Appears in *Proceedings of IJCAI-97*

A Reactive Planner for a Model-based Executive

Brian C. Williams
Computational Sciences Division MS 269-2
NASA Ames Research Center,
Moffett Field, CA 94035 USA
E-mail: williams@ptolemy.arc.nasa.gov

P. Pandurang Nayak
Recom Technologies
NASA Ames Research Center, MS 269-2
Moffett Field, CA 94035 USA
E-mail: nayak@ptolemy.arc.nasa.gov

Abstract

A new generation of reactive, model-based executives are emerging that make extensive use of component-based declarative models to analyze anomalous situations and generate novel sequences for the internal control of complex autonomous systems. Burton, a generative, model-based planner offers a core element that bridges the gap between current and target states within the reactive loop. Burton is a sound, complete, reactive planner that generates a single control action of a valid plan in average case constant time, and compensates for anomalies at every step. Burton will not generate irreversible, potentially damaging sequences, except to effect repairs. We present model compilation, causal analysis, and online policy construction methods that are key to Burton's performance.

Conventional wisdom has largely pushed deductive reasoning out of the reactive control loop for nearly a decade. However, recent search for the surprisingly elusive, hard satisfiability problem foretells a healthy return to deductive methods(Williams & Nayak 1996b; Kautz & Selman 1996) based on RISC-like search engines. This paper pushes this perspective down to reactive time scales, reporting on a model-based planner, called *Burton*, that is at the core of a model-based executive's reactive control loop. By solving the NP hard component of deductive problems at compile time, Burton exploits the expressiveness of NP hard methods, without assuming the risk of falling off the elusive cliff.

Burton's parent model-based executive is particularly well suited to controlling the complex internal behaviors of large scale autonomous systems, we call *immobile robots* (Williams & Nayak 1996a). What distinguishes this executive is its ability to sense and control hidden state variables indirectly, and the use of component models to identify these novel interaction paths. A marriage between this model-based executive and a classical, method-based executive provides a hybrid with an expressive scripting language and an extensive capability to generate novel responses to anomalous situations. Significant parts of this hybrid executive will be demonstrated in late 1998 on NASA's Deep Space One autonomous spacecraft (Pell *et al.* 1997).

The paper begins with an example from the spacecraft domain, and then introduces our concurrent transition system modeling formalism. Next we introduce model-based execution as identifying a current state (mode identification), generating an optimal target state (mode reconfiguration), and generating a control action to move towards the target (model-based reactive planning). The rest of the paper presents the Burton model-based reactive planner through a series of domain restrictions, model compilation, policy construction and online planning algorithms.

Example: autonomous spacecraft

First consider the underlying task. Figure 1 shows an idealized schematic of the main engine subsystem of the Cassini spacecraft and valve driver circuitry. It consists of a helium tank, two propellant tanks, two main engines, regulators, latch valves, and pyro valves. The helium tank pressurizes the propellant tanks. When propellant paths to a main engine are open, the propellants flow into the engine and produce thrust. The pyro valves are used to isolate parts of the engine. They can open or close only once.

Valves are controlled by valve drivers. Commands to the driver are sent via a control unit (VDECU). The driver and VDECU can be on or off, and recoverably or permanently failed. A recoverably failed component can be repaired by resetting it. A valve's state changes as a result of a command only if the corresponding driver and VDECU are on and healthy.

In planning an orbit insertion maneuver, a high-level deliberative planner, *e.g.*, (Muscettola 1994), generates a sequence of behavior goals, such as producing thrust. The reactive executive achieves this goal using its component models to generate control sequences that open the relevant set of valves leading to a main engine. Valves are commanded open indirectly, and the

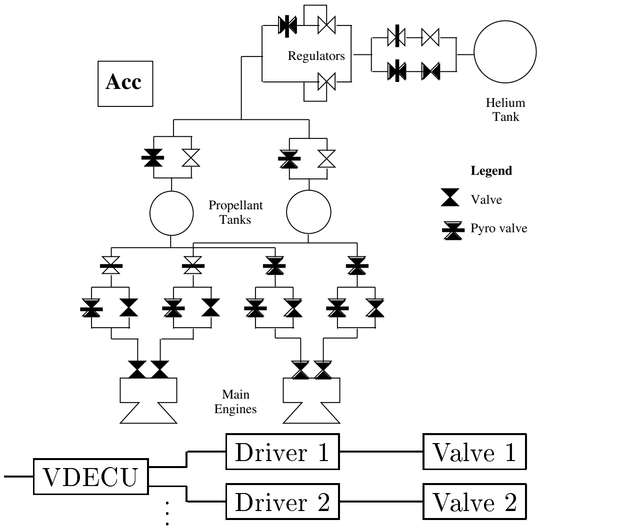


Figure 1: Schematics of engine and valve control circuitry. Valves are closed only when solid black.

executive must ensure that the control unit and driver leading to the valve are on and healthy prior to commanding the valve. Generating sequences to handle a breadth of novel situations requires extensive reasoning about physical processes as well as state changing actions. Doing this reactively is the focus of this paper.

Concurrent transition systems

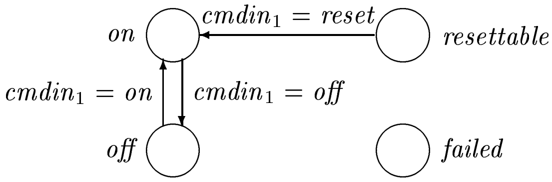
We start by reviewing the propositional, concurrent transition system formalism introduced in (Williams & Nayak 1996b), which represents normal operation, failure, and repair of real-time software and hardware. A *transition system* $\mathcal{S}$ is a tuple $\langle \Pi, \Sigma, \mathcal{T} \rangle$. Π is a set of *variables*, each ranging over a finite domain. Π is partitioned into the set Π_s of *state* variables, the set Π_c of *control* variables, and the set Π_d of *dependent* variables. Σ is the set of *feasible* assignments to variables in Π. Each element of a state variable's domain is categorized nominal or failure. Control variable values are determined exogenously by a controller. A *state* is an assignment to each variable in Π_s. The set Σ_s, the projection of Σ on variables in Π_s, is the set of all feasible states. $\mathcal{T}$ is a finite set of *transitions*. Each transition $\tau \in \mathcal{T}$ is a function $\tau : \Sigma \rightarrow \Sigma_s$, *i.e.*, $\tau(\sigma)$ is the state obtained by applying transition τ to any feasible assignment σ. The transition $\tau_n \in \mathcal{T}$ models the system's nominal behavior, while all other transitions model failures. A repair action is a transition that takes a state variable from a failure to nominal value.

A *trajectory* is a (finite or infinite) sequence of feasible states $S : s_0, s_1, \ldots$ such that for each s_i there is a feasible assignment $\sigma_i \in \Sigma$ which agrees with s_i on assignments to variables in Π_s and $s_{i+1} = \tau(\sigma_i)$ for some $\tau \in \mathcal{T}$. A trajectory that involves only the nominal transition τ_n is called a *nominal trajectory.* A *simple trajectory* does not repeat any state.

A transition system is specified using a restricted subset of propositional temporal logic that uses the $\bigcirc$ operator. The propositions in this logic are all of the form $y = e$, where y is a variable and e is an element in y's domain. $\bigcirc$ is the *next* operator denoting truth in the next state of a trajectory.

We specify the set Σ of feasible assignments by a propositional formula ρ_Σ, *i.e.*, Σ is the set of all variable assignments that satisfy ρ_Σ. A transition τ of $\mathcal{S}$ is specified by a formula ρ_τ, which is a conjunction of *transition specifications* ρ_{τ_i} of the form $\Phi_i \Rightarrow \bigcirc y_i = e_i$, where Φ_i is a propositional formula and y_i is a state variable. A state s_k can follow state s_j using τ if there is an assignment $\sigma_j \in \Sigma$ that agrees with s_j such that for all ρ_{τ_i} if σ_j satisfies Φ_i then s_k assigns e_i to y_i.

Example 1 Driver 1 has a state variable, dr_1, and two dependent variables $cmdin_1$ and $cmdout_1$. The domain of dr_1 is $\{on, off, resettable, failed\}$, $cmdin_1$ is $\{on, off, reset, open, close, none\}$, and $cmdout_1$ is $\{open, close, none\}$. dr_1's transition diagram is:



which is specified by formulae like:

$$dr_1 = resettable \wedge cmdin_1 = reset \Rightarrow \bigcirc dr_1 = on$$

which describes the effect of resetting a driver. The driver's feasible states are specified by formulae like:

$$(dr_1 = on \wedge cmdin_1 = open) \Rightarrow (cmdout_1 = open)$$

The VDECU has control input $drcmdin_1$, whose value is propagated to $cmdin_1$ when the VDECU is *on*:

$$(vdecu_1 = on \wedge drcmdin_1 = reset) \Rightarrow (cmdin_1 = reset)$$

Opening the valve is achieved by turning the VDECU and driver on, and then setting control input $drcmdin_1$ to *open*. The driver's output $cmdout_1$ is used to control the valve's state transitions.

Model-based reactive execution

We view an autonomous system as a high-level *planner* coupled to a low-level *reactive control system.* The planner generates a sequence of configuration goals, each a physical behavior like "achieve thrust." The reactive control system evolves along a trajectory that achieves these configuration goals. Reactive control is achieved using a *reactive executive* that generates a sequence of control actions, based on knowledge of the current state and configuration goal. A control action is an assignment to each control variable in Π_c, for example, corresponding to closing a switch or sending a reset message across a bus. The current state is (partially) observable through a set of variables $\Pi_o \subseteq \Pi_s \cup \Pi_d$, corresponding to sensors, and an observation is an assignment to each variable in Π_o.

Definition 1 A reactive control system is a tuple $\langle \mathcal{S}, \Theta, \mathcal{C} \rangle$, where $\mathcal{S}$ is a transition system, Θ is the initial state of $\mathcal{S}$, and $\mathcal{C}$ is a reactive executive. $\mathcal{C}$ takes as input the initial state Θ, a sequence $\gamma : g_0, g_1, \ldots$ of propositional formulae called *goal configurations*, and a sequence of observations $o : o_0, o_1, \ldots$, and incrementally generates a sequence of control actions $\mu : \mu_0, \mu_1, \ldots$ so that $\mathcal{S}$ evolves along a trajectory $s : s_0, s_1, \ldots$ that satisfies: (a) s_0 is Θ; (b) for each s_i there is an assignment $\sigma_i \in \Sigma$ that agrees with s_i, o_i, and μ_i on the corresponding subsets of variables; and (c) if s_{i+1} is the result of a nominal transition from s_i under μ_i, *i.e.*, $s_{i+1} = \tau_n(\sigma_i)$, then either s_{i+1} satisfies g_i or $\langle s_i, s_{i+1} \rangle$ is the prefix of a simple nominal trajectory that ends in a state s_j that satisfies g_i.

The idea is that a reactive executive continually tries to transition the system toward a state that satisfies the desired goals. It is reactive in the sense that it reacts immediately to changes in goals and to failures, *i.e.*, each control action μ_i is incrementally generated using the new information, o_i and g_i, in each state.

A *model-based executive*, uses a specification of a transition system to determine the desired control sequence in three stages—*mode identification* (MI), *mode reconfiguration* (MR) and *model-based reactive planning* (MRP). MI and MR set up the planning problem, identifying initial and target states, while MRP reactively generates a plan solution. More specifically, MI incrementally generates the set of most likely plant trajectories consistent with the plant transition model and the sequence of observations and control actions. This is maintained as a set of most likely current states. MR uses a plant transition model and the most likely current state generated by MI to determine a reachable target state that satisfies the goal configuration. MRP then generates the first action in a control sequence for moving from the most likely current state to the target state. After that action is performed MI confirms that the intended next state is achieved. MI and MR are discussed in (Williams & Nayak 1996b). This paper focuses on MRP.

A key decision underlying our model-based executive is the focus on the most likely trajectory generated by MI. The difficulty with the more conservative strategy of considering a single control sequence that covers a set of likely states (Williams & Nayak 1996b) is that the different states will frequently require different control sequences. While utility theory can be used to select between different control sequences for one that maximizes success (Friedrich & Nejdl 1992), the cost of generating multiple states and control sequences works against our goal of building a fast reactive executive.

The greedy approach introduces risk: the control action appropriate for the most likely trajectory may be inappropriate, or worse still damaging, if the actual state were otherwise. Furthermore, the reactive focus of the MRP precludes extensive deliberation on the long-term consequences of actions, thus leaving open the possibility that control actions, while not outright harmful, may degrade the system's capabilities. For example, firing a pyro valve is an irreversible action that forever cuts off access to parts of the propulsion system. Such actions should be taken after due deliberation by the high-level planner or human operators. Hence, fundamental to the reliability of our approach is the following requirement:

Requirement 1 MRP considers only reversible control actions, unless the only effect is to repair failures.[1]

Model-based reactive planning

The definition of MRP (Burton) follows from Definition 1 and the functions of MI and MR:

Definition 2 A *model-based reactive planner* (MRP) takes as input a specification of a transition system $\mathcal{S}$, a most likely initial state s_i (from MI), and a lowest cost target state t_i (from MR) that satisfies goal g_i. The MRP generates a control action μ_i such that for any assignment $\sigma_i \in \Sigma$ that agrees with s_i and μ_i, the state $s_{i+1} = \tau_n(\sigma_i)$ is either the target state t_i or $\langle s_i, s_{i+1} \rangle$ is the prefix of a simple nominal trajectory that ends in t_i.

Given the similarity of transitions and STRIPS operators, Burton's problem appears similar to STRIPS planning (Weld 1994). However, the critical difference is the distinction between classical planning and machine control. The primitive control action in a STRIPS plan is to invoke a plan operator, which *directly* modifies the state. Model-based planners, on the other hand, exert control by establishing values for control variables, which interact with internal state

[1] While repairing a failure is irreversible, it is important to allow a reactive executive to repair failures.

variables *indirectly* through co-temporal, physical interactions (ρ_Σ). This extension adds enormous expressivity to the planner.

This extension equally adds to the challenge of achieving reactivity. We eliminate intractability at modeling time through an automated model compilation method and four simple modeling requirements.

Compiling transition models

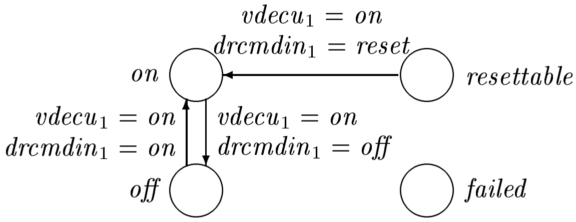
Recall that a transition is specified by a conjunction of formulae $\Phi_i \Rightarrow \bigcirc y_i = e_i$. What distinguishes this from a simple transition is that the formula Φ_i can contain propositions involving dependent variables that are not directly controllable, but depend on control variables through the arbitrary propositional formulae of ρ_Σ. This introduces a potential computational cliff.

To simplify the transition system, we first *compile away* all dependent variables and corresponding interactions of ρ_Σ, by generating all *prime implicants* of $\bigcirc y_i = e_i$ that do not contain dependent variables. More precisely, an implicant of $\bigcirc y_i = e_i$ is a conjunction I of propositions involving state and control variables such that the transition system specification, $\rho_\Sigma \cup \bigcup_{\tau \in \mathcal{T}} \rho_\tau$, entails the formula $I \Rightarrow \bigcirc y_i = e_i$. I is a prime implicant if no sub-conjunction of I is an implicant. Given that dependent variables cannot be directly controlled, either by control actions or by transitions, the prime implicants of $\bigcirc y_i = e_i$ completely characterize the conditions under which y_i has the value e_i in the next state. For example, one prime implicant of $\bigcirc dr_1 = on$ (from Example 1) is

$$dr_1 = resettable \wedge vdecu_1 = on \wedge drcmdin_1 = reset$$

While prime implicant generation is NP-hard, in practice even sizable implicants can be generated very fast. For example, we use an abductive best-first search (Williams & Nayak 1996b) to generate implicants from a spacecraft model consisting of over $12,000$ clauses in about 40 seconds on a Sparc 20. We compute these implicants at compile-time, thus avoiding the risk of falling over the computational cliff at run-time, while preserving expressivity in the modeling language.

The set of prime implicants constitutes the compiled transition specification, where each implicant specifies a transition for a single state variable. The transition specification of a single variable y_i is the set of prime implicants of $\bigcirc y_i = e_{ik}$, for every e_{ik} in y_i's domain. Antecedents of transition specifications now contain only state and control variables. For a transition of y_i, each antecedent involving a state variable other than y_i is called a *state condition,* and each antecedent involving a control variable is called a *control condition.* If y_i is not in the antecedent, we replicate the transition for every assignment of y_i. The transition diagram for the driver after compilation is:



The correspondence to a STRIPS operator is straightforward: state conditions, including antecedent y_i, form preconditions; the add (delete) list contains the next (previous) value of y_i. The transition is invoked by asserting all control conditions.

Exploiting properties of designed systems

Albeit simpler than the original specification, these compiled transitions have a complication not found in STRIPS. STRIPS operators are explicitly invoked, one at a time. In the above, a control action can invoke more than one transition. Furthermore, a transition can occur spontaneously, and cannot be prevented, if its antecedent contains no control variables.

While arbitrary transition systems can manifest these properties, physical hardware is typically designed to behave like STRIPS operators. It is usually the case that each state variable is separately commanded and state variables maintain their values in the absence of explicit commands. The following requirement prevents spontaneous state change:

Requirement 2 Each control variable has an idling assignment, and no idling assignment appears in any transition. The antecedent of every transition includes a (non-idling) control condition.

For example, $drcmdin_1$ has idling value *none*, and the prime implicant of $\bigcirc dr_1 = on$ mentioned above contains the non-idling $drcmdin_1 = reset$. Exploiting this restriction, all state changes can be prevented by assigning every control variable its idling assignment. The following additional restriction guarantees that transitions can be individually invoked:

Requirement 3 No set of control conditions of one transition is a proper subset of the control conditions of a different transition.

A single transition is invoked by asserting its control conditions, and assigning all other control variables their idle assignment. Jointly these two requirements reduce MRP to STRIPS planning.

Online component of reactive planning

Given a compiled transition system S', Burton quickly generates the first control action of a valid plan, given an initial state assignment Θ and a set of goal assignments γ.[2] Burton meets five desiderata. First, it only generates *non-destructive actions*, *i.e.*, an action will never undo the effects of previous actions that are still needed to achieve top-level goals. Second, Burton will not propose actions that lead to *deadend plans*, *i.e.*, it will not propose an action to achieve a subgoal when one of the sibling subgoals is unachievable. Third, Burton is *complete, i.e.*, if a planning problem that satisfies Requirements 1-4 is solvable, then Burton will generate a plan. However, Burton is not guaranteed to generate all valid plans. Fourth, Burton ensures *progress* to a goal, except when execution anomalies interfere, *i.e.*, the nominal trajectory traversed by Burton for a fixed target is loop free. Fifth, Burton operates at *reactive time scales*—its average runtime complexity is constant. This speed is essential to providing a model-based executive with response times comparable to traditional executives (Firby 1987; Simmons 1994).

Burton avoids runtime search, requires no algorithms for threat detection, and expends no effort determining future actions or planning for subgoals that are not supported by the first action. Traditional planners need such mechanisms to avoid destructive actions and deadend plans (Weld 1994). Burton accomplishes this speedup by exploiting the requirement, stated earlier, that all actions except repairs be reversible, and by exploiting certain topological properties of component connectivity that frequently occur in designed systems. The development of Burton's basic sequencing algorithm is the topic of the next three subsections, with the introduction of repair actions introduced in the fourth subsection. Finally, to achieve average case constant time Burton precompiles plans into reactive policies, without requiring enormous amounts of storage. This is the topic of the fifth subsection.

Exploiting causality

A major leverage point comes from exploiting the topological properties of component connectivity. The input/output connections of the compiled plant frequently do not contain feedback loops. When they do occur they are typically local and can easily be eliminated through careful modeling. To be precise:

Definition 3 A *causal graph* $\mathcal{G}$ for (compiled) transition system $\mathcal{S}$ is a directed graph whose vertices are the state variables of $\mathcal{S}$. $\mathcal{G}$ contains an edge from $v1$ to $v2$ if $v1$ occurs in the antecedent of one of $v2$'s transitions.

Requirement 4 The causal graph must be *acyclic.*

[2]Here on, each goal is a target variable assignment generated by MR, not the configuration goal input to MR.

The causal graph for the valve circuitry follows the schematic, and is acyclic.

The basic idea underlying Burton is to solve a conjunction of goals by working "upstream" along the acyclic causal graph, *i.e.*, it completes a conjunct $y_i = e_{ik}$ before conjunct $y_j = e_{jk}$ when y_j precedes y_i in the causal graph. For example, suppose the initial state has $vlv_1 = closed \wedge dr_1 = off$, and the target has $vlv_1 = open \wedge dr_1 = off$. Burton first focuses on $vlv_1 = open$ since the valve is downstream of the driver. As we will see, this upstream progression guarantees that Burton generates non-destructive actions without an explicit threat detection mechanism.

Burton generates a next control action with the NextAction function, shown below. This takes an initial state Θ, a (partial) target state γ (sorted by topological number as discussed later), a compiled transition system S' and the flag **top?** to indicate a top-level call. NextAction returns a next control action, **Failure** if no plan exists, or **Success** if the initial state is a target state. A control action is a partial set of control assignments. All control variables not mentioned are assigned their idle value.

Online Algorithm: NextAction(Θ,γ,S',**top?**)

1. **Solvable Goals?:** When **top?** = **True**, unless each goal $g \in \gamma$ is labeled **Reversible**, return **Failure**.
2. **Select unachieved goal:** Find an unachieved goal assignment with the lowest topological number. Goal $y = e_f \in \gamma$ is unachieved if it differs from y's initial assignment $y = e_i \in \Theta$. If all achieved return **Success**.
3. **Select next transition:** Let t_y be the transition graph in S for goal variable y. Find a path p in t_y from e_i to e_f along transitions labeled **Allowed**. Let SC and CC be the state and control conditions of the first transition along p.
4. **Enable transition:** Control = NextAction(Θ,SC,S',False). If Control = **Success** then state conditions SC are already satisfied, return CC to effect transition. Otherwise Control contains control assignments to progress on SC. Return Control.

Line 1 tests whether or not a conjunction of top-level goals can be achieved, as explained in the subsection after next. This involves a simple table lookup to see if each goal is labeled **Reversible**. Burton only introduces subgoals that can be achieved, hence the test is needed at the top level only. Line 2 works upstream along the causal graph of variables selecting the next unsatisfied (sub)goal assignment to be achieved. This upstream progression is achieved by exploiting a topological ordering, as explained in the next subsection.

Line 3 takes the first step towards achieving the selected goal. Given an initial assignment $y = e_i$ and a component transition system for y, a goal assignment $y = e_f$ is achieved by traversing a path along transitions of y from e_i to e_f. Respecting Restriction 1, Burton only traverses transitions with reversible effects or that perform a repair. As explained in the next section, the transitions that satisfy this restriction are those labeled **Allowed**. Line 3 identifies the first transition along this path from e_i to e_f. Line 4 subgoals on the state conditions of this first transition, which results in Burton moving further upstream. If one or more state conditions is unsatisfied then a next control action is computed in Line 4, and returned. If all state conditions are satisfied, then the transition is ready to be traversed and Line 4 returns the transition's control conditions as the next control action.

Avoiding destructive actions

Burton avoids generating destructive control actions (desiderata 1) by exploiting the acyclic nature of the causal graph. The only variables needed to achieve an assignment $y = e$ are y's ancestors in the graph. For example, turning on the driver requires use of the VDECU but not the valve. In addition, requirements 2 and 3 guarantee that invoking transitions for y and its ancestors, when performed one at a time, will not affect any other state variables.

This suggests that Burton can achieve a conjunction of goal assignments in an order that moves along the causal graph from descendants to ancestors, *i.e.*, a goal conjunct is achieved only after conjuncts that are its descendants. For example, $dr_1 = \mathit{off} \wedge vlv_1 = \mathit{open}$ is achieved by working on $vlv_1 = \mathit{open}$ then $dr_1 = \mathit{off}$. The same ordering holds for conjunctive subgoals, that is, state conditions of required transitions.

Destructive subgoal interaction may occur when a variable appears upstream as a subgoal to two conjuncts. To avoid this danger subgoals are achieved in depth first order, achieving one conjunct before starting on a second. For example, $dr_1 = \mathit{off} \wedge vlv_1 = \mathit{open}$ is achieved by turning on the driver, opening the valve, and finally turning the driver off. Achieving subgoals in depth first order also ensures that Burton always makes progress towards the goal (desiderata 4).

Depth first goal progression, together with the goal ordering constraint, is sufficient to ensure non-destructive actions. Depth first progression is imposed by Line 4 of NextAction. Line 2 imposes the ordering constraint without runtime cost through *topological numbering*. At compile time each state variable is given a topological number (tn), by performing a depth first search of the graph and numbering variables on the way out. For example, $tn(vlv_1) = 1, tn(dr_1) = 2, tn(vlv_2) = 3, tn(dr_2) = 4$, and $tn(vdecu_1) = 5$. Topological numbering imposes a total ordering that satisfies the constraint $tn(A) < tn(B)$ whenever A is a strict descendant of B in the causal graph. Hence the proper order of goal achievement for all conjunctive subgoals is determined at compile time by sorting the conditions of each transition by increasing topological number. Burton respects an upstream progression in line 2, simply by working successively through the sorted list of conditions.

Avoiding deadends through reversibility

Requirement 1 stated that Burton only perform reversible transitions, desiderata 2 stated that Burton avoid deadend plans, that is plans with unachievable subgoals, and desiderata 3 specified completeness. To be reactive Burton must achieve these without search. Each hinges on the following lemma:

Lemma 1 *$A \wedge B$ is reachable from Θ by reversible transitions exactly when A and B are separately reachable from Θ by reversible transitions.*

Proof: Assume without loss of generality that $tn(A) < tn(B)$. We previously showed that if A is achieved first, it won't be disturbed while achieving B. The transitions used to achieve A are reversible, hence *if nothing else* each variable other than A can be restored to its value in Θ and then B can be achieved. □

Suppose Burton has labeled as **Reversible** every assignment that is reachable from initial state Θ using **Reversible** transitions. By Lemma 1 a conjunction of goal assignments is achievable exactly when each conjunct is marked **Reversible**. Hence Burton can determine plan achievement at runtime simply by using one table lookup per top-level goal (line 1, NextAction).

Deadend plans, hence search, are eliminated by removing the possibility of unachievable subgoals. Assume Burton labels a transition **Allowed** only if its state conditions are each labeled **Reversible**. Then sequences of **Allowed** transitions between **Reversible** assignments contain no unachievable subgoals. These are the transitions generated by line 3 of NextAction. By lemma 1 all other sequences lead to deadends or involve irreversible transitions, hence Burton generates some plan if one exists (desiderata 3). Of course Burton can't generate all plans since subgoal interleaving isn't allowed. Finally, since depth first progression generates the control sequence in order, Burton generates the first control action without wasting work on the rest of the plan, hence achieving reactivity.

The labeling of assignments and transitions is precomputed for compiled transition system S' with initial state Θ by LabelSystem, and is linear in the size of S'.

Preprocess Algorithm: LabelSystem(S',Θ)
For each state variable y of S' in decreasing topological order:

1. For each transition τ_y of y, label τ_y **Allowed** if all its state conditions are labeled **Reversible**.
2. Compute the strongly connected components (SCCs) of the **Allowed** transitions of y.
3. Find y's initial value $y = e_i \in \Theta$. Label each assignment in the SCC of $y = e_i$ as **Reversible**.

LabelSystem starts at the roots of the causal graph, using topological numbering to move to a descendant only after its ancestors have been processed. Line 1 simply executes the definition of **Allowed**. Note when y_i is a root, none of its transitions contain state conditions, and hence are trivially **Allowed**. Lines 2 and 3 identify **Reversible** assignments. An assignment $y = e_k$ can be reversibly achieved if there exists a path along **Allowed** transitions from initial value e_i to e_k and back. Equivalently, the set of y's reversible assignments is the strongly connected component (SCC) of y's **Allowed** transition that contains e_i.

For example, starting with initial value *off*, the SCC for the VDECU, hence its **Reversible** assignment set, is $\{off, on\}$, with both $resettable$ and $failed$ excluded. Next, the driver has **Allowed** transitions between *on* and *off* due to the VDECU's SCC. With initial value *on*, the SCC for the driver is $\{off, on\}$.

Finally, note that LabelSystem is called infrequently. Actions generated by Burton only move within the SCCs of **Reversible** assignments, leaving the labeling unchanged. A relabeling is needed only if an exogenous action or failure moves Θ to an assignment that was not labeled **Reversible**.

Failure states and repair

To dramatically expand Burton's utility we incorporate repair actions. The occurrence of failures are outside Burton's control, since there are no nominal transitions that lead to failure. Hence a repair sequence is irreversible, albeit essential, and thus not covered by the development thus far. We extend Burton to permit repair sequences if they minimize irreversible effects. Burton never uses a failure to achieve a goal assignment if the failure is repairable. However, if it is not repairable, then Burton is allowed to exploit the component's faulty state. For example, suppose a switch is needed to be open, and it is permanently stuck open. Since stuck-open is irrepairable but has the desired effect, Burton exploits the failure mode.

Relaxing the reversibility constraint, if a state variable y is assigned a failure e_f, then Burton is permitted to traverse a sequence of allowed transitions from e_f to a nominal assignment, when such a path exists. If only one path exists, then y's reversible assignments are defined to be those in the SCC of the *first nominal assignment* reached along the path. For example if the driver is initially at $resettable$ then it may transition to *on* using $reset$, and the SCC is $\{off, on\}$. If no path exists to a nominal assignment, then the reversible assignments are those in the SCC that contains e_f. For example, if the driver is at $failed$, then the SCC is simply $\{failed\}$. Although not discussed here, Burton also handles the case where multiple paths to different SCCs exist.

Although Burton can now traverse irreversible transitions to effect repair, none of the assignments along this trajectory, up to the selected SCC can be used to satisfy a state condition or goal assignment. Hence this extension does not endanger Burton's previously discussed properties.

Generating concurrent policies

The runtime complexity of NextAction is thus far $O(e * m)$, where e is the maximum number of transitions in a single component transition system, and m is the maximum depth of the causal graph. The cost m is incurred on line 4, while subgoaling on assignments of upstream variables, and cost e is incurred on line 3, while searching for a path through transitions. Burton reduces cost to $O(m)$ by constructing for each variable y a *feasible policy* π_y. π_y maps pairs $\langle e_i, e_f\rangle$ to the sorted conditions of the first transition along a path from e_i to e_f. If y has n values, π_y is simply an n^2 table computed in $O(n * e)$ time, where entries for each e_i are determined by computing a spanning tree directed towards e_f that connects all values labeled **Reversible** by transitions labeled **Allowed**. Line 3 of NextAction becomes the table lookup:

$$3' : \langle SC, CC\rangle = \pi_y(e_i, e_f)$$

vlv_1	Target	
Current	*open*	*closed*
open	**Idle**	$dr_1 = on$ $drcmdin_1 = close$
closed	$dr_1 = on,$ $drcmdin_1 = open$	**Idle**
stuck	**Failure**	**Failure**

dr_1	Target	
Current	*on*	off
on	**Idle**	$drcmdin_1 = off$
off	$drcmdin_1 = on$	**Idle**
resettable	$drcmdin_1 = reset$	$drcmdin_1 = reset$

The driver and valve policies are shown above ($vdecu_1 = on$ is assumed and not shown). Suppose MI identifies the initial state $\{dr_1 = off, vlv_1 = closed\}$

and MR provides target $\{vlv_1 = open, dr_1 = off\}$, in topological order. Selecting the first target assignment, Burton looks up $vlv_1 : closed \rightarrow open$, and gets $\{dr_1 = on, drcmdin_1 = open\}$. The first is an unsatisfied state condition, so Burton looks up $\{dr_1 : off \rightarrow on\}$, getting $\{drcmdin_1 = on\}$. This control assignment is then invoked.

Burton's feasible policies are analogous to optimal policies in control theory. An important difference is that Burton constructs a set of concurrent policies, rather than a single policy for the complete state space. The latter grows exponential in the number of state variables, and is infeasible for models like the spacecraft, which contain over 80 state variables and well over 2^{80} states. In contrast, the concurrent policies grow only linearly in the number of state variables.

For a fixed target and no intervening failures, Burton generates successive control actions as a depth first traversal through the policy tables. This traversal maps out a subgoal tree, and Burton maintains an index into this tree during successive calls. To generate a sequence Burton traverses each tree edge exactly twice and generates one control action per vertex. Since the number of edges in a tree is bounded by the number of vertices, the *amortized average case complexity of generating each control action is a constant.* Desiderata 5, reactivity, is achieved.

Related work and conclusions

Burton combines causal models used in model-based reasoning with state transitions used in planning. Other researchers have combined model-based diagnosis with planning, primarily to generate repair plans (Friedrich & Nejdl 1992; Sun & Weld 1993). The main difference is that these systems include a STRIPS (rather than model-based) planning component with utility-theoretic measures for selecting amongst alternate plans. Their computational complexity make them inapplicable for on-board reactive execution.

Burton differs from traditional STRIPS planners (Weld 1994) in that plan operators (transitions) are generated by a compilation process from an underlying causal model of the system that includes both within and across state constraints (ρ_Σ and the ρ_τ, respectively). Furthermore, the compiled transition system is a specialized version of the general planning problem that Burton solves by a worst-case linear time, average case constant time algorithm. In contrast, general planning is PSPACE-complete. Korf (1987) defines a set of subgoals to be *serializable* if they can be solved in order without ever violating a subgoal solved earlier in the order. The requirement that the causal graph is acyclic ensures that Burton is only presented serializable sets of subgoals.

Finally, traditional reactive executives (Firby 1987; Simmons 1994) differ from Burton's model-based executive in that the former use explicitly scripted procedures to provide reactive execution, while the latter uses deductive reasoning from a causal model that combines an off-line compilation phase with an on-line policy generation phase. Other differences include the fact that traditional executives provide a richer set of control structures such as parallel execution, while Burton's executive provides a more sophisticated diagnosis and monitoring capability embodied in MI.

In summary, Burton is a sound and complete generative planner that uses expressive transition system models to provide the reactive sequencing capability of a model-based executive. It exploits off-line model compilation and the topological properties of component connectivity to incrementally generate control actions in average case constant time. These control actions are guaranteed to be non-destructive, while ensuring progress towards the goal and avoiding deadend plans.

Acknowledgements: We would like to thank Jim Kurien and Dan Weld for helpful comments on the paper.

References

Firby, R. 1987. An investigation into reactive planning in complex domains. In *Procs. of AAAI-87*, 202–206.

Friedrich, G., and Nejdl, W. 1992. Choosing observations and actions in model-based diagnosis/repair systems. In *Procs. of KR-92*, 489–498.

Kautz, H., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Procs. of AAAI-96*, 1194–1201.

Korf, R. 1987. Planning as search: A quantitative approach. *Artificial Intelligence* 33(1):65–88.

Muscettola, N. 1994. HSTS: Integrating planning and scheduling. In Fox, M., and Zweben, M., eds., *Intelligent Scheduling.* Morgan Kaufmann.

Pell, B.; Bernard, D.; Chien, S.; Gat, E.; Muscettola, N.; Nayak, P.; Wagner, M.; and Williams, B. 1997. An autonomous spacecraft agent prototype. In *Procs. of the First Int. Conf. on Autonomous Agents.*

Simmons, R. 1994. Structured control for autonomous robots. *IEEE Trans. on Robotics and Automation* 10(1).

Sun, Y., and Weld, D. 1993. A framework for model-based repair. In *Procs. of AAAI-93*, 182–187.

Weld, D. 1994. An introduction to least commitment planning. *AI Magazine* 15(4):27–61.

Williams, B., and Nayak, P. 1996a. Immobile robots: AI in the new millennium. *AI Magazine* 17(3):16–35.

Williams, B., and Nayak, P. 1996b. A model-based approach to reactive self-configuring systems. In *Procs. of AAAI-96*, 971–978.