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 self-reconfiguring autonomous system, that is reactive and uses component-based declarative models. The paper presents a formal characterization of the representation formalism used in Livingstone, and reports on our experience with the implementation in a variety of domains. Livingstone's representation formalism achieves broad coverage of hybrid software/hardware systems by coupling the concurrent transition system models underlying concurrent reactive languages with the discrete qualitative representations developed in model-based reasoning. We achieve a reactive system that performs significant deductions 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, conflict-based feedback controller that generates focussed, optimal responses. Livingstone automates all these tasks using a single model and a single core deductive engine, 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 the core autonomy architecture for Deep Space One, the first spacecraft for NASA's New Millenium program.

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 late 1996 and the probe (Deep Space One) to be launched in early 1998. The final pressure, low cost, is of an equal magnitude. Together this poses an extraordinary opportunity and challenge 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 the control software that coordinates interactions between subsystems. Finally, the space of failure scenarios a spacecraft will need to entertain over its lifespan will be far too large to generate software before flight that explicitly enumerates all contingencies. Hence the spacecraft will need to think through the consequences of reconfiguration options on the fly while ensuring reactivity.

We made substantial progress on each of these fronts through a system called *Livingstone*, an implemented kernel for a self-reconfiguring autonomous system, that is reactive and uses component-based declarative models. This paper presents a formal characterization of a reactive, model-based configuration management system underlying Livingstone. Several contributions are key: First, our modeling formalism represents a radical shift from first order logic, traditionally used to characterize model-based diagnostic systems. Our representation formalism achieves broad coverage of hybrid software/hardware systems by coupling the concurrent transition system models underlying concurrent reactive languages (Manna & Pnueli 1992) with the discrete qualitative representations developed in model-based reasoning. Reactivity is respected by restricting the model to concurrent propositional transition systems that are synchronous. Second, this approach helps to unify the classical dichotomy within AI between deduction and reactivity. We achieve a reactive system that performs significant deductions 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, conflict-based feedback controller that generates focussed, optimal responses. Third, the long held vision of the model-based reasoning community 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 tasks including: monitoring, tracking planner goal activations, confirming hardware modes, reconfiguring hardware, detecting anomalies, isolating faults, diagnosis,

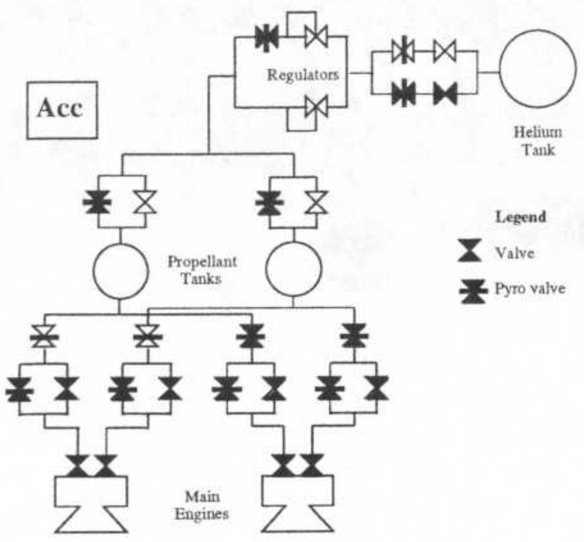

Figure 1: Engine schematic

fault recovery, safing and fault avoidance. Livingstone automates all these tasks using a single model and a single core deductive engine, thus making significant progress towards achieving the model-based vision.

Livingstone, tightly integrated with the HSTS planning/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 half a dozen or more failures, including unanticipated bugs in the simulator. Consequently, Livingstone, together with RAPS and HSTS have been selected as the core autonomy architecture of the New Millenium program, and will fly Deep Space One early 1998.

The rest of the paper is organized as follows. The next section introduces part of the spacecraft domain and the problem of configuration management in this domain. Section introduces transition systems, the key formalism for modeling hybrid concurrent systems. It also introduces the basic formalization of configuration management. Section discusses model-based configuration management, and discusses its key components: mode identification and mode reconfiguration. Section introduces algorithms for statistically optimal model-based configuration management using conflict-directed best-first search. Section presents an empirical evaluation of Livingstone using a suite of scenarios and domains. Conclusions and related work are discussed in Section .

Example: Autonomous Space Exploration

Figure 1 shows the schematic of the main engine subsystem of Cassini, the most complex spacecraft built to date. The main engine subsystem 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 tank pressure to a lower working pressure. When propellant paths to a main engine are open, the pressurization of the propellant 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 (either from open to closed or vice versa) exactly once. Their function is to isolate parts of the main engine subsystem until needed, or to isolate failed parts. The latch valve states are controlled using valve drivers (not shown), and the accelerometer (Acc) senses the thrust generated by the main engines. In the figure, valves in solid black are closed, while the others are open.

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 different 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. There are a number of other configurations corresponding 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 changing the state of 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 now 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 for the spacecraft constantly attempts to move the spacecraft into lowest cost configurations that achieve the desired high-level 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.

Configuration goals are generated dyamically through onboard planning, scheduling and execution capabilities. High-level goals are translated to partially-ordered resource timelines by the HSTS planner/scheduler. RAPS then executes these plans, dynamically decomposing them into sequences of configuration goals that are passed to Livingstone. RAPS complements Livingstone, for example, decomposing planner tokens, monitoring plan temporal constraints and coordinating replanning when configuration goals or temporal constraints cannot be satisfied.

Models of Concurrent Processes

Selecting a restricted, but appropriately expressive formalism for describing the plant is essential to achieving the competing goals of achieving reactivity on the one hand and richly expressing the properties of hybrid software/hardware systems. Extensive experience applying model-based diagnosis to causal systems suggests that propositional deductive engines with a focusing algorithm can be made extremely fast. We know of no first order formalism that achieves these properties, thus operating over fixed, finite domains is essential.

Reasoning about a component's configurations and autonomous repair requires the concepts of operating modes, failure modes, unmodeled failures, operating modes, repairable failures and configuration changes. These concepts can be expressed in a state diagram. Note in particular that repairable failures are represented by state 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.

Components operate simultaneously, communicating over wires. Hence we model components through concurrent, communicating transitions systems. Likewise, for software routines, it is well established that a broad class of reactive languages can be represented naturally as concurrent transition systems communicating through shared variables. We use the concurrent transition system model of Manna & Pnueli and its specification through temporal logic as a starting point (Manna & Pnueli 1992) for modeling both software and hardware.

Where our model differs from that of Manna & Pnueli, is that reactive software modifies its state through explicit variable assignments. On the other hand, a hardware component's behavior in a given state is governed by a set of constraints between variables. For digital systems these constraints are over finite domains, while for continuous components these domains are infinite. A variety of experiences applying qualitative modeling to diagnostic tasks for digital systems (Hamscher 1991), copiers and spacecraft propulsion, suggest that extremely simple qualitative representations over finite domains are quite adequate for tackling many hardware diagnostic problems. The added advantage of using qualitative models is that the models nominal behavior are extremely robust to detailed modeling errors and changes, avoiding false diagnoses. Hence behaviors within states are represented by constraints over finite domains, which in turn are encoded as propositional formula.

Other authors such as Zhang and Mackworth, McIlraith, Levesque and Reiter, and Poole have been considering related issues in reactive autonomous systems. The major difference between their work and ours is our focus on fast reactive inference using propositional encodings over finite domain.

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. More precisely,

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 $\mathcal{S}$ is designated the *nominal* transition, with all other transitions being *failure* transitions. Hence in any state a component may nondeterministically choose to perform either its nominal transition, corresponding to correct functioning, or a failure transition, which moves to a set of failure states. 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 $\mathcal{S} = \langle \Pi, \Sigma, \mathcal{T} \rangle$ can be naturally 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 $\rho_{\mathcal{S}}$.

Similarly, 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 and Ψ 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 formulae ρ_{τ_i} is called a *subtransition*. Hence taking a transition τ corresponds to taking all its subtransitions τ_i.

Note that specifying transition systems only draws upon the $\bigcirc$ operator, above and beyond standard propositional logic. This severely constrained use of temporal logic is an essential property of transition systems that will allow 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

$$\begin{aligned} mode = on \quad \Rightarrow \quad & (cmdin = open \Rightarrow cmdout = open) \\ & \wedge(cmdin = close \Rightarrow cmdout = close) \\ & \wedge\neg(cmdin = open \vee cmdin = close) \\ & \Rightarrow cmdout = none \\ mode = off \quad \Rightarrow \quad & cmdout = none \end{aligned}$$

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

$$\begin{aligned} & ((mode = on) \vee (mode = off)) \wedge cmdin = off \Rightarrow \\ & \quad \bigcirc mode = off \\ & ((mode = on) \vee (mode = off)) \wedge cmdin = on \Rightarrow \\ & \quad \bigcirc mode = on \\ & \neg(mode = failed) \wedge cmdin = reset \Rightarrow \bigcirc mode = on \\ & mode = reset \wedge \neg(cmdin = reset) \Rightarrow \bigcirc mode = reset \\ & mode = failed \Rightarrow \bigcirc mode = failed \end{aligned}$$

The driver also has a failure transition specified by the formula $\bigcirc mode = failed$.

Configuration management

Given a transition system and an initial state, *configuration management* involves evolving the transition system along a 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 that the specification of the plant's transition system be composed out of the specification for its component transition systems 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 a feasible state for each transition system in $\mathcal{CD}$. This means that for each $C \in CD$, $\rho_{\mathcal{S}} \models \rho_C$. However, $\rho_{\mathcal{S}}$ 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 the nominal transition or any of the failure transitions at

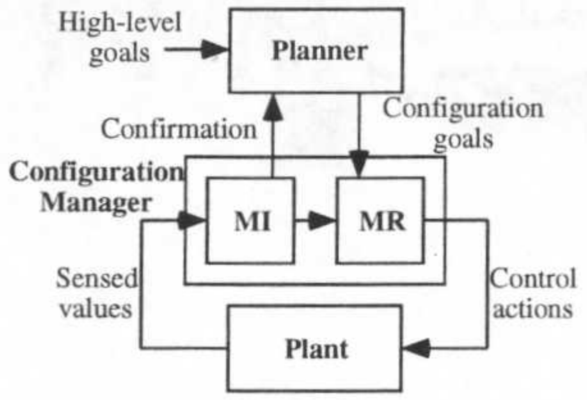

Figure 2: Model-based configuration management

each state change. Note that the nominal transition of a plant performs the nominal transition for each of its components, and that multiple simultaneous failures correspond to traversing multiple component failure transitions.

The set of trajectories for a plant transition system are specified in temporal logic by the formula ρ_{st}:

$$\begin{aligned} \rho_{st} \equiv\ & \rho_{\Theta} \wedge \Box \left(\rho_{\Sigma} \wedge \bigvee_i \left(\bigwedge_j \left(\Phi_{ij} \Rightarrow \bigcirc \Psi_{ij}\right)\right)\right) \\ & \bigwedge_i \bigcirc_i \left(\rho_{\text{obs}_i} \wedge \rho_{\mu_i}\right) \end{aligned}$$

where $\bigcirc_i \Phi$ is used to specify that Φ holds in the ith state, and is a syntactic short form defined by $\bigcirc_i \Phi \equiv \bigcirc \bigcirc_{i-1} \Phi$ and $\bigcirc_0 \Phi \equiv \Phi$.

Returning to the example, each hardware component in the schematic (figure 1) is modeled by a component transition system similar to that given in Section . Component communication, denoted by wires in the schematic, is modeled by shared variables between the corresponding component transition systems.

Model-based configuration management

We presume that the state of the plant is partially observable, and that the results of actions on the plant influence the observables and internal state variables through a set of physical processes. Our focus is on reactive configuration management systems that use a model to infer a plant's current state and to select optimal control actions to meet configuration goals.

More specifically, 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 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 concurrent transitions through interleaving. This, however, would place 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 fixed number of states. Hence we use an abstraction in which multiple commands are given synchronously. In the Newmaap spacecraft demonstration, for example, these synchronous commands were sent by a single processor through interleaving, and the next state sensor information returned to Livingstone is the state following the execution of all these commands.

We now develop a formal characterization of MI and MR. Recall that taking a transition τ_i corresponds to taking all of a set of 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

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

In the following characterizations, S_i denotes the set of possible states at time i before any control values are asserted by MR, μ_i denotes the control values asserted at time i, $\mathcal{O}_i$ denotes the observations at time i, and S_{μ_i} and $S_{\mathcal{O}_i}$ denote 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:

$$\begin{aligned} S_0 &= \{\Theta\} && (2) \\ S_{i+1} &= \left(\bigcup_j \tau_j(S_i \cap S_{\mu_i})\right) \cap \Sigma \cap S_{\mathcal{O}_{i+1}} && (3) \\ &\subseteq \bigcup_j \left(\bigcap_k \tau_{jk}(S_i \cap S_{\mu_i})\right) \cap \Sigma \cap S_{\mathcal{O}_{i+1}} && (4) \end{aligned}$$

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:

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

This is a sound but potentially incomplete characterization of the set of states in 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 a lot of 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:

$$\begin{aligned} \mathcal{M}_i &= \{\mu_j | \tau_n(S_i \cap S_{\mu_j}) \cap \Sigma \subseteq g_i\} \quad &(6) \\ &\supseteq \{\mu_j | \bigcap_k \tau_{nk}(S_i \cap S_{\mu_j}) \cap \Sigma \subseteq g_i\} \quad &(7) \end{aligned}$$

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:

$$\begin{aligned} \mathcal{M}_i \supseteq \{\mu_j | \quad & \rho_{S_i} \wedge \rho_{\mu_j} \text{is consistent and} \\ & \bigwedge_{\rho_{S_i} \wedge \rho_{\mu_j} \models \Phi_{nk}} \Psi_{nk} \wedge \rho_\Sigma \models \rho_{g_i}\} \quad (8) \end{aligned}$$

The first part of the characterization 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 a lot of states.

Statistically optimal configuration management

The previous section characterized the set of all feasible trajectories and control actions that can be generated by MI and MR. However, in practice, not all such trajectories and control actions need to be generated. Rather, just the likely trajectories and an optimal control action needs 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 transition system will select transition τ to evolve to the next state.[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 will incrementally track the likely trajectories by always 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 $p(\mathcal{O}_i|\tau) < 1$. Estimating this probability is difficult and requires more

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

research, but see (de Kleer & Williams 1987) for some ideas.

Finally, to view MI as a combinatorial optimization problem, recall that each plant transition consists of a single transition for each of its component transition systems. Hence, 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 set of states resulting from taking a plant transition is consistent with the observed values. The objective function f is just 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 discussed in Section , different control actions can have different costs that reflect differing resource requirements, etc. 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 best-first search. Conflicts are usually generated while checking to see whether a solution X_i satisfies the constraints C. In the case of MI and MR, we check C using unit propagation for propositional inference, so that simple LTMS-style dependency recording suffices to generate conflicts (Forbus & de Kleer 1993).

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 of the algorithm removes the first solution from the agenda, checks whether it is feasible, 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

```
function CBFS(X, C, f)
  Agenda = {{best-candidate(X)}}; Result = ∅;
  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 Cand;
      else
        Conf = a conflict that subsumes Cand;
        Succs = immediate successors of Cand 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

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 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 suffice to prove the correctness of *CBFS*, i.e., to show that even with conflict-directed focusing, all feasible solutions are generated in decreasing order of f. Our implemented algorithm further refines the notion of an immediate successor while preserving correctness. 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 always bounded by the total number of solutions that have been checked for feasibility. The details of this refinement are beyond the scope of this paper and appear in the longer version.

Implementation and experiments

We have implemented Livingstone, a model-based configuration manager, based on the ideas described in this paper. Livingstone was part of a rapid prototyping demonstration of an autonomous architecture for spacecraft control. 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 most challenging aspects of spacecraft control. The

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

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.

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 (initial nominal hardware configurations were established by RAPS). 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 is undesirable because it 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. Furthermore, initial experiments with the use of ideas from truth maintenance has demonstrated an order of magnitude speed-up.

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, and unknown. The probability of transitioning to the stuck-at modes was set at 0.099 and transitioning to the unknown mode was set to 0.002. We ran 20 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 rather than on a Sparc 5. Note once again the power of conflict-directed search to dramatically focus search. Interestingly, it is worth noting that these results are comparable to the results from the very best ATMS-based implementations, even though Livingstone uses no ATMS.

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 combinational circuits

Livingstone is also being applied to two other problems. The first is ASSAP, another rapid prototyping effort demonstrating other aspects of spacecraft autonomy including autonomous science and navigation. The second is the autonomous real-time control of a scientific instrument called a Bioreactor. Both these projects are still underway, and final results are forthcoming. More excitingly, the success of the Newmaap demonstration has launched Livingstone to new heights: Livingstone is going to be part of the flight software of the first New Millennium mission, called Deep Space One, to be launched in early 1998. We expect final delivery of Livingstone to this project late in 1996.

Conclusions

In this paper we introduced Livingstone, a fast, reactive, model-based self-configuring system, which provides a kernel for model-based autonomy. 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.

Three technical features of Livingstone are particularly worth highlighting. First, Livingstone uses concurrent transition systems as its underlying modeling paradigm. Transition systems provide the appropriate modeling paradigm because autonomous systems are concurrent hardware/software hybrids that fundamentally need to reason about their state and how the state evolves over time. Furthermore, transition systems provide a natural characterization of configuration management as a kind of discrete control system.

Second, Livingstone is a fast reactive configuration manager. Reactivity is achieved by restricting reasoning to just the current state and the next state. Fast inference is achieved by focusing on propositional transition systems, unit propagation, and qualitative modeling. The interesting and important result of applying Livingstone to Newmaap, Deep Space One, ASSAP, and the Bioreactor project 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.

References

Brglez, F., and Fujiwara, H. 1985. A neutral netlist of 10 combinational benchmark circuits and a target translator in FORTRAN. Distributed on a tape to participants of the Special Session on ATPG and Fault Simulation, Int. Symposium on Circuits and Systems.

de Kleer, J., and Williams, B. C. 1987. Diagnosing multiple faults. *Artificial Intelligence* 32(1):97–130. Reprinted in (Hamscher, Console, & de Kleer 1992).

de Kleer, J., and Williams, B. C. 1989. Diagnosis with behavioral modes. In *Proceedings of IJCAI-89*, 1324–1330. Reprinted in (Hamscher, Console, & de Kleer 1992).

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

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

Forbus, K. D., and de Kleer, J. 1993. *Building Problem Solvers*. MIT Press.

Hamscher, W.; Console, L.; and de Kleer, J. 1992. *Readings in Model-Based Diagnosis*. San Mateo, CA: Morgan Kaufmann.

Hamscher, W. C. 1991. Modeling digital circuits for troubleshooting. *Artificial Intelligence* 51:223–271.

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.