A Model-based Approach to Reactive Self-Configuring Systems*

Brian C. Williams and P. Pandurang Nayak
Recom Technologies, NASA Ames Research Center, MS 269-2
Moffett Field, CA 94305 USA
E-mail: `williams,nayak@ptolemy.arc.nasa.gov`

Abstract

This paper describes Livingstone, an implemented kernel for a model-based reactive self-configuring autonomous system. It presents a formal characterization of Livingstone's representation formalism, and reports on our experience with the implementation in a variety of domains. Livingstone provides a reactive system that performs significant deduction in the sense/response loop by drawing on our past experience at building fast propositional conflict-based algorithms for model-based diagnosis, and by framing a model-based configuration manager as a propositional feedback controller that generates focused, optimal responses. Livingstone's representation formalism achieves broad coverage of hybrid hardware/software systems by coupling the transition system models underlying concurrent reactive languages with the qualitative representations developed in model-based reasoning. Livingstone automates a wide variety of tasks using a single model and a single core algorithm, thus making significant progress towards achieving a central goal of model-based reasoning. Livingstone, together with the HSTS planning and scheduling engine and the RAPS executive, has been selected as part of the core autonomy architecture for NASA's first New Millennium spacecraft.

Introduction and Desiderata

NASA has put forth the challenge of establishing a "virtual presence" in space through a fleet of intelligent space probes that autonomously explore the nooks and crannies of the solar system. This "presence" is to be established at an Apollo-era pace, with software for the first probe to be completed in 1997 and the probe (Deep Space 1) to be launched in 1998. The final pressure, low cost, is of an equal magnitude. Together this poses an extraordinary challenge and opportunity for AI. To achieve robustness during years in the harsh environs of space the spacecraft will need to radically reconfigure itself in response to failures, and then navigate around these failures during its remaining days. To achieve low cost and fast deployment, one-of-a-kind space probes will need to be plugged together quickly, using component-based models wherever possible to automatically generate flight software. Finally, the space of failure scenarios and associated responses will be far too large to use software that requires prelaunch enumeration of all contingencies. Instead, the spacecraft will have to reactively think through the consequences of its reconfiguration options.

We made substantial progress on each of these fronts through a system called *Livingstone*, an implemented kernel for a model-based reactive self-configuring autonomous system. This paper presents a formalization of the reactive, model-based configuration manager underlying Livingstone. Several contributions are key. First, the approach unifies the dichotomy within AI between deduction and reactivity (Agre & Chapman 1987; Brooks 1991). We achieve a reactive system that performs significant deduction in the sense/response loop by drawing on our past experience at building fast propositional conflict-based algorithms for model-based diagnosis, and by framing a model-based configuration manager as a propositional feedback controller that generates focused, optimal responses. Second, our modeling formalism represents a radical shift from first order logic, traditionally used to characterize model-based diagnostic systems. It achieves broad coverage of hybrid hardware/software systems by coupling the transition system models underlying concurrent reactive languages (Manna & Pnueli 1992) with the qualitative representations developed in model-based reasoning. Reactivity is respected by restricting the model to concurrent propositional transition systems that are synchronous. Third, the long held vision of model-based reasoning has been to use a single central model to support a diversity of engineering tasks. For model-based autonomous systems this means using a single model to support a variety of execution tasks including tracking planner goals, confirming hardware modes, reconfiguring hardware, detecting anomalies, isolating faults, diagnosis, fault recovery, and safing. Livingstone automates all these tasks using a single model and a single core algorithm, thus making significant progress towards achieving the model-based vision.

Livingstone, integrated with the HSTS planning and

*Authors listed in reverse alphabetical order.

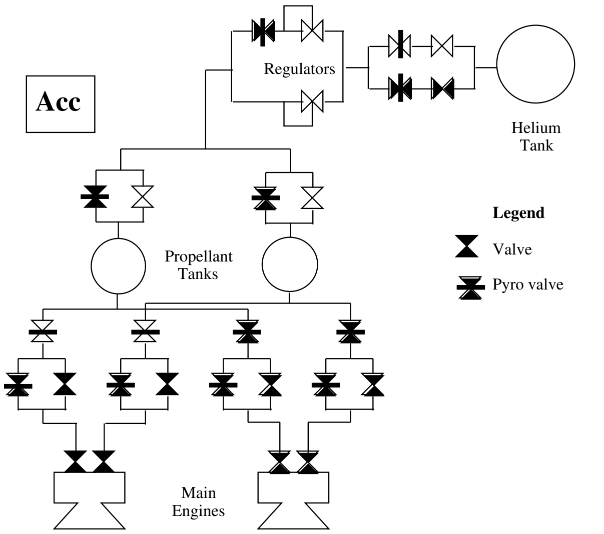

Figure 1: Engine schematic. Valves in solid black are closed, while the others are open.

scheduling system (Muscettola 1994) and the RAPS executive (Firby 1995), was demonstrated to successfully navigate the simulated NewMaap spacecraft into Saturn orbit during its one hour insertion window, despite about half a dozen failures. Consequently, Livingstone, RAPS, and HSTS have been selected to fly Deep Space 1, forming the core autonomy architecture of NASA's New Millennium program. In this architecture (Pell *et al.* 1996) HSTS translates high-level goals into partially-ordered tokens on resource timelines. RAPS executes planner tokens by translating them into low-level spacecraft commands while enforcing temporal constraints between tokens. Livingstone tracks spacecraft state and planner tokens, and reconfigures for failed tokens.

The rest of the paper is organized as follows. In the next section we introduce the spacecraft domain and the problem of configuration management. We then introduce transition systems, the key formalism for modeling hybrid concurrent systems, and a formalization of configuration management. Next, we discuss model-based configuration management and its key components: mode identification and mode reconfiguration. We then introduce algorithms for statistically optimal model-based configuration management using conflict-directed best-first search, followed by an empirical evaluation of Livingstone.

Example: Autonomous Space Exploration

Figure 1 shows an idealized schematic of the main engine subsystem of Cassini, the most complex spacecraft built to date. It consists of a helium tank, a fuel tank, an oxidizer tank, a pair of main engines, regulators, latch valves, pyro valves, and pipes. The helium tank pressurizes the two propellant tanks, with the regulators acting to reduce the high helium pressure to a lower working pressure. When propellant paths to a main engine are open, the pressurized tanks forces fuel and oxidizer into the main engine, where they combine and spontaneously ignite, producing thrust. The pyro valves can be fired exactly once, i.e., they can change state exactly once, either from open to closed or vice versa. Their function is to isolate parts of the main engine subsystem until needed, or to isolate failed parts. The latch valves are controlled using valve drivers (not shown), and the accelerometer (Acc) senses the thrust generated by the main engines.

Starting from the configuration shown in the figure, the high-level goal of producing thrust can be achieved using a variety of different configurations: thrust can be provided by either main engine, and there are a number of ways of opening propellant paths to either main engine. For example, thrust can be provided by opening the latch valves leading to the engine on the left, or by firing a pair of pyros and opening a set of latch valves leading to the engine on the right. Other configurations correspond to various combinations of pyro firings. The different configurations have different characteristics since pyro firings are irreversible actions and since firing pyro valves requires significantly more power than opening or closing latch valves.

Suppose that the main engine subsystem has been configured to provide thrust from the left main engine by opening the latch valves leading to it. Suppose that this engine fails, e.g., by overheating, so that it fails to provide the desired thrust. To ensure that the desired thrust is provided even in this situation, the spacecraft must be transitioned to a new configuration in which thrust is now provided by the main engine on the right. Ideally, this is achieved by firing the two pyro valves leading to the right side, and opening the remaining latch valves (rather than firing additional pyro valves).

A configuration manager constantly attempts to move the spacecraft into lowest cost configurations that achieve a set of high-level dynamically changing goals. When the spacecraft strays from the chosen configuration due to failures, the configuration manager analyzes sensor data to identify the current configuration of the spacecraft, and then moves the spacecraft to a new configuration which, once again, achieves the desired configuration goals. In this sense a configuration manager is a discrete control system that ensures that the spacecraft's configuration always achieves the set point defined by the configuration goals.

Models of Concurrent Processes

Reasoning about a system's configurations and autonomous reconfiguration requires the concepts of operating and failure modes, repairable failures, and configuration changes. These concepts can be expressed in a state diagram: repairable failures are transitions from a failure state to a nominal state; configuration changes are between nominal states; and failures are transitions from a nominal to a failure state.

Selecting a restricted, but adequately expressive, formalism for describing the configurations of a hybrid hardware/software system is essential to achieving the competing goals of reactivity and expressivity. First-order formulations, though expressive, are overly general and do not lend themselves to efficient reasoning. Propositional formulations lend themselves to efficient reasoning, but are inadequate for representing concepts such as state change. Hence, we use a concurrent transition system formulation and a temporal logic specification (Manna & Pnueli 1992) as a starting point for modeling hardware and software. Components operate concurrently, communicating over "wires," and hence can be modeled as concurrent communicating transition systems. Likewise, for software routines, a broad class of reactive languages can be represented naturally as concurrent transition systems communicating through shared variables.

Where our model differs from that of Manna & Pnueli, is that reactive software procedurally modifies its state through explicit variable assignments. On the other hand, a hardware component's behavior in a state is governed by a set of discrete and continuous declarative constraints. These constraints can be computationally expensive to reason about in all their detail. However, experience applying qualitative modeling to diagnostic tasks for digital systems, copiers, and spacecraft propulsion, suggests that simple qualitative representations over small finite domains are quite adequate for modeling continuous and discrete systems. The added advantage of using qualitative models is that they are extremely robust to changes in the details of the underlying model. Hence behaviors within states are represented by constraints over finite domains, and are encoded as propositional formulae which can be reasoned with efficiently.

Other authors such as (Kuipers & Astrom 1994; Nerode & Kohn 1993; Poole 1995; Zhang & Mackworth 1995) have been developing formal methods for representing and reasoning about reactive autonomous systems. The major difference between their work and ours is our focus on fast reactive inference using propositional encodings over finite domains.

Transition systems

We model a concurrent process as a *transition system*. Intuitively, a transition system consists of a set of state variables defining the system's state space and a set of transitions between the states in the state space.

Definition 1 A *transition system* $\mathcal{S}$ is a tuple $\langle \Pi, \Sigma, \mathcal{T} \rangle$, where

- Π is a finite set of *state variables*. Each state variable ranges over a finite domain.
- Σ is the *feasible* subset of the *state space*. Each state in the state space assigns to each variable in Π a value from its domain.
- $\mathcal{T}$ is a finite set of *transitions* between states. Each transition $\tau \in \mathcal{T}$ is a function $\tau : \Sigma \rightarrow 2^{\Sigma}$ representing a state transforming action, where $\tau(s)$ denotes the set of possible states obtained by applying transition τ in state s.

A *trajectory* for $\mathcal{S}$ is a sequence of feasible states $\sigma : s_0, s_1, \ldots$ such that for all $i \geq 0$, $s_{i+1} \in \tau(s_i)$ for some $\tau \in \mathcal{T}$. In this paper we assume that one of the transitions of $\mathcal{S}$, called τ_n, is designated the *nominal* transition, with all other transitions being *failure* transitions. Hence in any state a component may non-deterministically choose to perform either its nominal transition, corresponding to correct functioning, or a failure transition, corresponding to a component failure. Furthermore in response to a successful repair action, the nominal transition will move the system from a failure state to a nominal state.

A transition system $\mathcal{S} = \langle \Pi, \Sigma, \mathcal{T} \rangle$ is specified using a propositional temporal logic. Such specifications are built using *state formulae* and the $\bigcirc$ operator. A state formula is an ordinary propositional formula in which all propositions are of the form $y_k = e_k$, where y_k is a state variable and e_k is an element of y_k's domain. $\bigcirc$ is the *next* operator of temporal logic denoting truth in the next state in a trajectory.

A state s defines a truth assignment in the natural way: proposition $y_k = e_k$ is true iff the value of y_k is e_k in s. A state s satisfies a state formula ϕ precisely when the truth assignment corresponding to s satisfies ϕ. The set of states characterized by a state formula ϕ is the set of all states that satisfy ϕ. Hence, we specify the set of feasible states of $\mathcal{S}$ by a state formula $\rho_{\mathcal{S}}$.

A transition τ is specified by a formula ρ_τ, which is a conjunction of formulae ρ_{τ_i} of the form $\Phi_i \Rightarrow \bigcirc\Psi_i$, where Φ_i and Ψ_i are state formulae. A feasible state s_k can follow a feasible state s_j in a trajectory of $\mathcal{S}$ using transition τ iff for all formulae ρ_{τ_i}, if s_j satisfies the antecedent of ρ_{τ_i}, then s_k satisfies the consequent of ρ_{τ_i}. A transition τ_i that models a formula ρ_{τ_i} is called a *subtransition*. Hence taking a transition τ corresponds to taking all its subtransitions τ_i.

Note that this specification only adds the $\bigcirc$ operator to standard propositional logic. This severely constrained use of temporal logic is an essential property that allows us to perform deductions reactively.

Example 1 The transition system corresponding to a valve driver consists of 3 state variables $\{mode, cmdin, cmdout\}$, where *mode* represents the driver's mode (*on*, *off*, *resettable* or *failed*), *cmdin* represents commands to the driver and its associated valve (*on*, *off*, *reset*, *open*, *close*, *none*), and *cmdout* represents the commands output to its valve (*open*, *close*, or *none*). The feasible states of the driver are specified by the formula

$$\begin{array}{lcl} mode = on & \Rightarrow & (cmdin = open \Rightarrow cmdout = open) \\ & & \wedge\ (cmdin = close \Rightarrow cmdout = close) \\ & & \wedge\ ((cmdin \neq open \wedge\ cmdin \neq close) \\ & & \quad \Rightarrow cmdout = none) \\ mode = off & \Rightarrow & cmdout = none \end{array}$$

together with formulae like $(mode \neq on) \vee (mode \neq off)$, ...that assert that variables have unique values. The driver's nominal transition is specified by the following set of formulae:

$$((mode = on) \vee (mode = off)) \wedge (cmdin = off) \Rightarrow \bigcirc(mode = off)$$
$$((mode = on) \vee (mode = off)) \wedge (cmdin = on) \Rightarrow \bigcirc(mode = on)$$
$$(mode \neq failed) \wedge (cmdin = reset) \Rightarrow \bigcirc(mode = on)$$
$$(mode = resettable) \wedge (cmdin \neq reset) \Rightarrow \bigcirc(mode = resettable)$$
$$(mode = failed) \Rightarrow \bigcirc(mode = failed)$$

The driver also has two failure transitions specified by the formulae $\bigcirc(mode = failed)$ and $\bigcirc(mode = resettable)$, respectively.

Configuration management

We view an autonomous system as a combination of a high-level *planner* and a reactive *configuration manager* that controls a plant (Figure 2). The planner generates a sequence of hardware configuration goals. The configuration manager evolves the plant transition system along the desired trajectory. The combination of a transition system and a configuration manager is called a *configuration system*. More precisely,

Definition 2 A *configuration system* is a tuple $\langle \mathcal{S}, \Theta, \sigma \rangle$, where $\mathcal{S}$ is a transition system, Θ is a feasible state of $\mathcal{S}$ representing its initial state, and $\sigma : g_0, g_1, \ldots$ is a sequence of state formulae called *goal configurations*. A configuration system generates a *configuration* trajectory $\sigma : s_0, s_1 \ldots$ for $\mathcal{S}$ such that s_0 is Θ and either s_{i+1} satisfies g_i or $s_{i+1} \in \tau(s_i)$ for some failure transition τ.

Configuration management is achieved by sensing and controlling the state of a transition system. The state of a transition system is (partially) observable through a set of variables $\mathcal{O} \subseteq \Pi$. The next state of a transition system can be controlled through an exogenous set of variables $\mu \subseteq \Pi$. We assume that μ are exogenous so that the transitions of the system do not determine the values of variables in μ. We also assume that the values of $\mathcal{O}$ in a given state are independent of the values of μ at that state, though they may depend on the values of μ at the previous state.

Definition 3 A *configuration manager* $\mathcal{C}$ for a transition system $\mathcal{S}$ is an online controller that takes as input an initial state, a sequence of goal configurations, and a sequence of values for sensed variables $\mathcal{O}$, and incrementally generates a sequence of values for control variables μ such that the combination of $\mathcal{C}$ and $\mathcal{S}$ is a configuration system.

A *model-based configuration manager* is a configuration manager that uses a specification of the transition system to compute the desired sequence of control values. We discuss this in detail shortly.

Plant transition system

We model a plant as a transition system composed of a set of concurrent component transition systems that communicate through shared variables. The component transition systems of a plant operate synchronously, that is, at each plant transition every component performs a state transition. The motivation for imposing synchrony is given in the next section. We require the plant's specification to be composed out of its components' specification as follows:

Definition 4 A plant transition system $\mathcal{S} = \langle \Pi, \Sigma, \mathcal{T} \rangle$ composed of a set $\mathcal{CD}$ of component transition systems is a transition system such that;

- The set of state variables of each transition system in $\mathcal{CD}$ is a subset of Π. The plant transition system may introduce additional variables not in any of its component transition systems.
- Each state in Σ, when restricted to the appropriate subset of variables, is feasible for each transition system in $\mathcal{CD}$, i.e.., for each $C \in CD$, $\rho_{\mathcal{S}} \models \rho_C$, though $\rho_{\mathcal{S}}$ can be stronger than the conjunction of the ρ_C.
- Each transition $\tau \in \mathcal{T}$ performs one transition τ_C for each transition system $C \in \mathcal{CD}$. This means that

$$\rho_\tau \Leftrightarrow \bigwedge_{C \in \mathcal{CD}} \rho_{\tau_C}$$

The concept of synchronous, concurrent actions is captured by requiring that each component performs a transition for each state change. Nondeterminism lies in the fact that each component can traverse either its nominal transition or any of its failure transitions. The nominal transition of a plant performs the nominal transition for each of its components. Multiple simultaneous failures correspond to traversing multiple component failure transitions.

Returning to the example, each hardware component in Figure 1 is modeled by a component transition system. Component communication, denoted by wires in the figure, is modeled by shared variables between the corresponding component transition systems.

Model-based configuration management

We now introduce configuration managers that make extensive use of a model to infer a plant's current state and to select optimal control actions to meet configuration goals. This is essential in situations where mistakes may lead to disaster, ruling out simple trial-and-error approaches. A *model-based* configuration manager uses a plant transition model $\mathcal{M}$ to determine the desired control sequence in two stages—*mode identification* (MI) and *mode reconfiguration* (MR). MI incrementally generates the set of all plant trajectories consistent with the plant transition model and the sequence of plant control and sensed values. MR uses a plant transition model and the partial trajectories

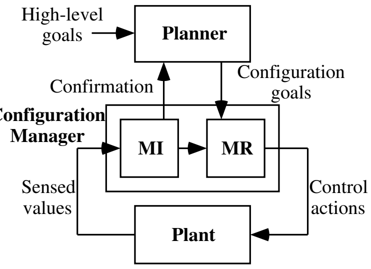

Figure 2: Model-based configuration management

generated by MI up to the current state to determine a set of control values such that all predicted trajectories achieve the configuration goal in the next state.

Both MI and MR are reactive. MI infers the current state from knowledge of the previous state and observations within the current state. MR only considers actions that achieve the configuration goal within the next state. Given these commitments, the decision to model component transitions as synchronous is key. An alternative is to model multiple transitions through interleaving. This, however, places an arbitrary distance between the current state and the state in which the goal is achieved, defeating a desire to limit inference to a small number of states. Hence we use an abstraction in which component transitions occur synchronously, even though the underlying hardware may interleave the transitions. The abstraction is correct if different interleavings produce the same final result.

We now formally characterize MI and MR. Recall that taking a transition τ_i corresponds to taking all subtransitions τ_{ij}. A transition τ_i can be defined to apply over a set of states S in the natural way:

$$\tau_i(S) = \bigcup_{s \in S} \tau_i(s)$$

Similarly we define $\tau_{ij}(S)$ for each subtransition τ_{ij} of τ_i. We can show that

$$\tau_i(S) \subseteq \bigcap_j \tau_{ij}(S) \tag{1}$$

In the following, S_i is the set of possible states at time i before any control values are asserted by MR, μ_i is the control values asserted at time i, $\mathcal{O}_i$ is the observations at time i, and S_{μ_i} and $S_{\mathcal{O}_i}$ is the set of states in which control and sensed variables have values specified in μ_i and $\mathcal{O}_i$, respectively. Hence, $S_i \cap S_{\mu_i}$ is the set of possible states at time i.

We characterize both MI and MR in two ways—first model theoretically and then using state formulas.

Mode Identification

MI incrementally generate the sequence $S_0, S_1, \ldots$ using a model of the transitions and knowledge of the control actions μ_i as follows:

$$S_0 = \{\Theta\} \tag{2}$$

$$S_{i+1} = \left(\bigcup_j \tau_j(S_i \cap S_{\mu_i})\right) \cap \Sigma \cap S_{\mathcal{O}_{i+1}} \tag{3}$$

$$\subseteq \bigcup_j \left(\bigcap_k \tau_{jk}(S_i \cap S_{\mu_i})\right) \cap \Sigma \cap S_{\mathcal{O}_{i+1}} \tag{4}$$

where the final inclusion follows from Equation 1. Equation 4 is useful because it is a characterization of S_{i+1} in terms of the subtransitions τ_{jk}. This allows us to develop the following characterization of S_{i+1} in terms of state formulae:

$$\rho_{S_{i+1}} \equiv \bigvee_{\tau_j} \left(\bigwedge_{\rho_{S_i} \wedge \rho_{S_{\mu_i}} \models \Phi_{jk}} \Psi_{jk} \right) \wedge \rho_\Sigma \wedge \rho_{\mathcal{O}_{i+1}} \tag{5}$$

This is a sound but potentially incomplete characterization of S_{i+1}, i.e., every state in S_{i+1} satisfies $\rho_{S_{i+1}}$ but not all states that satisfy $\rho_{S_{i+1}}$ are necessarily in S_{i+1}. However, generating $\rho_{S_{i+1}}$ requires only that the entailment of the antecedent of each subtransition be checked. On the other hand, generating a complete characterization based on Equation 3 would require enumerating all the states in S_i, which can be computationally expensive if S_i contains many states.

Mode Reconfiguration

MR incrementally generates the next set of control values μ_i using a model of the nominal transition τ_n, the desired goal configuration g_i, and the current set of possible states S_i. The model-theoretic characterization of $\mathcal{M}_i$, the set of possible control actions that MR can take at time i, is as follows:

$$\mathcal{M}_i = \{\mu_j | \tau_n(S_i \cap S_{\mu_j}) \cap \Sigma \subseteq g_i\} \tag{6}$$

$$\supseteq \{\mu_j | \bigcap_k \tau_{nk}(S_i \cap S_{\mu_j}) \cap \Sigma \subseteq g_i\} \tag{7}$$

where, once again, the latter inclusion follows from Equation 1. As with MI, this weaker characterization of $\mathcal{M}_i$ is useful because it is in terms of the subtransitions τ_{nk}. This allows us to develop the following characterization of $\mathcal{M}_i$ in terms of state formulae:

$$\mathcal{M}_i \supseteq \{\mu_j | \ \rho_{S_i} \wedge \rho_{\mu_j} \text{is consistent and} \bigwedge_{\rho_{S_i} \wedge \rho_{\mu_j} \models \Phi_{nk}} \Psi_{nk} \wedge \rho_\Sigma \models \rho_{g_i}\} \tag{8}$$

The first part says that the control actions must be consistent with the current state, since without this condition the goals can be simply achieved by making the world inconsistent. Equation 8 is a sound but potentially incomplete characterization of the set of control actions in $\mathcal{M}_i$, i.e., every control action that satisfies the condition on the right hand side is in $\mathcal{M}_i$, but not necessarily vice versa. However, checking whether a given μ_j is an adequate control action only requires that the entailment of the antecedent of each subtransition be checked. On the other hand, generating a complete characterization based on Equation 6 would require enumerating all the states in S_i, which can be computationally expensive if S_i contains many states.

If $\mathcal{M}_i$ is empty, no actions achieve the required goal. The planner then initiates replanning to dynamically change the sequence of configuration goals.

Statistically optimal configuration management

The previous section characterized the set of all feasible trajectories and control actions. However, in practice, not all such trajectories and control actions need to be generated. Rather, just the likely trajectories and an optimal control action need to be generated. We efficiently generate these by recasting MI and MR as *combinatorial optimization problems*.

A combinatorial optimization problem is a tuple (X, C, f), where X is a finite set of variables with finite domains, C is set of constraints over X, and f is an objective function. A feasible solution is an assignment to each variable in X a value from its domain such that all constraints in C are satisfied. The problem is to find one or more of the leading feasible solutions, i.e., to generate a prefix of the sequence of feasible solutions ordered in decreasing order of f.

Mode Identification

Equation 3 characterizes the trajectory generation problem as identifying the set of all transitions from the previous state that yield current states consistent with the current observations. Recall that a transition system has one nominal transition and a set of failure transitions. In any state, the transition system non-deterministically selects exactly one of these transitions to evolve to the next state. We quantify this non-deterministic choice by associating a probability with each transition: $p(\tau)$ is the probability that the plant selects transition τ.[1]

With this viewpoint, we recast MI's task to be one of identifying the likely trajectories of the plant. In keeping with the reactive nature of configuration management, MI incrementally tracks likely trajectories by extending the current set of trajectories by the likely transitions. The only change required in Equation 5 is that, rather than the disjunct ranging over all transitions τ_j, it ranges over the subset of likely transitions.

The likelihood of a transition is its posterior probability $p(\tau|\mathcal{O}_i)$. This posterior is estimated in the standard way using Bayes Rule:

$$p(\tau|\mathcal{O}_i) = \frac{p(\mathcal{O}_i|\tau)p(\tau)}{p(\mathcal{O}_i)} \propto p(\mathcal{O}_i|\tau)p(\tau)$$

If $\tau(S_{i-1})$ and $\mathcal{O}_i$ are disjoint sets then clearly $p(\mathcal{O}_i|\tau) = 0$. Similarly, if $\tau(S_{i-1}) \subseteq \mathcal{O}_i$ then $\mathcal{O}_i$ is entailed and $p(\mathcal{O}_i|\tau) = 1$, and hence the posterior probability of τ is proportional to the prior. If neither of the above two situations arises then $p(\mathcal{O}_i|\tau) \leq 1$. Estimating this probability is difficult and requires more research, but see (de Kleer & Williams 1987).

Finally, to view MI as a combinatorial optimization problem, recall that each plant transition consists of a single transition for each of its components. We introduce a variable into X for each component in the plant whose values are the possible component transitions. Each plant transition corresponds to an assignment of values to variables in X. C is the constraint that the states resulting from taking a plant transition is consistent with the observed values. The objective function f is the probability of a plant transition. The resulting combinatorial optimization problem hence identifies the leading transitions at each state, allowing MI to track the set of likely trajectories.

Mode reconfiguration

Equation 6 characterizes the reconfiguration problem as one of identifying a control action that ensures that the result of taking the nominal transition yields states in which the configuration goal is satisfied. Recasting MR as a combinatorial optimization problem is straightforward. The variables X are just the control variables μ with identical domains. C is the constraint in Equation 5 that μ_j must satisfy to be in $\mathcal{M}_i$. Finally, as noted earlier, different control actions can have different costs that reflect differing resource requirements. We take f to be negative of the cost of a control action. The resulting combinatorial optimization problem hence identifies the lowest cost control action that achieves the goal configuration in the next state.

Conflict-directed best first search

We solve the above combinatorial optimization problems using a *conflict directed best first search*, similar in spirit to (de Kleer & Williams 1989; Dressler & Struss 1994). A conflict is a partial solution such that any solution containing the conflict is guaranteed to be infeasible. Hence, a single conflict can rule out the feasibility of a large number of solutions, thereby focusing the search. Conflicts are generated while checking to see if a solution X_i satisfies the constraints C.

Our conflict-directed best-first search algorithm, *CBFS*, is shown in in Figure 3. It has two major components: (a) an agenda that holds unprocessed solutions in decreasing order of f; and (b) a procedure to generate the *immediate successors* of a solution. The main loop removes the first solution from the agenda, checks its feasibility, and adds in the solution's immediate successors to the agenda. When a solution X_i is infeasible, we assume that the process of checking the constraints C returns a part of X_i as a conflict N_i. We focus the search by generating only those immediate successors of X_i that are not subsumed by N_i, i.e., do not agree with N_i on all variables.

Intuitively, solution X_j is an immediate successor of solution X_i only if $f(X_i) \geq f(X_j)$ and X_i and X_j differ

[1] We make the simplifying assumption that the probability of a transition is independent of the current state.

function *CBFS*(X, C, f)
  Agenda = {{*best-solution(X)*}}; *Result* = $\emptyset$;
  while *Agenda* is not empty **do**
    Soln = *pop(Agenda)*;
    if *Soln* satisfies C **then**
      Add *Soln* to *Result*;
      if enough solutions have been found **then**
        return *Result*;
      else *Succs* = immediate successors *Soln*;
    else
      Conf = a conflict that subsumes *Soln*;
      Succs = immediate successors of *Soln* not
        subsumed by *Conf*;
    endif
    Insert each solution in *Succs* into *Agenda*
      in decreasing f order;
  endwhile
  return *Result*;
end *CBFS*

Figure 3: Conflict directed best first search algorithm for combinatorial optimization

only in the value assigned to a single variable (ties are broken consistently to prevent loops in the successor graph). One can show this definition of the immediate successors of a solution suffices to prove the correctness of *CBFS*, i.e., to show that all feasible solutions are generated in decreasing order of f.

Our implemented algorithm further refines the notion of an immediate successor. The major benefit of this refinement is that each time a solution is removed from the agenda, at most two new solutions are added on, so that the size of the agenda is bounded by the total number of solutions that have been checked for feasibility, thus preserving reactivity (details are beyond the scope of this paper). For MI, we use full propositional satisfiability to check C (transition consistency). Interestingly, reactivity is preserved since the causal nature of a plant's state constraints means that full satisfiability requires little search. For MR, we preserve reactivity by using unit propagation to check C (entailment of goals), reflecting the fact that entailment is usually harder than satisfiability. Finally, note that $CBFS$ does not require minimal conflicts. Empirically, the first conflict found by the constraint checker provides enough focusing, so that the extra effort to find minimal conflicts is unnecessary.

Implementation and experiments

We have implemented Livingstone based on the ideas described in this paper. Livingstone was part of a rapid prototyping demonstration of an autonomous architecture for spacecraft control, together with the HSTS planning/scheduling engine and the RAPS executive (Pell *et al.* 1996). In this architecture, RAPS further decomposes and orders HSTS output before handing goals to Livingstone. To evaluate the architecture, spacecraft engineers at JPL defined the Newmaap spacecraft and scenario. The Newmaap spacecraft is a scaled down version of the Cassini spacecraft that retains the most challenging aspects of spacecraft control. The Newmaap scenario was based on the most complex mission phase of the Cassini spacecraft—successful insertion into Saturn's orbit even in the event of any single point of failure. Table 1 provides summary information about Livingstone's model of the Newmaap spacecraft, demonstrating its complexity.

Number of components	80
Average modes/component	3.5
Number of propositions	3424
Number of clauses	11101

Table 1: NewMaap spacecraft model properties

Failure Scenario	MI			MR	
	Chck	Accpt	Time	Chck	Time
EGA preaim	7	2	2.2	4	1.7
BPLVD	5	2	2.7	8	2.9
IRU	4	2	1.5	4	1.6
EGA burn	7	2	2.2	11	3.6
ACC	4	2	2.5	5	1.9
ME hot	6	2	2.4	13	3.8
Acc low	16	3	5.5	20	6.1

Table 2: Results from the seven Newmaap failure recovery scenarios

The Newmaap scenario included seven failure scenarios. From Livingstone's viewpoint, each scenario required identifying the failure transitions using MI and deciding on a set of control actions to recover from the failure using MR. Table 2 shows the results of running Livingstone on these scenarios. The first column names each of the scenarios; a discussion of the details of these scenarios is beyond the scope of this paper. The second and fifth columns show the number of solutions checked by algorithm $CBFS$ when applied to MI and MR, respectively. On can see that even though the spacecraft model is large, the use of conflicts dramatically focuses the search. The third column shows the number of leading trajectory extensions identified by MI. The limited sensing available on the Newmaap spacecraft often makes it impossible to identify unique trajectories. This is generally true on spacecraft, since adding sensors increases spacecraft weight. The fourth and sixth columns show the time in seconds on a Sparc 5 spent by MI and MR on each scenario, once again demonstrating the efficiency of our approach.

Livingstone's MI component was also tested on ten combinational circuits from a standard test suite (Brglez & Fujiwara 1985). Each component in these circuits was assumed to be in one of four modes: ok, stuck-at-1, stuck-at-0, unknown. The probability of transitioning to the stuck-at modes was set at 0.099 and to the unknown mode was set at 0.002. We ran 20

Devices	# of components	# of clauses	Checked	Time
c17	6	18	18	0.1
c432	160	514	58	4.7
c499	202	714	43	4.5
c880	383	1112	36	4.0
c1355	546	1610	52	12.3
c1908	880	2378	64	22.8
c2670	1193	3269	93	28.8
c3540	1669	4608	140	113.3
c5315	2307	6693	84	61.2
c7552	3512	9656	71	61.5

Table 3: Testing MI on a standard suite of circuits

experiments on each circuit using a random fault and a random input vector sensitive to this fault. MI stopped generating trajectories after either 10 leading trajectories had been generated, or when the next trajectory was 100 times more unlikely than the most likely trajectory. Table 3 shows the results of our experiments. The columns are self-explanatory, except that the time is the number of seconds on a Sparc 2. Note once again the power of conflict-directed search to dramatically focus search. Interestingly, these results are comparable to the results from the very best ATMS-based implementations, even though Livingstone uses no ATMS. Furthermore, initial experiments with a partial LTMS have demonstrated an order of magnitude speed-up.

Livingstone is also being applied to the autonomous real-time control of a scientific instrument called a Bioreactor. This project is still underway, and final results are forthcoming. More excitingly, the success of the Newmaap demonstration has launched Livingstone to new heights: Livingstone, together with HSTS and RAPS, is going to be part of the flight software of the first New Millennium mission, called Deep Space One, to be launched in 1998. We expect final delivery of Livingstone to this project in 1997.

Conclusions

In this paper we introduced Livingstone, a reactive, model-based self-configuring system, which provides a kernel for model-based autonomy. It represents an important step toward our goal of developing a fully model-based autonomous system (Williams 1996).

Three technical features of Livingstone are particularly worth highlighting. First, our modeling formalism achieves broad coverage of hybrid hardware/software systems by coupling the transition system models underlying concurrent reactive languages (Manna & Pnueli 1992) with the qualitative representations developed in model-based reasoning. Second, we achieve a reactive system that performs significant deduction in the sense/response loop by using propositional transition systems, qualitative models, and synchronous components transitions. The interesting and important result of Newmaap, Deep Space One, and the Bioreactor is that Livingstone's models and restricted inference are still expressive enough to solve important problems in a diverse set of domains. Third, Livingstone casts mode identification and mode reconfiguration as combinatorial optimization problems, and uses a core conflict-directed best-first search to solve them. The ubiquity of combinatorial optimization problems and the power of conflict-directed search are central themes in Livingstone.

Livingstone, the HSTS planning/scheduling system, and the RAPS executive, have been selected to form the core autonomy architecture of Deep Space One, the first flight of NASA's New Millennium program.

Acknowledgements: We would like to thank Nicola Muscettola and Barney Pell for valuable discussions and comments on the paper.

References

Agre, P., and Chapman, D. 1987. Pengi: An implementation of a theory of activity. In *Procs. of AAAI-87*.

Brglez, F., and Fujiwara, H. 1985. A neutral netlist of 10 combinational benchmark circuits. In *Int. Symp. on Circuits and Systems*.

Brooks, R. A. 1991. Intelligence without reason. In *Procs. of IJCAI-91*, 569–595.

de Kleer, J., and Williams, B. C. 1987. Diagnosing multiple faults. *Artificial Intelligence* 32(1):97–130.

de Kleer, J., and Williams, B. C. 1989. Diagnosis with behavioral modes. In *Procs. of IJCAI-89*, 1324–1330.

Dressler, O., and Struss, P. 1994. Model-based diagnosis with the default-based diagnosis engine: Effective control strategies that work in practice. In *Procs. of ECAI-94*.

Firby, R. J. 1995. The RAP language manual. Working Note AAP-6, University of Chicago.

Kuipers, B., and Astrom, K. 1994. The composition and validation of heterogenous control laws. *Automatica* 30(2):233–249.

Manna, Z., and Pnueli, A. 1992. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag.

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

Nerode, A., and Kohn, W. 1993. Models for hybrid systems. In Grossman, R. L. *et al*, eds., *Hybrid Systems.* Springer-Verlag. 317–356.

Pell, B.; Bernard, D. E.; Chien, S. A.; Gat, E.; Muscettola, N.; Nayak, P. P.; Wagner, M. D.; and Williams, B. C. 1996. A remote agent prototype for spacecraft autonomy In *Procs. of SPIE Conf. on Optical Science, Engineering, and Instrumentation.*

Poole, D. 1995. Sensing and acting in the independent choice logic. In *Procs. of the AAAI Spring Symp. on Extending Theories of Action*, 163–168.

Williams, B. C. 1996. Model-based autonomous systems in the new millennium. In *Procs. of AIPS-96.*

Zhang, Y., and Mackworth, A. K. 1995. Constraint nets: A semantic model for hybrid dynamic systems. *Journal of Theoretical Computer Science* 138(1):211–239.