary condition for the block.
2. The precondition that the gripper is free becomes the left hand side of the necessary condition for the gripper.
3. The substitution in B for the block, that it is gripped, becomes the right hand side of the necessary condition for the block. However in B it is stated explicitly that the gripped block is no longer on the table and not clear. In OCL this is assumed from the substate.
4. The substitution in B that gripper is in use becomes the right hand side of the necessary condition for the gripper.

Gripping a Block from a Block Comparing B and OCL versions of 'gripping block1 from block2', (Figures 5 and 3) we see that there is a change in the state of both blocks after the operation. For B, it is enough to state that block2 is now clear. However for OCL this is not enough and we must distinguish between 2 substates - where block2 is on the table and where block2 is on another block. Since we have made a change in the state of block2 we must be precise about the whole of its state. The different outcomes for block2 give rise to the two actions in OCL. As in the previous operation, what is implicit in OCL must be explicit in B. Thus we must say that the gripped block is no longer free.

A Comparison of the use of B and OCL to acquire planning domain knowledge

Here we summarise the results of our comparison:

1. The B language and toolkit is an industrial strength formal specification and development method, whereas OCL is a tool used in research and education. The exercise showed some problems with GIPO - in particular that its static validation checks should be extended to test the consistency of the initial state and goal expressions against the general domain invariants.

2. B allows the user to encode more precise details about the relations in the domain than GIPO - they can be be relations, 1-1 functions etc. This level of precision is certainly not available in most planning languages, and is attractive in safety-related applications.

3. As OCL is aimed specifically at planning, it has inbuilt structures and mechanisms that anticipate the entities that are to be represented. This makes the encoding rather more compact than the B specification. Encoding in B, as one would expect from a general language, one is left with more choices and decisions in the encoding process. The correspondence we used in our case reduced the encoding choices, but still the B encoding is rather 'flat' in that the invariant list contains all predicate and invariant information.

4. Both languages assume default persistence and a closed world. The differences in this respect are subtle, in that in B a variable involved in a precondition remains unaltered by default. However in OCL a 'prevail' is required if precondition variables are unchanged.

5. Regarding validation and debugging, both languages have effective, automated tool support which performs validation/consistency checks and identifies the presence of bugs. Not surprisingly, the B toolkit was more reliable at finding inconsistencies in some cases, as it demands a more detailed specification.

To further explore the comparison, we show how, in the Blocks world, the OCL specification of substates can be derived from the B invariant conjuncts (1–16) in Figure 4. We observe that the following three sets are disjoint and 'partition' Block:

$$\begin{array}{l}\mathrm{dom}(\mathit{Gripped}),\\ \mathrm{dom}(\mathit{On\_Table}) \rhd \{\mathit{TRUE}\},\\ \mathrm{dom}(\mathit{On\_Block})\end{array}$$

Although the partition is not the same as the substate partition in OCL, it is possible to obtain an equivalent partition if we first note that:
$\mathrm{dom}(\mathit{Clear} \rhd \{\mathit{TRUE}\})$ and $\mathrm{dom}(\mathit{Clear} \rhd \{\mathit{FALSE}\})$ partition *Block* and we have

$$\begin{array}{l}\mathrm{dom}\,\mathit{On\_Table} \rhd \{\mathit{TRUE}\} = \mathrm{dom}\,\mathit{On\_Table} \rhd \{\mathit{TRUE}\} \cap\\ \qquad (\mathrm{dom}\,\mathit{Clear} \rhd \{\mathit{TRUE}\}) \cup (\mathrm{dom}\,\mathit{Clear} \rhd \{\mathit{FALSE}\})\end{array}$$

and from (10) the partition now becomes:

$$\begin{array}{l}\mathrm{dom}\,\mathit{Gripped},\\ (\mathrm{dom}\,\mathit{On\_Table} \rhd \{\mathit{TRUE}\}) \cap (\mathrm{dom}\,\mathit{Clear} \rhd \{\mathit{TRUE}\}),\\ (\mathrm{dom}\,\mathit{On\_Table} \rhd \{\mathit{TRUE}\}) \cap (\mathrm{dom}\,\mathit{Clear} \rhd \{\mathit{FALSE}\}),\\ \mathrm{dom}\,\mathit{On\_Block}.\end{array}$$

This can be made explicit and in a similar format to the OCL version using conjuncts from the invariant. For example from (2):

$$\begin{array}{l}\mathrm{dom}\,\mathit{On\_Block} \cap (\mathrm{dom}(\mathit{Clear} \rhd \{\mathit{FALSE}\}) =\\ \mathrm{dom}\,\mathit{On\_Block} \cap (\mathrm{dom}(\mathit{Clear} \rhd \{\mathit{FALSE}\}) \cap \mathit{Block} =\\ \mathrm{dom}\,\mathit{On\_Block} \cap (\mathrm{dom}(\mathit{Clear} \rhd \{\mathit{FALSE}\}) \cap\\ (\mathrm{dom}\,\mathit{On\_Table} \rhd \{\mathit{FALSE}\} \cup \mathrm{dom}\,\mathit{On\_Table} \rhd \{\mathit{TRUE}\}) =\\ \mathrm{dom}\,\mathit{On\_Block} \cap (\mathrm{dom}(\mathit{Clear} \rhd \{\mathit{FALSE}\})\\ \cap(\mathrm{dom}\,\mathit{On\_Table} \rhd \{\mathit{FALSE}\}\end{array}$$

In a similar manner using other conjuncts, this set can be equivalently expressed:

$$\begin{array}{l}\mathrm{dom}\,\mathit{On\_Block} \cap (\mathrm{dom}(\mathit{Clear} \rhd \{\mathit{FALSE}\})\\ \qquad \cap(\mathrm{dom}\,\mathit{On\_Table} \rhd \{\mathit{FALSE}\}\\ \qquad \cap\ (\mathit{Block} - \mathrm{dom}\,\mathit{Gripped})\end{array}$$

This, with (6) can be expanded out:

$$\begin{array}{l}\forall\, \mathit{blk}1 \in \mathit{Block}.(\exists\, \mathit{blk}2 \in \mathit{Block}.(\mathit{On\_Block}(\mathit{blk}1) = \mathit{blk}2\\ \wedge\ \mathit{blk}1 \neq \mathit{blk}2 \wedge \mathit{Clear}(\mathit{blk}1) = \mathit{FALSE}))\\ \ldots\end{array}$$

which is equivalent to the substate

```
[on_block(B,B1),ne(B,B1)],
```

given the local closed world assumption.

Conclusions

In this paper we have investigated the use of a formal method to capture planning domain models. We have compared the method (together with its commercially-available tool support) with a planning-oriented method. The comparison shows a remarkable similarity between the two. The advantages in using a method such as B are that it is mathematically based so that formal reasoning can be used to deduce desirable (and potentially undesirable) properties. Support for the method is available via tools - such as the Toolkit. However, the disadvantages are that there are no special planning - oriented features, and that the B specification, once validated, would have to be translated into a more planner-friendly language in order to be used with current planning engines.

References

B-Core (UK) Ltd. http://www.b-core.com/.

Hoffmann, J. 2000. A Heuristic for Domain Independent Planning and its Use in an Enforced Hill-climbing Algorithm. In *Proceedings of the 14th Workshop on Planning and Configuration - New Results in Planning, Scheduling and Design*.

McCluskey, T. L., and Porteous, J. M. 1997. Engineering and Compiling Planning Domain Models to Promote Validity and Efficiency. *Artificial Intelligence* 95:1–65.

Munoz-Avila, H. M.; Aha, D. W.; Nau, D.; Weber, R.; Breslow, L.; and Yaman, F. 2001. SiN: Integrating case-based reasoning with task decomposition. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, 999–1004.

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

Russell, S. 1992. Efficient memory-bounded search algorithms. In *Proc. ECAI*.

Schneider, S. 2001. *The B-Method: An Introduction*. Palgrave.

Simpson, R. M., and McCluskey, T. L. 2003. Plan Authoring with Continuous Effects. In *Proceedings of the 22nd UK Planning and Scheduling Workshop (PLANSIG-2003), Glasgow, Scotland*.

Simpson, R. M.; McCluskey, T. L.; Zhao, W.; Aylett, R. S.; and Doniat, C. 2001. GIPO: An Integrated Graphical Tool to support Knowledge Engineering in AI Planning. In *Proceedings of the 6th European Conference on Planning*.

West, M. M., and Kitchin, D. E. 2003. Testing Domain Model Validation Tools. In *Proceedings of the 22nd Workshop of the UK Planning and Scheduling SIG, Glasgow*.

Probabilistic Monitoring from Mixed Software and Hardware Specifications

Tsoline Mikaelian, Brian C. Williams, Martin Sachenbacher
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
32 Vassar St. Room 32-275, Cambridge, MA 02139
{tsoline, williams, sachenba}@mit.edu

Abstract

We introduce a capability for online monitoring and diagnosis of stochastic systems with complex behavior. Our work complements offline verification techniques for embedded systems. In most complex systems today, hardware is augmented with software functions that influence the system's behavior. In this paper hardware models are extended to include the behavior of associated embedded software, resulting in more comprehensive estimates of a system's state trajectories. Capturing the behavior of software is much more complex than that of hardware due to the potentially enormous state space of a program. This complexity is addressed by using probabilistic, hierarchical, constraint-based automata (PHCA) that allow the uniform and compact encoding of both hardware and software behavior. We introduce a novel approach that frames PHCA-based diagnosis as a soft constraint optimization problem over a finite time horizon. The problem is solved using efficient, decomposition-based optimization techniques. The solutions correspond to the most likely evolutions of the software-extended system.

Introduction

Traditionally, model-based verification of embedded systems has focused on determining program correctness using techniques such as symbolic model checking (J. R. Burch & Hwang 1992). However, verification is performed offline during design and development, and is not guaranteed to verify against all possible system failures. To complement offline verification techniques, we introduce a novel capability for online monitoring and diagnosis of systems with complex, non-deterministic behavior. While verification techniques typically result in counterexamples, monitoring and diagnosis result in estimates of the system's state trajectories.

Model-based monitoring has mainly operated on hardware systems (de Kleer & Williams 1987; Dressler & Struss 1996). For instance, given an observation sequence, the Livingstone (Williams & Nayak 1996) diagnostic engine estimates the state of hardware components based on hidden Markov models that describe each component's behavior in terms of nominal and faulty modes. Researchers at the other end of the spectrum have applied model-based diagnosis to software debugging (Mayer & Stumptner 2004). This paper explores the middle ground between the two, in particular the online monitoring and diagnosis of systems with combined hardware and software behavior.

Many complex systems today, such as spacecraft, robotic networks, automobiles and medical devices consist of hardware components whose functionality is extended or controlled by embedded software. Examples of devices with software-extended behavior include a communications module with an associated device driver, and an inertial navigation unit with embedded software for trajectory determination. The embedded software in each of these systems interacts with the hardware components and influences their behavior. In order to correctly estimate the state of these devices, it is essential to consider their software-extended behavior.

As an example of a complex system, consider vision-based navigation for an autonomous rover exploring the surface of a planet. The camera used within the navigation system is an instance of a device that has software-extended behavior: the image processing software embedded within the camera module augments the functionality of the camera by processing each image and determining whether it's corrupt. A sensor measuring the camera voltage may be used for estimating the physical state of the camera. A hardware model of the camera describes its physical behavior in terms of inputs, outputs and available sensor measurements. A diagnosis engine such as Livingstone that uses only hardware models will not be able to reason about a corrupt image. The embedded software provides additional information on the quality of the image that is essential for correctly diagnosing the navigation system. To see why this is the case, consider a scenario in which the camera sensor measures a zero voltage. Based solely on hardware models of the camera, the measurement sensor and the battery, the most likely diagnoses will include camera failure, low battery voltage and sensor fault. However, given a software-extended model of the camera that models the process of obtaining a corrupt image, the diagnostic engine may use the information on the quality of the image. Knowing that the processed image is not corrupt, the most likely diagnosis that the measurement sensor is broken may be deduced.

The above scenario demonstrates that a monitoring engine for complex systems with software-extended behavior must: 1) monitor the behavior of both the hardware and its embedded software so that the software state can be used for di-

agnosing the hardware, and 2) reason about the system state given delayed symptoms. An instance of a delayed symptom is the image quality determined by the camera software after it has completed all stages of image processing.

In this paper we introduce a novel model-based monitoring and diagnostic system that operates on software-extended behavior models, to meet requirements 1) and 2) listed above. In contrast to previous work on model-based verification and software debugging (Mayer & Stumptner 2004), the purpose of this work is to leverage information within the embedded software to refine the estimates of physical systems. As such, we are not addressing the problem of diagnosing software bugs. Without loss of generality, we assume that software bugs discovered at runtime are handled by a separate exception handling mechanism.

First, we address modeling issues. Capturing the behavior of software is much more complex than that of hardware due to the hierarchical structure of a program and the potentially large number of its execution paths. We address this complexity by using probabilistic, hierarchical, constraint-based automata (PHCA) (Williams, Chung, & Gupta 2001) that can uniformly and compactly encode both hardware and software behavior. Building upon our previous work, we introduce a novel capability for monitoring systems with software-extended behavior in the presence of delayed symptoms. While Livingstone-2 (L2) (Kurien & Nayak 2000) handles delayed symptoms for diagnosing hardware systems, our approach generalizes this capability to software-extended behavior by posing the PHCA-based diagnosis problem over a finite time horizon. We frame diagnosis as constraint optimization problem based on soft constraints that encode the structure and semantics of PHCA. The problem is solved using efficient, decomposition-based optimization techniques, resulting in the most likely estimates of the software-extended system.

Modeling Software-Extended Behavior

Figure 1 shows the software-extended camera module for the vision-based navigation scenario described above. In this example, the failure probabilities for each of the battery, camera and sensor are 10%, 5% and 1% respectively. A typical behavioral model of the camera is shown on the left of Figure 2. The camera can be in one of 3 modes: on, off or broken. The hardware behavior in each of the modes is specified in terms of inputs to the camera such as the power and the behavior of camera components such as the shutter. The broken mode is unconstrained in order to accommodate novel types of failures. Mode transitions can occur proba-

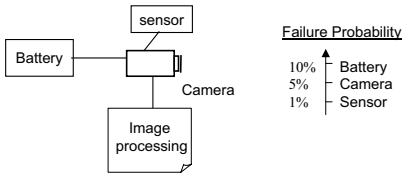


Figure 1: Camera Module for Navigation System

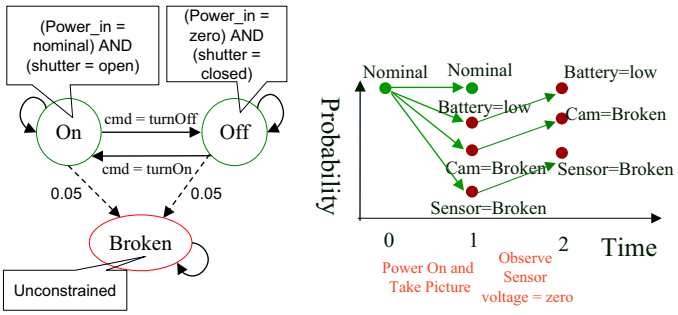


Figure 2: *left:* Behavior Model for the Camera Component. *right:* Most likely diagnoses of the camera module based on hardware component models. Nominal state = no failures.

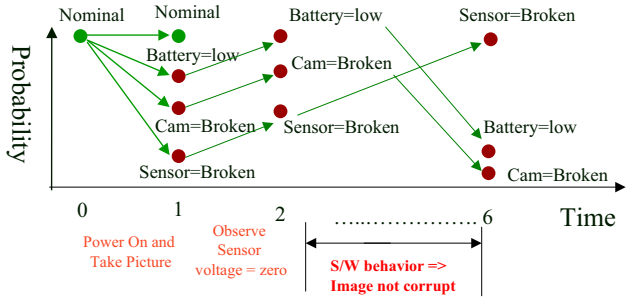


Figure 3: Most likely diagnoses of the camera module based on the software-extended behavior models.

bilistically, or as a result of issued commands. The battery and the sensor components can be modeled in a similar way. For the scenario introduced above, the most likely diagnoses of the module can be generated based on the hardware models alone, as shown on the right of Figure 2. However, the image processing software provides extended functionality that is not described by the model in Figure 2. The specification of the embedded software can offer important evidence that substantially alters the diagnosis. A sample specification of the behavior of the image processing software may take the following form:

> *If an image is taken by the camera, process it to determine whether it's corrupt. If the image is corrupt, discard it and reset the camera; retry until a non-corrupt image is obtained for navigation. Once a high quality image is stored, wait for new image request from navigation unit.*

Such a specification abstracts the behavior of the image processing software implemented in an embedded programming language such as Esterel (Berry & Gonthier 1992) or RMPL (Williams, Chung, & Gupta 2001). For the above scenario, the behavior of the embedded software provides diagnostic information necessary to correctly estimate the state of the camera module. Given that the image is not corrupt, the possibility that the camera is broken becomes very unlikely. This is illustrated in Figure 3.

Unlike a hardware component that can typically be described by a single mode of behavior, monitoring soft-

ware behavior necessitates tracking simultaneous hierarchical modes. A modeling formalism that will allow the specification of software behavior must support: 1) full concurrency for modeling sequential and parallel threads of behavior, 2) conditional behavior, 3) iteration, 4) preemption, 5) probabilistic behavior for modeling uncertainty and 6) propositional logic constraints for specifying co-temporal relationships among variables. The following section reviews the modeling framework for handling these requirements.

Probabilistic, Hierarchical Constraint-based Automata (PHCA)

Probabilistic, hierarchical, constraint-based automata (PHCA) were introduced in (Williams, Chung, & Gupta 2001) as a compact encoding of Hidden Markov Models (HMMs), for modeling complex systems.

Definition 1 (PHCA)
A PHCA is a tuple $< \Sigma, P_\Theta, \Pi, O, C, P_T >$, where:

- Σ is a set of locations, partitioned into primitive locations Σ_p and composite locations Σ_c. Each composite location denotes a hierarchical, constraint automaton. A location may be marked or unmarked. A marked location represents an active branch.
- $P_\Theta(\Theta_i)$ denotes the probability that $\Theta_i \subseteq \Sigma$ is the set of start locations (initial state). Each composite location $l_i \subseteq \Sigma_c$ may have a set of start locations that are marked when l_i is marked.
- Π is a set of variables with finite domains. $C[\Pi]$ is the set of all finite domain constraints over Π.
- $O \subseteq \Pi$ is the set of observable variables.
- $C : \Sigma \rightarrow C[\Pi]$ associates with each location $l_i \subseteq \Sigma$ a finite domain constraint $C(l_i)$.
- $P_T(l_i)$, for each $l_i \subseteq \Sigma_p$, is a probability distribution over a set of transition functions $T(l_i) : \Sigma_p^{(t)} \times C[\Pi]^{(t)} \rightarrow 2^{\Sigma^{(t+1)}}$. Each transition function maps a marked location into a set of locations to be marked at the next time step, provided that the transition's guard constraint is entailed.

Definition 2 (PHCA State)
The state of a PHCA at time t is a set of marked locations called a marking $m^{(t)} \subset \Sigma$.

Figure 4 shows a PHCA model of the camera module in Figure 1. The "On" composite location contains three subautomata that correspond to primitive locations "Initializing", "Idle" and "Taking Picture". Each composite or primitive location of the PHCA may have behavioral constraints. The behavioral constraint of a composite location, such as $(power\_in = nominal)$ for the "On" location, is inherited by each of the subautomata within that composite hierarchy. In addition to the physical camera behavior, the model incorporates qualitative software behavior such as processing the quality of an image. Furthermore, based on the image quality, the possible camera configurations may be constrained

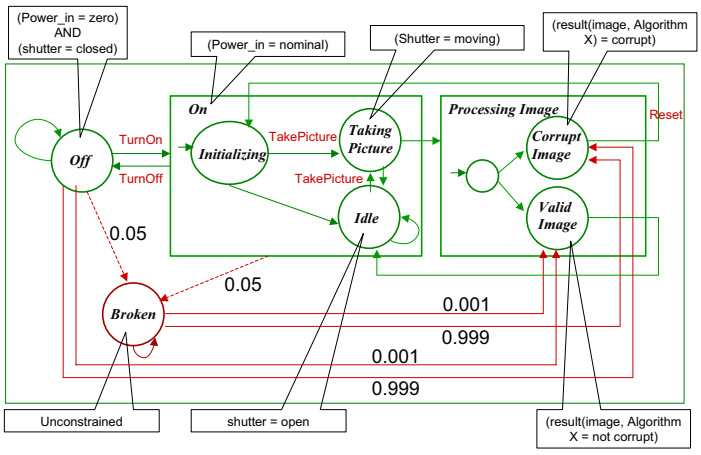


Figure 4: PHCA model for the camera/image processing module. Circles represent primitive locations, boxes represent composite locations and small arrows represent start locations.

by the embedded software. For example, if the image is determined to be corrupt, the software attempts to reset the camera. This restricts the camera behavior to transition to the Initializing location.

Recall that Figure 3 shows the most likely state trajectories based on the software-extended PHCA model. At time step 2, as the sensor measurement indicates zero voltage, the most likely diagnosis trajectories are 1) battery = low with 10% probability, 2) camera = broken with 5% probability and 3) sensor is broken with 1% probability. For the first trajectory that indicates that the battery is low, the power to the camera is not nominal, hence the camera will stay in the "Off" location. For the second trajectory, the camera will be in the "Broken" location. For the third trajectory that indicates that the sensor is broken, the power input to the camera will be unconstrained, and hence the PHCA state of the camera may include a marking of the "On" location. Although the evolutions of this third trajectory have an initially low probability of 1%, at time step 6 they become more likely than the others as the embedded software determines that the image is valid. The reason is because the second most likely trajectory at time 2 with camera = "Broken" location marked has a 0.001 probability of generating a valid image, thus making the probability of that trajectory 0.005% at time 6. This latter trajectory is less probable than those trajectories stemming from the sensor being broken with 1% probability. Similarly, the first trajectory with battery = low and camera = Off becomes less likely at time step 6 as there is 0.001% probability of processing a valid image while the camera is "Off".

PHCA models have the following advantages that support their use for diagnosing systems with software-extended behavior. First, since HMMs may be intractable, PHCA encoding is essential to support real-time, model-based deduction. Second, PHCAs provide the expressivity to model the behavior of embedded software by satisfying requirements 1)-6) above. Third, the hierarchical nature of the automata

enables modeling of complex concurrent and sequential behaviors, similar to hierarchical Statecharts (Harel 1987). As an example of concurrency, the PHCA in Figure 4 allows the simultaneous marking of the "On" location of the camera, as well as the "Initializing", "Idle", or "Taking Picture" locations. This is in contrast to diagnosis based on non-hierarchical models that can estimate each component to be in a single mode of operation. State estimates of components may be required at different levels of granularity. For example, an image-based navigation function may require high level camera state estimates such as "On" or "Off". On the other hand, a function that coordinates imaging activities may need more detailed camera state estimates such as "Initializing" or "Taking Picture". Simultaneous marking of several camera locations such as "On" and "Initializing", allows their use within functions that require estimates at different levels of granularity.

The following sections introduce a novel diagnostic system based on the PHCA modeling framework. We first introduce our approach for diagnosis over a single time step, and then extend it to handle delayed symptoms. Our approach results in a capability for diagnosing systems with software-extended behavior in the presence of delayed symptoms. Furthermore, our formulation of the diagnosis problem enables the use of powerful decomposition techniques for efficient solution extraction.

Diagnosis as Constraint Optimization based on PHCA Models

We frame diagnosis based on PHCA models as a soft constraint optimization problem (COP) (Schiex, Fargier, & Verfaillie 1995). The COP encodes the PHCA models as probabilistic constraints, such that the optimal solutions correspond to the most likely PHCA state trajectories. The soft constraint formulation allows a separation between probability specification and variables to be solved for. Thus, we can associate probabilities with constraints that encode transitions, while solving for state variables.

Definition 3 (Constraint Optimization Problem) A constraint optimization problem (COP) is a triple (X, D, F) where $X = \{X_1, ..., X_n\}$ is a set of variables with corresponding set of finite domains $D = \{D_1, ..., D_n\}$, and $F = \{F_1, ..., F_n\}$ is a set of preference functions $F_i : (S_i, R_i) \rightarrow C_i$ where (S_i, R_i) is a constraint and C_i is a set of preference (or cost) values. Each constraint (S_i, R_i) consists of a scope $S_i = \{X_{i1}, ..., X_{ik}\}$ representing a subset of variables X, and a relation $R_i \subseteq D_{i1} \times ... \times D_{ik}$ on S_i that defines all tuples of values for variables in S_i that are compatible with each other. Each preference function F_i maps the tuples of (S_i, R_i) to values C_i. The solution to variables of interest (solution variables) $Y \subseteq X$ is an assignment to Y that is consistent with all constraints, has a consistent extension to all variables X, and minimizes (or maximizes) a global objective function defined in terms of preference functions F_i.

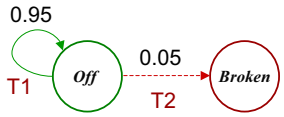
Given a PHCA state at time t and an assignment to observable and command variables in Π (see Definition 1) at times t and $t+1$, in order to estimate PHCA state at time $t+1$, we encode both the structure and execution semantics of the PHCA as a COP, consisting of:

- Set of variables $X_\Sigma \cup \Pi \cup X_{Exec}$, where $X_\Sigma = \{L_1, ..., L_n\}$ is a set of variables that correspond to PHCA locations $l_i \in \Sigma$, Π is the set of PHCA variables, and $X_{Exec} = \{E_1, ..., E_n\}$ is a set of auxiliary variables used for encode the execution semantics of the PHCA.
- Set of finite, discrete-valued domains $D_{X_\Sigma} \cup D_\Pi \cup D_{X_{Exec}}$, where $D_{X_\Sigma} = \{Marked, Unmarked\}$ is the domain for each variable in X_Σ, D_Π is the set of domains for PHCA variables Π, and D_{Exec} is a set of domains for variables X_{Exec}.
- Set of constraints R that include the behavioral constraints associated with locations within the PHCA, as well as encoding of the PHCA execution semantics.
- Preferences in the form of probabilities associated with tuples of constraints R. Tuples of hard constraints that are disallowed by the constraint are assigned probability 0.0, while the tuples allowed by the constraint are assigned probability 1.0. Tuples of soft constraints are mapped to a range of probability values based on the PHCA model. These probability values reflect the probability distribution P_Θ of PHCA start states and probabilities associated with PHCA transitions P_T.
- The optimal solution to the COP is an assignment to solution variables X_Σ that represent the state of the PHCA, while maximizing the probability of the transitions that lead to that state from the previous time step. This corresponds to a state assignment that maximizes the product of the probabilities of the enabled constraint tuples.

A key to framing PHCA-based diagnosis as COP is the formulation of the constraints R that capture the execution semantics of the PHCA. PHCA execution involves determining the entailment of behavioral constraints, identifying enabled transitions from a current PHCA state, and taking those transitions to determine the next state. Referring back to the PHCA example in Figure 4, if we assume that at time t the PHCA state is $< On < Idle >>$ and that the transition guard constraint $(command = TakePicture)$ is entailed, and at time t+1 the behavioral constraint $(shutter = moving)$ of the transition's target location is entailed, then the PHCA state at time t+1 will be $< On < TakingPicture >>$. To encode entailment of conditions such as $(command = TakePicture)$, a variable E_T is introduced with domain $\{Entailed, Not{-}Entailed\}$ to denote whether the transition guard condition is entailed. Entailment of a condition is then formulated as a COP constraint that allows the assignment $E_T = Entailed$ to be associated with tuples that list all possible assignments to the variable $command$ that entail the condition $(command = TakePicture)$. Entailment constraints are generated for all locations that have behavioral constraints and for all transitions that have guard constraints.

The following example on the left of Figure 5 shows a probabilistic choice between two transitions for a section of

the PHCA in Figure 4. In order to encode this probabilistic choice, we first introduce a location variable $X_{Off}^{(t)}$ for time t, with domain $\{Marked, Unmarked\}$. Then auxiliary variables $E_{T1}^{(t)}$ and $E_{T2}^{(t)}$ with domain $\{Enabled, Disabled\}$ are introduced for transitions T1 and T2 respectively.



$X_{Off}^{(t)}$	$E_{T1}^{(t)}$	$E_{T2}^{(t)}$	Prob.
Marked	Enabled	Disabled	0.95
Marked	Disabled	Enabled	0.05
Unmarked	Disabled	Disabled	1.0

Figure 5: *left:* PHCA with two probabilistic transitions. *right:* Probabilistic transition constraint.

The COP constraint that encodes the probabilistic choice among the two transitions T1 and T2 is formulated logically:
$X_{Off}^{(t)} = Marked \equiv (\exists\ T \in \{T1, T2\}\ |\ :\ E_T^{(t)} = Enabled \wedge (\forall\ T' \in \{\{T1, T2\} - T\}\ |\ : E_{T'}^{(t)} = Disabled))$
$\bigwedge\ X_{Off}^{(t)} = Unmarked \equiv (\forall\ T \in \{T1, T2\}\ |\ :\ E_T^{(t)} = Disabled)$

This logical formula is compiled into a set of tuples with associated probability values, as shown in Figure 5 (*right*). The tuples are mapped to probability values by the following preference function:

$$F_T = \begin{cases} Prob(T_i) & if\ (\exists T_i^{(t)} : E_{T_i}^{(t)} = Enabled) \\ 1.0 & otherwise \end{cases}$$

The above constraint identifies the enabled transition, but does not encode taking the transition. In general, the following constraint encodes taking enabled transitions, unless the behavior constraint of the transition's target location is not entailed:
$(\forall\ L \in \Sigma\ |\ :\ ((\exists\ \tau \in \{T | Target(T) = L\}\ |\ : E_\tau^{(t-1)} = Enabled) \wedge Behavior_L^{(t)} = Entailed) \Rightarrow X_L^{(t)} = Marked)$

where E_τ represents a transition variable, $Behavior_L$ is an entailment variable for the behavior constraints of location L $\cup$ its composite parent if L is within a hierarchy, and X_L is the location variable of L. The constraint is instantiated for each location of the PHCA, as indicated by $\forall\ L \in \Sigma$.

Some semantic rules apply to PHCA hierarchies. For example, when a composite location becomes marked, all of its start locations become marked. Since "Initializing" is a start location of the composite "On" location, a PHCA in state $< Off >$ may transition to state $< On < Initializing >>$. Furthermore, a composite location should be marked if any of its subautomata are marked. The COP constraints must correctly capture such PHCA semantics and encode mutual exclusions to avoid interference and conflicting effects among the constraints. For brevity, the complete encoding of constraints is not presented.

The formulation of diagnosis as COP is performed offline. Given a PHCA, we have implemented a compiler that automatically generates the corresponding COP. The COP is then used in an online solution phase by dynamically updating it to incorporate constraints on new observations and issued commands. The solutions to the COP can be generated up to a given probability threshold using a constraint optimization solver for soft constraints (Sachenbacher & Williams 2004). The solutions incorporate the probability distribution on the initial states as encoded by the COP. The most likely solutions generated at a time step t dynamically update the COP to constrain the set of start states for solving the COP at time step t+1. For example, as Figure 3 shows, state estimates at time 2 may only be reached through those at time 1. Thus limiting the number of state trajectories maintained at each time step has implications for diagnosing faults that manifest delayed symptoms.

Diagnosis with Delayed Symptoms

Ideally, diagnosis will maintain a complete probability distribution of all possible system states. However, maintaining all possible state trajectories at each time step is intractable because of exponential growth in state space. Thus at every time step a limited number of trajectories are typically maintained. A potential problem with this approach is that it may miss the best diagnosis if a trajectory through a pruned state that is initially very unlikely becomes very likely after additional evidence. Figure 6 illustrates this situation for the camera module, where the initially unlikely state $(Sensor = Broken)$ is pruned, resulting in the best diagnosis to be unreachable when additional evidence is available at time 6.

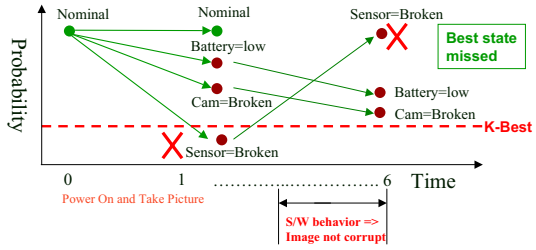


Figure 6: Missed diagnosis as a result of tracking a limited number of trajectories (K-Best)

Dealing with delayed symptoms is particularly important for diagnosing systems with software-extended behavior, due to typically delayed observations associated with software processing. Livingstone-2 (L2) (Kurien & Nayak 2000) addresses the problem of delayed symptoms for diagnosing hardware systems. We generalize the L2 capability to PHCA-based diagnosis.

We extend our COP formulation of PHCA-based diagnosis to provide flexibility for regenerating the most likely diagnoses over a finite time horizon rather than a single previous step. Thus, we frame the COP over a finite time horizon (N-stages) and leverage the N-stage history of observations and issued commands to generate the most likely diagnosis trajectories over the horizon. This involves augmenting the COP in the previous section to include model variables and constraints for each time step within the N-stage horizon. The solutions to the COP become assign-

ments to location variables $X_{\Sigma}^{(t)}$, t $\in \{0..N\}$, representing PHCA state trajectories that have maximum probability within the horizon. This probability corresponds to the product of transition probabilities enabled within that trajectory, multiplied by the probability of the initial state of the trajectory. As time progresses during the online solution phase, the N-stage horizon is shifted from $(t \rightarrow t+N)$ to $(t+1 \rightarrow t+N+1)$ and the COP over the new horizon is dynamically updated by constraining its start states at time t+1 to match the solutions from the previous iteration. This reformulation still limits the number of trajectories tracked to a given probability threshold, as described in the previous section. Referring to Figure 6, if we consider a time horizon $(0 \rightarrow 6)$, diagnosis trajectories will be regenerated starting from the $(Nominal)$ state at time 0. Therefore, even though the number of trajectories is limited, the trajectory ending at state $(Sensor = Broken)$ at time 6 will have the highest probability based on the delayed observation. Consequently, the state $(Sensor = Broken)$ at time 2 will be maintained because it is part of the most likely trajectory at time 6.

Decreasing the probability threshold for the trajectories being tracked solves the delayed-symptom problem by maintaining a larger number of states at each time step. However, for a system with many combinations of similar failure states with high probability, the number of trajectories maintained will have to be very large in order to be able to account for a delayed symptom that supports an initially low probability state. For such systems, considering even a small number of previous time steps gives enough flexibility to regenerate the correct diagnosis.

Implementation and Discussion

The PHCA model-based monitoring capability, described above, has been implemented in C++. Figure 7 shows the offline compilation phase and the online solution phase of the diagnosis process.

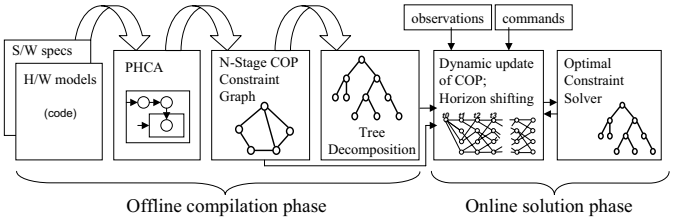


Figure 7: Process diagram for PHCA-based diagnosis

In the offline phase, the N-Stage COP is generated automatically, given a PHCA model and parameter N. To enhance the efficiency of the online solution phase, tree decomposition (Gottlob, Leone, & Scarcello 2000) is applied to decompose the COP into independent subproblems. This enables backtrack-free solution extraction during the online phase (Dechter 2003). In our implementation, the COP is decomposed using a tree decomposition package that implements bucket elimination (Kask, Dechter, & Larrosa 2003).

The online monitoring and diagnosis process uses both the COP and its corresponding tree decomposition. The online phase consists of a loop that shifts the time horizon, updates and solves the COP at each iteration. The COP is updated by incorporating new observations and commands, and constraining the start states to track the trajectories obtained within the previous horizon. At each iteration of the loop, the updated COP is solved using an implementation of the decomposition-based constraint optimization algorithm in (Sachenbacher & Williams 2004) that can generate diagnoses up to a given probability threshold.

For the camera model with $N = 2$, the COP has ~ 150 variables and ~ 100 constraints and is solved online in ~ 1 sec, resulting in more comprehensive diagnoses than previous hardware models. Future work includes evaluating the efficiency of the COP formulation using several complex scenarios, optimizing the COP formulation by minimizing the number of variables and constraints generated, investigating the optimal size of the diagnosis horizon and its relationship to the number of trajectories tracked.

Acknowledgments

This research is sponsored in part by NASA CETDP under contract NNA04CK91A and by the DARPA SRS program under contract FA8750-04-2-0243.

References

Berry, G., and Gonthier, G. 1992. The *esterel* programming language: design, semantics and implementation. *Science of Computer Programming* 19(2):87–152.

de Kleer, J., and Williams, B. C. 1987. Diagnosing multiple faults. *Artificial Intelligence* 32(1):97–130.

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann.

Dressler, O., and Struss, P. 1996. The consistency-based approach to automated diagnosis of devices. *Principles of Knowledge Representation* 267–311.

Gottlob, G.; Leone, N.; and Scarcello, F. 2000. A comparison of structural csp decomposition methods. *Artificial Intelligence* 124(2):243–282.

Harel, D. 1987. Statecharts: a visual formalism for complex systems. *Science of Computer Programming* 8(3):231–274.

J. R. Burch, E. M. Clarke, K. L. M. D. L. D., and Hwang, J. 1992. Symbolic model checking: 10^{20} states and beyond. In *Information and Computation*, volume 98(2), 142–170.

Kask, K.; Dechter, R.; and Larrosa, J. 2003. Unifying cluster-tree decompositions for automated reasoning. Technical report, U. of California at Irvine.

Kurien, J., and Nayak, P. 2000. Back to the future for consistency-based trajectory tracking. In *Proc. AAAI-00*.

Mayer, W., and Stumptner, M. 2004. Approximate modeling for debugging of program loops. In *Proc. DX-04*.

Sachenbacher, M., and Williams, B. C. 2004. Diagnosis as semiring-based constraint optimization. In *Proc. ECAI-04*.

Schiex, T.; Fargier, H.; and Verfaillie, G. 1995. Valued constraint satisfaction problems:hard and easy problems. In *Proc. IJCAI-95*.

Williams, B. C., and Nayak, P. 1996. A model-based approach to reactive self-configuring systems. In *Proc. AAAI-96*, 971–978.

Williams, B. C.; Chung, S.; and Gupta, V. 2001. Mode estimation of model-based programs: monitoring systems with complex behavior. In *Proc. IJCAI-01*.

A Formal Framework for Goal Net Analysis

Carolyn Talcott and Grit Denker
SRI International
333 Ravenswood Ave
Menlo Park, California 94025
firstname.lastname@sri.com

Abstract

Planning systems are often not very transparent because details about plan generation are hidden inside software components. This makes it difficult to understand, and in consequence, to trust them. We propose a formal framework for planning systems that incorporates all important aspects ranging from plans, to domain models, to planning and execution. Our framework uses a formal language and analysis to specify and validate the correctness of planning system components and their interactions. The result is a formal checklist to which planning systems can be exposed to increase their level of dependability.

1. Introduction

Model-based planning systems (PSs) provide tools for developing autonomous remote agents. However, system designers and engineers are reluctant to use PSs due to their impression that such systems are unpredictable and not controllable. We are developing a formal framework for analysis of PSs including verification and validation methods based on the use of *formal checklists* for providing increased dependability of PSs. Formal checklists specify light weight formal analyses intended to detect a variety of potential problems such as errors in the underlying domain models, inconsistencies in complex plans or execution schedules, and failure to provide for unexpected conditions. Applying different levels of formal checklist give different levels of assurance of dependability.

Our formal framework is inspired by the MDS model-based goal-operated architecture for autonomous space systems (Dvorak *et al.* 2000). Key ideas of the MDS approach include:

- All knowledge of system state is maintained in a collection of state variables.
- The system is operated by specifying goals, that is, constraints on state variables over an interval of time.
- Complex goals are elaborated to goal nets consisting of a network of time points linked by (sub)goals and time constraints (timed constraint nets). The elaboration process is a form of planning. At the lowest level are executable goals.
- There is a goal achiever for each state variable that interacts with the environment (typically a device such as a rover), issuing commands to meet the constraints of executable goals, and reading sensors to maintain a model of the system state.

We use the rewriting logic language Maude (Clavel *et al.* 2003a; 2003b) to specify the formal framework as well as instantiations to be analyzed. Maude supports a variety of light-weight analysis techniques (see Section 5). architecture and interactions between architectural components and domain/device models, as well as goals, goal nets, goal elaboration and scheduling. Constraints on the behavior of each component are specified that support modular analysis. These give rise to checklist elements to be verified for specific instantiations. We also model how components such as the goal net and goal achievers interact in carrying out a goal-based operation. The formalization of timed constraint nets provides an abstract notion of time that can be instantiated to reason about timing properties at appropriate levels of detail. The result is a comprehensive formal specification of PS components and their interactions that can be exposed to a variety formal verification and validation tests to detect possible errors.

In this paper we focus on the formalization and analysis of goal nets. In an earlier paper (Denker & Talcott 2004) we described the formalization of goal achievers and an instantiation to a very simple rover device. In the future, we will extend the framework to encompass goal elaboration and further develop the checklist suite to cover additional properties and aspects of planning and execution. In Section 2 we introduce the kinds of goal net analyses that we intend to support with our formal framework. The semantic concepts and formal notation that form the basis for our specification and analysis are presented in Section 3. In Section 4 we sketch the formal model of goal nets. We briefly explain in Section 5 which analysis capabilities of the Maude toolkit make this language particularly suitable for the task at hand. In addition, we propose several validation and verification checklist elements for goal nets. We discuss related work and future extensions of our framework in Section 6 and conclude with a brief summary in Section

7. More details about the framework presented in this paper (including Maude specifications of some of the components) can be found at `http://www.csl.sri.com/users/denker/remoteAgents`.

2. Objectives

A goal net represents a plan for the successful execution of higher-level tasks. Depending on the abstraction level of the goal net, it can correspond to a fully instantiated plan with executable steps, it can relate higher-level goals to lower-lever, executable goals, it can represent several alternatives out of which one is selected only at runtime when system parameters are defined, or it can leave certain goals or tasks under-specified and requires re-planning at runtime or advice from a user.

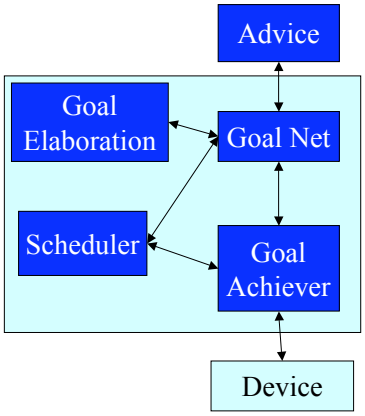


Figure 1: Components Formalized

The purpose of our formal framework is to capture the most important aspects of goal net representation, elaboration, and advice and to provide a list of checks for these three aspects of goals nets so as to increase the level of dependability that a goal net will ultimately be executable and achieve the overall goals. The framework will treat at least the components shown in Figure 1. Executing a goal net requires information exchange among several components including the goal net, goal achievers, device and scheduler. The goal net issues constraint requests to the goal achiever. The goal achiever issues commands to the device and takes sensor reading. The scheduler synchronizes the goal net and the goal achievers. The goal elaboration and the advice components both interact with the goal net, though they are used in different contexts. The goal elaboration process is an automated process whereas the advising component involves a human being.

We propose a goal net analysis taxonomy that defines a set of tests that result in increasingly dependable goal nets.

Static goal net analysis. We can perform static checks on goal nets that are fully refined into executable goals. Checks are, for example, well-formedness and consistency checks, or testing to what extent executable goals may have side effects on the system state that would result in goals interfering with one another. These checks are done offline, before the goal net is deployed into a system. This kind of static analysis can be performed on fully refined goal nets that only refer to executable goals, as well as on hierarchical goal nets that are fully refined into executable goals. Though, in general our framework can handle both, hierarchical goal nets and goal nets that are only comprised of executable goals, the current formalization does only handle goal nets with executable goals. In the future we will investigate what kinds of checks can be performed on hierachical goal nets. Moreover, we will also extent the framework to include alternatives in goal nets and propose static checks for those cases.

Dynamic goal elaboration analysis. The next step will be to incorporate the process of dynamic goal net elaboration. Assume a situation where the specific plan for achieving a goal depends on past values of state variables or the history of exchanged messages between system components. Elaboration of a not yet fully refined goal has to be postponed until runtime, when history information becomes available. Effectively, goal net planning and execution will become interleaved processes. We intend to capture this by extending our formal framework with a formalization of the goal elaboration process and its interaction with the other processes and components in the architecture.

Analysis of interactive goal net advice. Finally, we will also address the issue of interactive goal net modification. Users may change the goal net dynamically during its execution. One may add a new subnet that addresses runtime problems or increases the functionality of a goal net to handle exceptions. In addition, the user may decide on alternative or additional goals as a mission proceeds. Future extensions of our formal framework will have a presentation of the advice component and its interactions with other components.

Analyses of static goal nets is simpler and can be more precise as more is known about possible behaviors than for the case of dynamically generated or modifiable goal nets. Modular analysis is especially important to support safe runtime editing.

In summary, the formal models of all architectural components, their behavior and their interactions enables the use of formal analysis models to uncover errors and unexpected behavior. In this paper we present the first steps towards to this formal framework that models (possibly hierarchical) goal nets, goal achievers, schedulers and devices.

3. The Formal Framework

We build our formalization on the concepts of object and component. Objects are independent computational units (like actors) that interact via message passing. Components are collections of (sub) components and objects encapsulated by an interface. A component interface specifies which objects are visible from outside (receptionists) and which external objects are visible from inside (externals) as well as the messages that can be sent or received. We will treat single objects as components when convenient. Treating objects as components makes it easy to refine an object to a collection of objects suitably encapsulated. For example, the light colored box of Figure 1 is a component containing Goal Net, Goal Elaboration, Goal Achiever and Scheduler (sub) components. The goal net and goal achiever components contain multiple objects while we have chosen to model the scheduler component as a single object. Often interfaces can be organized as a set of sub-interfaces, each corresponding to interactions with another component. The interface of the component in Figure 1 has two parts: the Advice interface and the Device interface.

The semantics of components can be given at several levels of detail: the possible computations (sequences of state transitions and interactions with external objects); event-partial orders (message receives) including external interactions); interaction paths (just observing interactions with external objects). The hierarchical organization of components provides modularity at both the syntax and semantics levels (Talcott 1998). Thus, we can specify and analyze subsystems and their interactions at different levels of granularity. For example we can compose the semantics (at any level of detail) of the Goal Net, Goal Achiever, Goal Elaboration, and Scheduler to obtain the semantics of the whole component, and each of these sub components can be analyzed separately.

Figure 2 abstractly depicts the four main components and their interactions of the framework that we have formalized so far. The components are device (such as a rover), goal achiever, goal net, and a scheduler that coordinates the goal achiever and goal net—and their interactions. The structure and behavior of a goal net is described in some detail in the next section. We conclude this section with a short introduction to rewriting logic and Maude, and a brief summary of the goal-achiever and the scheduler structure and behavior. Details for the latter can be found in (Denker & Talcott 2004; 2003).

3.1 Rewriting Logic and Maude

Rewriting logic (Meseguer 1992) is a logical formalism that is based on two simple ideas: states of a system are represented as elements of an algebraic data type; and the behavior of a system is given by local transitions between states described by *rewrite rules*. A rewrite rule has the form $t \Rightarrow t'$ `if` c where t and t' are terms representing a local part of the system state and c is a boolean term. This rule says that when the system has a subcomponent matching t such that the instantiation of c holds, that subcomponent can evolve to t', possibly concurrently with changes described by rules matching other parts of the system state.

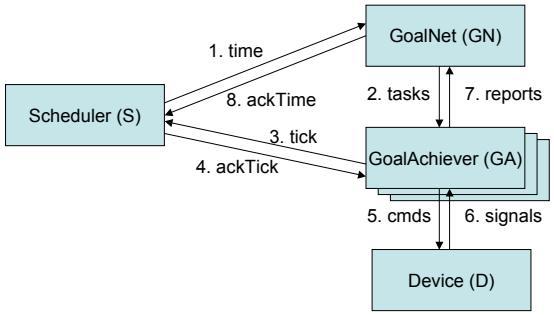


Figure 2: Architecture: showing the four main components and overall flow of data and control.

Maude (Clavel *et al.* 2003a; 2003b) is a language and specification environment based on rewriting logic. The Maude environment includes a very efficient rewrite engine with several built-in rewrite strategies for prototyping as well as tools for analysis (see Section 5). Maude sources, executables for several platforms, the manual, a primer, cases studies and papers are available from the Maude web site `http://maude.cs.uiuc.edu`.

Objects and messages are represented as terms in Maude. We use object syntax of the form

```
[ oid : C | a1: v1, ... an: vn ]
```

where `oid` is an object identifier, `C` is a class identifier `a1: v1` is an attribute with name `a1` and value `v1`. Object behavior is specified by giving rules for receiving messages. A typical object rule has the form

```
[ oid : C | atts ] msg
 =>  [ oid : C | atts' ] newmsgs
```

where `msg` is a message addressed to `oid`, `newmsgs` is a (possibly empty) multiset of messages sent, and `atts'` is the object's updated attribute set. A configuration is a multiset of objects and messages. If a configuration contains an object and message matching the left-hand side of the above rule, then the whole configuration will rewrite by replacing the object and message by the corresponding instantiation of the right-hand side of the rule. Components are represented as terms with two parts, an interface and a configuration.

3.2 Goal Achievers

A system has a set of state variables that encompass all knowledge of system (and environment) state: quantities

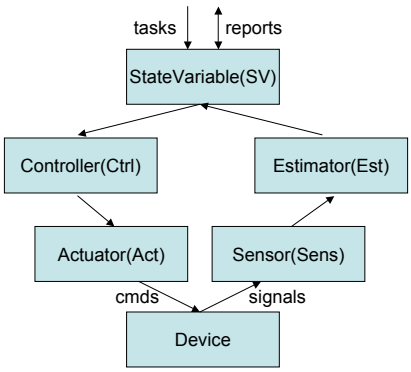


Figure 3: Goal Achiever and Device

that can be computed from sensor readings and domain models. There is a goal achiever component for each state variable. Each goal achiever encompasses five components as shown in Figure 3, namely state variable, controller, actuator, sensor, and estimator. In our framework each of these components is formalized as a single object. The state variable is the interface of a goal achiever component to the goal net and to other goal achiever components. The actuator and sensor form the device interface. Start constraint requests are sent to a state variable by an executable goal. If not already busy, the state variable informs the controller of the new constraint and the controller generates a course of action (a sequence of device commands) expected to lead to satisfaction of the constraint. To achieve a constraint a goal achiever operates in a cycle controlled by the state variable. When triggered, by a tick event from the scheduler, the state variable enters an MDS cycle or so-called goal achiever cycle, sending the current value to the controller. The controller checks to see if the constraint is satisfied. If so, it reports success to the state variable, which in turn reports success to its goal. Otherwise the controller issues the next command in its course of action to the actuator, which in turn issues the appropriate instruction to the device. The device reports state changes to the sensor that, in turn, forwards the latest measurements to the estimator. The estimator updates the value of state variable. The state variable reports new values to any objects (other state variables, goals, ...) registered for notification.

3.3 Scheduler

The scheduler controls system operation using clock cycles. Each clock cycle has two phases: a goal net phase, and a goal achiever phase. For the goal net phase the goal net is sent a time message. In response, the goal net updates its internal state according to the new time. This may result in new constraint requests sent to goal achievers. For the goal achiever phase, a tick event is sent to each goal achiever (one for each state variable). In response each goal achiever with an active constraint will execute one goal achiever cycle. This may result in commands sent to the device, sensor reading, notification of new state variable values, and reports of success or failure sent by state variables to requesting goals. When the goal-achiever phase completes the clock time is incremented and the scheduler starts a new cycle.

We specify interaction invariants that must hold for the system that are useful in carrying out component analyses and lifting these to overall system properties. One example is that if an executable goal has start time t, then the state variable will have received the constraint request before it receives the tick for the clock cycle a time t. Another example is that if a state variable deems a constraint satisfied or failed during its phase, then the requesting goal will have received a report to this effect before the next clock cycle starts. Also, all activity of the goal-net phase must complete before the goal-achiever phase is started, and all activity of the goal-achiever phase must complete before the next clock cycle is initiated.

4. Goal Nets

Formally, a goal net is a graph whose nodes are time points and whose edges are goals and time constraints. A time point has a time value, that can be unspecified, or a time value in some time domain (for example 3pm Earth time on July 30, 2005). As the goal net is executed time points acquire specific values by 'firing'. Each goal has two time points associated with it, the beginning time point (edge source) and the ending time point (edge target). It also specifies a state variable constraint that is to hold in the time interval between its starting and ending time points. Each time constraint also has a beginning and ending time point. A time constraint contains an interval $[min, max]$ that specifies the minimum and maximum allowed difference between the values of its ending and beginning time points. For example, a constraint $[20, 30]$ for two time points $TP0$ and $TP1$ means that the time point $TP1$ cannot fire earlier than 20 time units after time point $TP0$ fired, and it must fire within 30 time units after $TP0$ fired. We classify goals as achieving (for example driving to a location, or heating to a specified temperature) or maintaining (parking at a location, keeping the temperature during a certain interval, or monitoring the battery level). A goal net must be acyclic, and thus determines a partial order on time points (and their values). Figure 4 shows an example goal net.

In this figure, squares denote goals, ovals denote time points and hexagons denote time constraints. On the left side of Figure 4 we have a goal $G1$ with starting time point $TP1$ and ending time point $TP2$. This means, that the constraint of goal $G1$ should hold in the time interval between $TP1$ and $TP2$. Suppose $G1$ is 'park at location L' for 10-15 time units. Suppose further that the flight rules say that the battery level must stay above 30%. Then $G1$ might elaborate

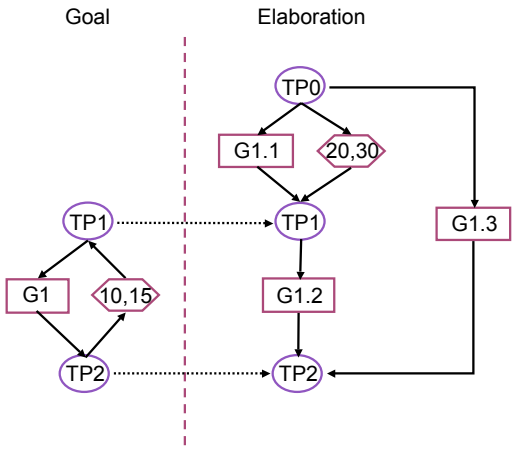


Figure 4: Goal Nets: Time points and Goals

to the goal net on the right side of Figure 4. The elaboration has three subgoals $G1.1$, $G1.2$, and $G1.3$. $G1.1$ is an achieving goal to drive to location L, and this must be done before parking. Thus a new time point $TP0$ is introduced as its starting time point with $TP1$ as its ending time point. There is a time constraint $[20, 30]$ on time points $TP0$ and $TP1$. The constraint says that $TP1$ cannot fire earlier then 20 time units after $TP0$ has fired and it has to fire at the latest 30 time units after $TP0$ has fired. If driving does not reach location L within 30 time units $G1.1$ is not achieved. $G1.2$ is a maintaining goal, to park at the current location, and $G1.3$ is a maintaining goal that monitors the battery level, thus ensuring that the battery level flight rule is obeyed. The time constraint between $TP1$ and $TP2$ expresses the desired duration of parking.

In general a goal net will have both executable and non-executable goals. Non-executable goals are elaborated to subnets containing subgoals with additional time points and possibly additional time constraints. Thus, goal nets are structurally organized using two dimensions. One dimension is the partial order on time points. The other dimension represents the hierarchical structure of goals in the goal net due to goal elaboration. For example in Figure 4 $G1$ is a non-executable goal and has the executable goals $G1.1$, $G1.2$, and $G1.3$ as subgoals. Time points, goals, and time constraints, are formally represented as objects with the underlying graph connectivity information recorded in object attributes. In addition to the time point, goal, and time constraint objects, a goal net component contains a goal net object that serves as the interface to the scheduler and and advisor. In this paper we only consider interactions with the scheduler, that is, the response to time messages. The interactions of a goal net component during the goal net phase is illustrated in figure 5.

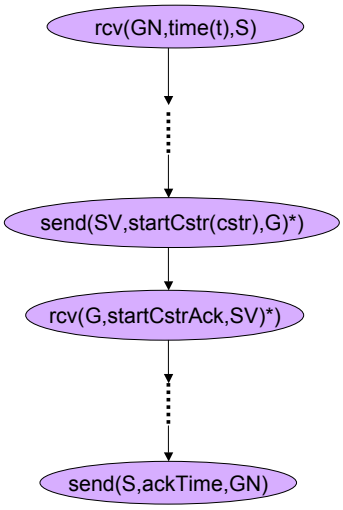


Figure 5: Goal Net Interactions

The goal net object receives a time event from the scheduler. The goal net object forwards time messages to each time point that has not fired, giving it an opportunity to fire if it is ready. Internally a time point will consult its constraints and possibly trigger firing of goals. All that is observed at the component level is constraint requests `startCstr(...)` sent to state variables from goals, fired as a result of the propagating time message, and the corresponding acknowledgments. Internally, after all goals received acknowledgements from the state variables, the goals will in turn send acknowledgements of the time event to the timepoints, which will acknowledge the time event to the goal net. All these message interchanges are not visible at the component level. Only the resulting acknowledgement `ackTime` of the goal net to the scheduler is visible.

In the following subsections we outline the formalization in Maude of the behavior of goal net objects, time points, time constraints, and goals. These behaviors have been formalized in Maude. Here we use an informal notation describing the interactions from each objects point of view inspired by the specification diagram formalism (Smith & Talcott 2002). The notation essentially describes regular expressions of send/receive events and local state updates.

4.1 Goal net objects

A goal net object keeps track of all time points, stored in two attributes: `openTPs`, time points that have not fired; and `firedTPs`, time points that have fired. A goal net object is formalized as a Maude term of the form

```
[ GN: GoalNet | openTPs: tps,
                firedTPs: tps', atts]
```

where `atts` represents additional attributes needed for keeping track of processing state.

When a goal net object receives a time event `(GN,time(t),S)` from scheduler `S`, it forwards this event to all open time points, processes all acknowledgments, and then reports completion to `S`. The following specifies this interaction from the goal net object's point of view.

```
rcv(GN,time(t),S):
  for tp in tps do
    send(tp,time(t),GN)
  endFor;
  for tp in tps do
    rcv(GN,ack,tp)
    if ack == Fired
    then move tp from openTPs to firedTPs
  endFor;
  send(S,ackTime(t),GN) .
```

In practice, at the cost of some additional bookkeeping, the goal net object needs to only forward time events to time points that might be affected.

4.2 Time Constraints

Time constraints express requirements on the minimum and maximum time interval between firing of two time points. A time constraint knows its start and end time points, the value of the start time point and the upper and lower bounds on the time interval. The value of a time point is unspecified until it fires. To handle this situation, we use a sort `Time?`, that extends the sort `Time` with an undefined time value, `unkTime`. A time constraint also knows its parent goal identifier, in order to be able to report constraint failures for the parent goal to handle. Time constraint objects are formalized in Maude as terms of the form

```
[ C : Constraint |
  startTp: tp0, endTp: tp1,
  start-time: t?, parent: p,
  imin: i, imax: j ]
```

A time constraint can receive a fired event from its starting time point, time events and resolve events from its ending time point. A fired event sets the starting time. A resolve event `resolve(t,reason)` signals a time constraint conflict and is forwarded to the time constraint's parent. A time event `time(t)` is interpreted as a request for the time constraint status `cstatus(?t,t,i,j)` where `cstatus` is defined, using the notation above, by

```
cstatus(t?,t,i,j) =
  if t? = unkTime
  then unkStatus
  else status  endIf;
  where status = early if t < t? + i
        status = ok if t? + i <= t < t? + j
        status = fire if t = t? + j
        status = late if t > t? + j
```

The following summarizes the behavior of a time constraint object.

```
rcv(c, fired(t), tp0):
  start-time := t;
  send(tp0,firedAck,c) .

rcv(c, resolve(t,reason), tp1):
  send(p,resolve(t,reason),c);
  rcv(c,resolveAck,p);
  send(tp1,resolveAck,c) .

rcv(c, time(t), tp1):
  send(tp1, cstatus(t?,i,j), c) .
```

The rule describing the behavior of the time constraint upon receipt of a fired event from a timepoint is formalized in Maude as follows:

```
[ c : Constraint |
   startTp: tp0, endTp: tp1, start-time: t?,
   parent: p,  imin: i, imax: j ]
msg(c, fired(t), tp0)
=>
[ c : Constraint |
   startTp: tp0, endTp: tp1, start-time:t,
   parent: p, imin: i, imax: j ]
msg(tp0,firedAck,c) .
```

4.3 Time points

Time points are partially ordered via time constraints and goals. Each time point knows the goals for which it is the start point as well as those goals for which it is the endpoint. We separate the "end goals" into those goals which are required for the endpoint to fire and those that are not required.

For example, in Figure 4 one can imagine that goal $G1.1$ is required for $TP1$ whereas goal $G1.2$ is not required for $TP2$. This means, that $TP2$ can fire even if goal $G1.2$ has not reported completion. Time points also know the constraints for which they are start- and end-points. Time points that have fired have a time value assigned, open time points will have the value `unkTime`. Time point objects are formalized in Maude as terms of the form

```
[ tp : Timepoint |
  value: t?, startC: cstrs, endC: cstrs',
  startG: goals, endGReq: goals',
  endGOther: goals'']
```

where `t?` is a (possibly unknown) time value, `cstrs`, `cstrs'` are sets of identifiers of constraint objects, and `goals`, `goals'`, `goals''` are sets of identifiers of goal objects.

Time points receive `time` events from the goal net object and `done` events from goal objects for which they are the end point. When a time point receives a time event from the goal net it first forwards the events to all constraints for which it is the ending time point, and collects a summary of the status reports using a combination function `&` that is associative, commutative, and idempotent with identity `ok`. The summary result is one of `ok`, `fire`, `late`, `conflict` where

```
late & early = late & fire = conflict
late & unkStatus =  late
```

```
  fire & early = fire & unkStatus = conflict
  early & unkStatus = early
```

The time point uses the collected status reports to decide whether to timeout (a constraint upper bound has been exceeded), request time constraint conflict resolution, fire (firing preconditions hold), or pass (some firing precondition fails or firing is not forced).

The behavior of time point response to time events is summarized in the following

```
rcv(tp, time(t), GN):
  for c in endC do
    send(c, time(t), tp);
  endFor;
  result := ok;
  for c in endC do
    rcv(tp, cstatus, c);
    result := result & cstatus;
  endFor;
  if result = late
  then for x in endGReq + endGOther
         send(x, timeout(t), tp);
         rcv(tp, timeoutAck, x);
       endFor;
       for x in endC do
         send(x, resolve(t,late), tp);
         rcv(tp, resolvetAck, x);
       endFor;
       for x in startG + startC do
         send(x, fired(t), tp);
         rcv(tp, firedAck, x);
       endFor;
       send(GN, timeAck(false), tp);
  else if result = conflict
       then for x in endC do
              send(x, resolve(t), tp);
              rcv(tp, timeouAck, x);
            endFor;
            send(GN, timeAck(false), tp);
       else if result = fire
            then do fire(t)
            else choose
            (send(GN, timeAck(false), tp)
             or fire(t));
  endIf  endIf  endIf

where
  fire(t):
    for x in startG + endGOther + startC
      send(x, fired(t), tp);
      rcv(tp, firedAck, x);
      send(GN, timeAck(true), tp)
    endFor .
```

Note that waiting too long (passing too often), or firing too soon can cause later constraints to fail. Checklist properties and constraint net analysis can be employed to avoid this.

When a time point receives a `done` message from a goal, the time point deletes this goal from the set of required goals and acknowledges the receipt of the `done` message.

```
rcv(tp, done, g):
   remove g from endGreq;
   send(g, doneAck,tp) .
```

4.4. Goals

Goals are either executable or non-executable. Non-executable goals maintain a set of children that constitute the result of elaboration of the goal. For example, in Figure 4, the goal $G1$ has children $G1.1, G1.2,$ and $G1.3$. Once elaborated, the main role of a non-executable goal is to resolve conflicts and recover from constraint failures. Here we focus on executable goals.

Executable goals simply manage interaction with the goal achiever component. Each executable goal represents a constraint on a state variable over a time interval represented by a pair of time points. Thus an executable goal knows its constraint, the state variable being constrained, its starting and ending time points and its parent goal. It also knows whether or not its completion is required for the ending time point to fire. This is represented by a boolean flag.

Executable goals are formalized as Maude terms of the form

```
[ g : ExecGoal |
  cstr: C, statevar: sv, parent: p,
  startTp: tp0, endTp: tp1, req: b ]
```

When a goal receives a fire message from its starting time point it starts the goal achiever cycle by sending a start constraint message to the corresponding state variable. It is possible that the state variable is busy with another constraint. In that case, the state variable will acknowledge the start constraint message with a ”busy failure”. Thus, the goal cannot be achieved at the moment. This is reported to the parent goal and could cause a different mechanism, such as goal elaboration, to adjust the goal net. In addition, if the goal is required for its ending time point, this time point is notified that the goal has completed. In either case the goal acknowledges the fired message. The + in the following pseudo code denotes an internal non-determinism, meaning that not the goal is to decide which one of the two branches it will execute.

```
rcv(g, fired(t),tp0):
  send(sv,startConstr(cstr), g);
   ( rcv(g,startCstrAck,sv)
     +
     ((rcv(g,cstrFail(reason),sv);
       (send(p,cstrFail(reason),g);
        rcv(p,reportAck,tp1);)
        ||
       (send(tp1,done,g);
        rcv(g,doneAck,tp1);)))
     ) ;
    send(tp0,firedAck,g);
```

When a goal receives a report (of successful or unsuccessful) constraint satisfaction from is state variable, it notifies its parent and, if required, its end time point.

```
rcv(g, report, sv):
  send(p,report,g);
  rcv(p,reportAck,tp1);
  if b
  then  send(tp1,done,g);
        rcv(g,doneAck,tp1);
  endIf;
  send(sv,reportAck,g);
```

A non-required goal may receive a fired message from its endpoint. Such goals are typically maintaining goals, and the end time point firing means that the goals job is done. In this case the goal notifies its state variable and its parent.

```
rcv(g, fired(t),tp1):
  send(sv,stopConstr(cstr), g);
  rcv(g,report,sv);
  send(p,report,g);
  rcv(p,reportAck,tp1);
  send(tp1,firedAck,g) .
```

A goal may also receive timeout events from its end time point or an abort event from its parent. We omit the details.

5. Analysis Checklist for Goal Nets

Now we describe in more detail some of the goal net analyses that we expect to include as checklist items. Besides the modeling and execution capabilities, Maude also provides efficient built-in search and model checking capabilities. Thus, many of the analyses can be carried out using tools in the Maude environment. In addition, Maude is reflective (Clavel 1998; Clavel & Meseguer 1996) providing a meta-level module that reflects both the syntax and semantics of Maude. Using reflection, special purpose execution and search strategies, module transformations, special purpose analyses, and user interfaces can be programmed. Also using reflection, theory mappings can be defined that map Maude specifications to a form that can be analyzed by tools developed for other logics.

As discussed in (Denker & Talcott 2004), a simple test is just to execute a goal net by composing it with goal achievers and a device (modeled at some appropriate level of abstraction). Then one can use the Maude search capability to look for expected and unexpected outcomes.

In the following we discuss three general classes of analysis referred to briefly as structural, behavioral, and domain.

Structural analyses

Structural analyses are intended to insure that basic architectural constraints are met within and across components.

Time Constraint Consistency. This analysis checks that for all constraints the value of the `imax` attribute (the upper bound) is greater than or equal to the value of the `imin` attribute (the lower bound).

Link Consistency. Connectivity of a goal net is represented implicitly and redundantly in the `startTp endTp` attributes of its goals and time constraints and the `startG`, `endGReq`, and `endGOther` attributes of its time points. It is important that these attributes present a consistent view. For example, if the `endC` attribute of a time point `tp` has a constraint `C` as one of its elements, then the `endTp` of `C` must have the value `tp`. Similar consistency conditions must hold for time constraint start points, and for goal start and end points and the corresponding time point attributes. The link consistency analysis checks that these conditions are satisfied. In addition, it must check that if a goal is a member of the `endGReq` attribute of a time point (the goal must report done before the time point can fire), then the required attribute of the goal is set to `true`.

Acyclicity. The underlying graph of a goal net, and the subgoal relation must both be acyclic. The acyclicity analysis checks that this is the case.

Behavior analyses

Behavior analysis is intended to insure that during execution a system will not reach a "bad" state.

Goal Net Consistency. One of the checks applied to a formally specified goal net is to verify that the goal net behavior does not result in inconsistent reply messages from time constraint objects. For example, it should not be possible that one time constraint object replies to a `time(t)` message from a time point with `early` and another one with `late`. Similarly, the combination `early` and `fire` is inconsistent. Using the formal specification of goal nets, we can expose a given goal net to check for those inconsistencies. For this purpose, we use Maude's model checking capabilities. We try to contradict the statement that there is no reachable state in which two inconsistent replies from time constraints were received. If the model checker finds a counter example it will provide details about the inconsistency in the goal net.

Timely constraint checking It is important that time constraints are checked as soon as possible, i.e. as soon as the start time point is known, satisfaction by putative end time points can be checked. The timely constraint checking analysis checks that each time constraint has the necessary information available to accurately determine status when it receives a time(t) request from its end time point.

Mission rules In most situations there are global constraints on the system state that must hold independent of particular goals. For example a remote rover activity should not drain the battery dangerously low. Or the some sensor temperature reading should remain within certain bounds (to avoid equipment damage). These global constraints are called flight or mission rules. It is the responsibility of the mission designer to spell out all such rules. Then model checking, possibly in combination with domain specific analyses, can be used to check that a goal net does not violate mission rules.

Avoiding time constraint failure. In general, a time point has some flexibility as to when it fires. That is, there may be several clock cycles when it is OK for it to fire, but not required. Without additional information, firing as soon as all constraints report ok could lead to later time-constraint violations as could postponing until some constraint says fire. There are algorithms (Dechter, Meiri, & Pearl 1991) to label nodes of a time constraint net with intervals representing constraints on the value of the time point. For example the minimal net algorithm assigns feasible intervals to each time point so that if for a given node, any value in the interval is picked, there is an assignment for the rest of the nodes with in their specified interval, that meet the initial constraints. This sort of instrumentation can help to avoid the above problem or to detect unrealizable constraints before execution.

Avoiding timeouts. If estimates of goal achievement time (how many cycles a goal achiever will take to succeed with given constraints) are given this analysis uses simple scheduling ideas to determine if it is possible to meet the time constraints. In the absence of estimates it can try to provide plausible bounds.

Adequacy of the formal device model.

The device model of the goal achiever component is formally modeled. This allows to analyze goal nets with respect to the formal device model, that is, under the assumption that the device model is correct, we can test the level of dependency of the goal net. Tests like this can be performed in an laboratory environment that does not require expensive equipment. If we also have an interface implemented between the goal achiever component and the real-world device, then we can rerun the same tests that we performed on the formal device model. Comparing the results from both test suites allows us to compare the formal device model with the actual real-world behavior of the device. We envision that in the future a formally defined module, the so-called *Device Model Corrector* will process the results of this comparison and initiate changes in the formal device model.

6. Related Work and Future Directions

One of the differences of our approach to more classical planning systems is that we consider all different components in the planning and execution process. Instead of focusing on a particular planning algorithm, our focus is on the formalization of the planning system components and their interactions. In particular, we aim to take advantage of model checking approaches to validate the correctness of component interfaces and to uncover possible inconsistencies due to the inherent concurrency of components such as goal nets, goal achievers, goal elaboraton, external advice, scheduler, etc.

Our framework uses a very abstract notion of time that can be further refined into more concrete notions of time as required by the application domain. In the future we will investigate to what extent discretized and continuous actions and plans as supported by PDDL2.1 (Fox & Long 2003) are expressible within our framework. Some concepts in PDDL2.1, such as numeric expressions, conditions and effects are directly expressible within our framework, since Maude supports these features. Nevertheless, the current PDDL2.1 language is more flexible with respect to temporally annotated conditions and effects and we will investigate whether extensions to our framework will be straightforward and what effect they have on the component behavior specifications and the component interface specifications.

Wilkins and desJardins compare knowledge-rich planning approaches that use domain knowledge with minimal-knowledge planning approaches in (Wilkins & desJardins 2001). They argue that the use of domain knowledge increases expressiveness, allows for plan modification during execution and is more scaleable. Our formal framework uses formal domain models in various places (e.g., goal structures and human advise among others), and thus, falls into the knowledge-rich planning category. Other domain-specific information such as search-control techniques can be implemented in the goal elaboration module. One could for example imagine that the goal elaboration process exploits the formal device models. This could be done by using model checking approaches to determine the reachability of a certain device state. A model checker would also deliver the sequence of states that lead to the desired state. Doing this for several devices and goals, the resulting information can be fed into the goal elaboration process to optimize an overall strategy that achieves multiple goals.

One of the areas that we have not yet investigated is the extensibility of our framework to probabilistic planning techniques (cf. (Pro 2004)) and the integration of planning and learning techniques (cf. (Veloso *et al.* 1995)).

Though the framework already provides for hierachical goal nets, only the behavior of goal nets that are comprised of executable goals has been formalized in Maude. Not only do we have to define the behavior of hierarchical goal nets, but we also have to investigate the consquences for our formal checklists. A hierarchical goal may be refined into several executable or non-executable goals. These goals may have constraints for different state variables. Thus, a hierarchical goal might need to refer to a list of state variables for which it attempts to achieve constraints. Hierarchical goals have start and end time point associated. These time points may have time constraints with other time points that are start or end time points of goals of a different hierarchy level. Overall, there will be challenges to overcome in defining the behavior of hierarchical goal nets as well as interaction constraints to be defined that assure correct goal net execution.

7. Concluding Remarks

In this paper we presented a formal framework for planning systems. In contrast to other work in the area of planning, we use a comprehensive approach to specifying and validating planning system components and their interaction. We propose to use a formal language and checklists of formal analyses. This paper constitutes a first step in the direction of a general formal framework for planning systems. We focussed on goal nets and their dependability. In the future we will extend our framework and investigate the incorporation of other existing planning concepts, and we will also provide an extended list of checklists. Our vision is to identify verification and validation mechanisms that can be applied to planning systems with the goal of increasing their dependability and the trust of users in their adequacy.

References

Clavel, M., and Meseguer, J. 1996. Reflection and strategies in rewriting logic. In *Rewriting Logic Workshop'96*, number 4 in Electronic Notes in Theoretical Computer Science. Elsevier. `http://www.elsevier.nl/locate/entcs/volume4.html`.

Clavel, M.; Durán, F.; Eker, S.; Lincoln, P.; Marti-Oliet, N.;

Meseguer, J.; and Talcott, C. 2003a. Maude 2.0 Manual. http://maude.cs.uiuc.edu.

Clavel, M.; Durán, F.; Eker, S.; Lincoln, P.; Martí-Oliet, N.; Meseguer, J.; and Talcott, C. L. 2003b. The Maude 2.0 system. In Nieuwenhuis, R., ed., *Rewriting Techniques and Applications (RTA 2003)*, volume 2706, 76–87. Springer-Verlag.

Clavel, M. 1998. *Reflection in General Logics, Rewriting Logic, and Maude*. Ph.D. Dissertation, University of Navarre.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence* 49:61–95.

Denker, G., and Talcott, C. 2003. Maude specification of the MDS architecture and examples. Technical Report SRI-CSL-03-03, Computer Science Laboratory, SRI International. www.csl.sri.com/users/denker/publ/DenTal03.pdf.

Denker, G., and Talcott, C. 2004. Formal checklists for remote agent dependability. In *5th Intern. Workshop on Rewriting Logic and Its Applications, Barcelona, Spain, March 27-28, 2004*.

Dvorak, D.; Rasmussen, R.; Reeves, G.; and Sacks, A. 2000. Software Architecture Themes In JPL's Mission Data System. In *IEEE Aerospace Conference, USA*.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20:61–124.

Meseguer, J. 1992. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science* 96(1):73–155.

2004. Proc. for the probabilistic planning track of ipc-04. http://www.cs.rutgers.edu/~mlittman/topics/ipc04-pt/proceedings.

Smith, S. F., and Talcott, C. L. 2002. Specification diagrams for actor systems. *Higer-Order and Symbolic Computation* 15(4):301–348.

Talcott, C. L. 1998. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation* 11(3):281–343.

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

Wilkins, D., and desJardins, M. 2001. A call for knowledge-based planning. *American Association for Artificial Intelligence* 99–115.