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

Copyright © 2005, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.
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-level, 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.

![Diagram](attachment:component-diagram.png)

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 goal 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 hierarchical goal nets. Moreover, we will also extend 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 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.

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

$$[ \text{oid} : \text{C} | a_1 : v_1, \ldots, a_n : v_n ]$$

where $\text{oid}$ is an object identifier, $\text{C}$ is a class identifier $a_1 : v_1$ is an attribute with name $a_1$ and value $v_1$. Object behavior is specified by giving rules for receiving messages. A typical object rule has the form

$$[ \text{oid} : \text{C} | \text{atts} ] \text{msg} \Rightarrow [ \text{oid} : \text{C} | \text{atts}' ] \text{newmsgs}$$

where $\text{msg}$ is a message addressed to $\text{oid}$, $\text{newmsgs}$ is a (possibly empty) multiset of messages sent, and $\text{atts}'$ 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...
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 $T0$ and $T1$ means that the time point $T1$ cannot fire earlier than 20 time units after time point $T0$ fired, and it must fire within 30 time units after $T0$ 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 $T0$ and ending time point $T2$. This means, that the constraint of goal $G1$ should hold in the time interval between $T0$ and $T2$. 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 3: Goal Achiever and Device](image-url)
Figure 4: Goal Nets: Time points and Goals

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 \texttt{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 \texttt{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

\[ [ \text{GN: GoalNet} | \text{openTPs: tps, firedTPs: tps'}, \text{atts}] \]

where atts represents additional attributes needed for keeping track of processing state.

When a goal net object receives a time event \((\text{GN, time}(t), S)\) 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.

\[
\text{rcv(GN, time(t), S):}
\]

\[
\text{for tp in tps do}
\]

\[
\text{send(tp, time(t), GN)}
\]

\[
\text{endFor;}
\]

\[
\text{for tp in tps do}
\]

\[
\text{rcv(GN, ack, tp)}
\]

\[
\text{if ack == Fired}
\]

\[
\text{then move tp from openTPs to firedTPs}
\]

\[
\text{endFor;}
\]

\[
\text{send(S, ackTime(t), GN)}
\]

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 \(\text{Time}\), that extends the sort \(\text{Time}\) with an undefined time value, \(\text{unkTime}\). 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

\[ [ \text{C: Constraint} | \text{startTp: tp0, endTp: tp1, start-time: t?, parent: p, imin: i, imax: j } ] \]

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 \(\text{resolve}(t, reason)\) signals a time constraint conflict and is forwarded to the time constraint’s parent. A time event \(\text{time}(t)\) is interpreted as a request for the time constraint status \(\text{cstatus}(t?,t,i,j)\) where \(\text{cstatus}\) is defined, using the notation above, by

\[
\text{cstatus}(t?,t,i,j) =
\]

\[
\text{if t? = unkTime}
\]

\[
\text{then unkStatus}
\]

\[
\text{else status endIf;}
\]

where \(\text{status = early if t < t? + i}
\]

\[
\text{status = ok if t? + i <= t < t? + j}
\]

\[
\text{status = fire if t = t? + j}
\]

\[
\text{status = late if t > t? + j}
\]

The following summarizes the behavior of a time constraint object.

\[
\text{rcv(c, fired(t), tp0): start-time := t; send(tp0, firedAck, c) .}
\]

\[
\text{rcv(c, resolve(t, reason), tp1): send(p, resolve(t, reason), c);}
\]

\[
\text{rcv(c, resolveAck, p); send(tp1, resolveAck, c) .}
\]

\[
\text{rcv(c, time(t), tp1): send(tp1, cstatus(t?,i,j), c) .}
\]

The rule describing the behavior of the time constraint upon receipt of a fired event from a timepoint is formalized in Maude as follows:

\[
[ \text{c: Constraint} | \text{startTp: tp0, endTp: tp1, start-time: t?, parent: p, imin: i, imax: j } ]
\]

\[
\text{msg(c, fired(t), tp0) => [ c: Constraint |}
\]

\[
\text{startTp: tp0, endTp: tp1, start-time: t, parent: p, imin: i, imax: j ]}
\]

\[
\text{msg(tp0, firedAck, c) .}
\]

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 \(\text{unkTime}\). Time point objects are formalized in Maude as terms of the form

\[ [ \text{tp: Timepoint} | \text{value: t?, startC: cstrs, endC: cstrs’, startG: goals, endGReq: goals’, endGOther: goals’’} ] \]

where \(t?\) is a (possibly unknown) time value, \(\text{cstrs, cstrs’}\) are sets of identifiers of constraint objects, and \(\text{goals, goals’, goals’’}\) 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 \text{ok}.

The summary result is one of \text{ok}, \text{fire}, \text{late}, \text{conflict}

\[
\text{late & early = late & fire = conflict}
\]

\[
\text{late & unkStatus = late}
\]
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

\[
\text{rcv}(tp, \text{time}(t), \text{GN}): \\
\text{for } c \in \text{endC} \text{ do} \\
\quad \text{send}(c, \text{time}(t), tp); \\
\text{endFor}; \\
\text{result} := \text{ok}; \\
\text{for } c \in \text{endC} \text{ do} \\
\quad \text{rcv}(tp, \text{cstatus}, c); \\
\text{result} := \text{result} \& \text{cstatus}; \\
\text{endFor}; \\
\text{if result} = \text{late} \\
\text{then for } x \in \text{endGReq + endGOther} \\
\quad \text{send}(x, \text{timeout}(t), tp); \\
\quad \text{rcv}(tp, \text{timeoutAck}, x); \\
\text{endFor}; \\
\text{for } x \in \text{endC} \text{ do} \\
\quad \text{send}(x, \text{resolve}(t, \text{late}), tp); \\
\quad \text{rcv}(tp, \text{resolveAck}, x); \\
\text{endFor}; \\
\text{for } x \in \text{startG} + \text{startC} \text{ do} \\
\quad \text{send}(x, \text{fired}(t), tp); \\
\quad \text{rcv}(tp, \text{firedAck}, x); \\
\text{endFor}; \\
\quad \text{send}(\text{GN}, \text{timeAck}(\text{false}), tp); \\
\text{else if result} = \text{conflict} \\
\text{then for } x \in \text{endC} \text{ do} \\
\quad \text{send}(x, \text{resolve}(t), tp); \\
\quad \text{rcv}(tp, \text{timeoutAck}, x); \\
\text{endFor}; \\
\quad \text{send}(\text{GN}, \text{timeAck}(\text{false}), tp); \\
\text{else if result} = \text{fire} \\
\text{then do fire(t)} \\
\text{else choose} \\
\quad (\text{send}(\text{GN}, \text{timeAck}(\text{false}), tp) \\
\quad \text{or fire(t)}); \\
\text{endIf} \text{ endIf}
\]

where

\[
\text{fire(t):} \\
\text{for } x \in \text{startG} + \text{endGOther} + \text{startC} \\
\text{send}(x, \text{fired}(t), tp); \\
\text{rcv}(tp, \text{firedAck}, x); \\
\text{send}(\text{GN}, \text{timeAck}(\text{true}), tp) \\
\text{endFor}.
\]

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.

\[
\text{rcv}(tp, \text{done}, g): \\
\text{remove } g \text{ from endGreq; } \\
\text{send}(g, \text{doneAck}, tp).
\]

### 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

\[
[ \text{g} : \text{ExecGoal} | \\
\text{cstr: C, statevar: sv, parent: p,} \\
\text{startTp: tp0, endTp: tp1, req: b} ]
\]

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.

\[
\text{rcv}(g, \text{fired}(t), tp0): \\
\quad \text{send}(sv, \text{startCstr} \text{cstr}, g); \\
\quad (\text{rcv}(g, \text{startCstrAck}), g) \\
\quad + \\
\quad ((\text{rcv}(g, \text{cstrFail}(reason), sv); \\
\quad \text{send}(p, \text{cstrFail}(reason), g); \\
\quad \text{rcv}(p, \text{reportAck}, tp1));) \\
\quad | | \\
\quad (\text{send}(tp1, \text{done}, g); \\
\quad \text{rcv}(g, \text{doneAck}, tp1));) \\
\text{)}; \\
\text{send}(tp0, \text{firedAck}, g);
\]

When a goal receives a report (of successful or unsuccessful) constraint satisfaction from its state variable, it notifies its parent and, if required, its end time point.

\[
\text{rcv}(g, \text{report}, sv): \\
\quad \text{send}(p, \text{report}, g); \\
\quad \text{rcv}(p, \text{reportAck}, tp1); \\
\quad \text{if b then send}(tp1, \text{done}, g); \\
\quad \text{rcv}(g, \text{doneAck}, tp1); \\
\quad \text{endIf; } \\
\quad \text{send}(sv, \text{reportAck}, g);
\]
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.

\[
\text{rcv}(g, \text{fired}(t), t) : \\
\quad \text{send}(sv, \text{stopConstr}(\text{cstr}), g) ; \\
\text{rcv}(g, \text{report}, sv) ; \\
\text{rcv}(p, \text{report}, g) ; \\
\text{rcv}(p, \text{reportAck}, t) ; \\
\text{send}(t, \text{firedAck}, g) .
\]

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 $\text{imax}$ attribute (the upper bound) is greater than or equal to the value of the $\text{imin}$ attribute (the lower bound).

Link Consistency. Connectivity of a goal net is represented implicitly and redundantly in the $\text{startTp}$ and $\text{endTp}$ attributes of its goals and time constraints and the $\text{startG}$, $\text{endGReq}$, and $\text{endGOther}$ attributes of its time points. It is important that these attributes present a consistent view. For example, if the $\text{endC}$ attribute of a time point $\text{tp}$ has a constraint $C$ as one of its elements, then the $\text{endTp}$ of $C$ must have the value $\text{tp}$. 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 $\text{endGReq}$ 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 $\text{true}$.

Acyclicity. The underlying graph of a goal net, and the subgoal relation must both be cyclic. 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 $\text{time(t)}$ message from a time point with $\text{early}$ and another one with $\text{late}$. Similarly, the combination $\text{early}$ and $\text{fire}$ 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 $\text{time(t)}$ 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 elaboration, 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 scalable. 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 hierarchical 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 consequences 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.; Durán, F.; Eker, S.; Lincoln, P.; Marti-Oliet, N;


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