Automating Composite System Design: a Proposal

Stephen Fickas

CIS-TR-92-02
January 10, 1992

Department of Computer and Information Science
University of Oregon
Eugene, OR 97403

1. Introduction

Our group's historical interest has been in knowledge-based *software* development (AI&SE). However, in the course of our research, we found that we could not formally explain or reproduce the features of standard[1] software designs by focusing solely on the software and its immediate interface to human and hardware systems. Further evidence of this dilemma was found in human systems analysts we studied [Fickas&Nagarajan, 1988]; they focused on policies and concerns which cut across human, hardware and software components. This has led us to an interest in the design of *composite systems*, ones that encompass multiple agents involved in ongoing, interactive activities [Feather, 1987b]. In composite systems, software agents are treated the same as human and physical agents, as components to be integrated together to solve larger system constraints.

We have developed a design model for producing composite systems. The model is founded on an earlier design tool called Glitter [Fickas, 1985], which used automated problem-solving techniques to assist implementation of specifications. While Glitter was promising in several respects, it was overly complex in its implementation, and crude in its interaction with a human designer. Our new model, called Critter, uses a well understood problem solving paradigm, that of state-based search. We have tested Critter by attempting to *rationally reconstruct* or reproduce existing, well-documented composite system designs, some containing no software components (e.g., pre-computer transportation systems), some containing a mixture of software, human, and physical components (e.g., the canonical elevator system [IWSSD4, 1987]), and some containing software components solely (e.g., email transport systems) [Fickas&Helm, 1991].

Applying Critter to rational reconstruction problems has brought encouraging results to date; we will discuss these results in this proposal by following two rational reconstructions we have performed. At the same time our results, like many in the formal methods community, may be criticized as (a) reduced in scale, and (b) difficult to evaluate systematically. We propose a program of research which we believe will:

1. Begin to scale Critter up to work on problems of realistic complexity
2. Generate results that can be evaluated by other researchers in our field and beyond

Our proposal is to apply Critter in the domain of networked applications. Starting from a formal statement of goals, we will use Critter to semi-automatically design a working distributed application that will support group activities on the Internet. We believe this is an important and feasible step toward our ultimate goal of automating the design of composite systems composed of a full range of software, hardware, and human components. We also argue that implementation is a missing but necessary research step in our field: to produce evaluatable results we will implement our designs and make them available for validation outside of our lab.

In the remainder of this proposal, we present the Critter model, and our results to date. We explain our new research problem and its theoretical and methodological advantages. We then close with a discussion of the work most closely related to ours. We only summarize our formal representations in the body of the proposal; the actual details are placed in appendices A and B.

[1] Here *standard* refers to the canonical set of specification and design problems established at IWSSD 87. These include the library problem, the elevator problem, and the patient monitoring problem.

2. Our place in the lifecycle

Before describing Critter in more detail, it may help to set some context. Figure 1 places Critter in the more general system lifecycle we envision.

In the *requirements acquisition* process, analysts acquire informal requirements from users and other concerned parties, in the form of diagrams, text documents, interview transcripts, and so on. The output of this process typically remains informal, although some research tools produce a formal representation.

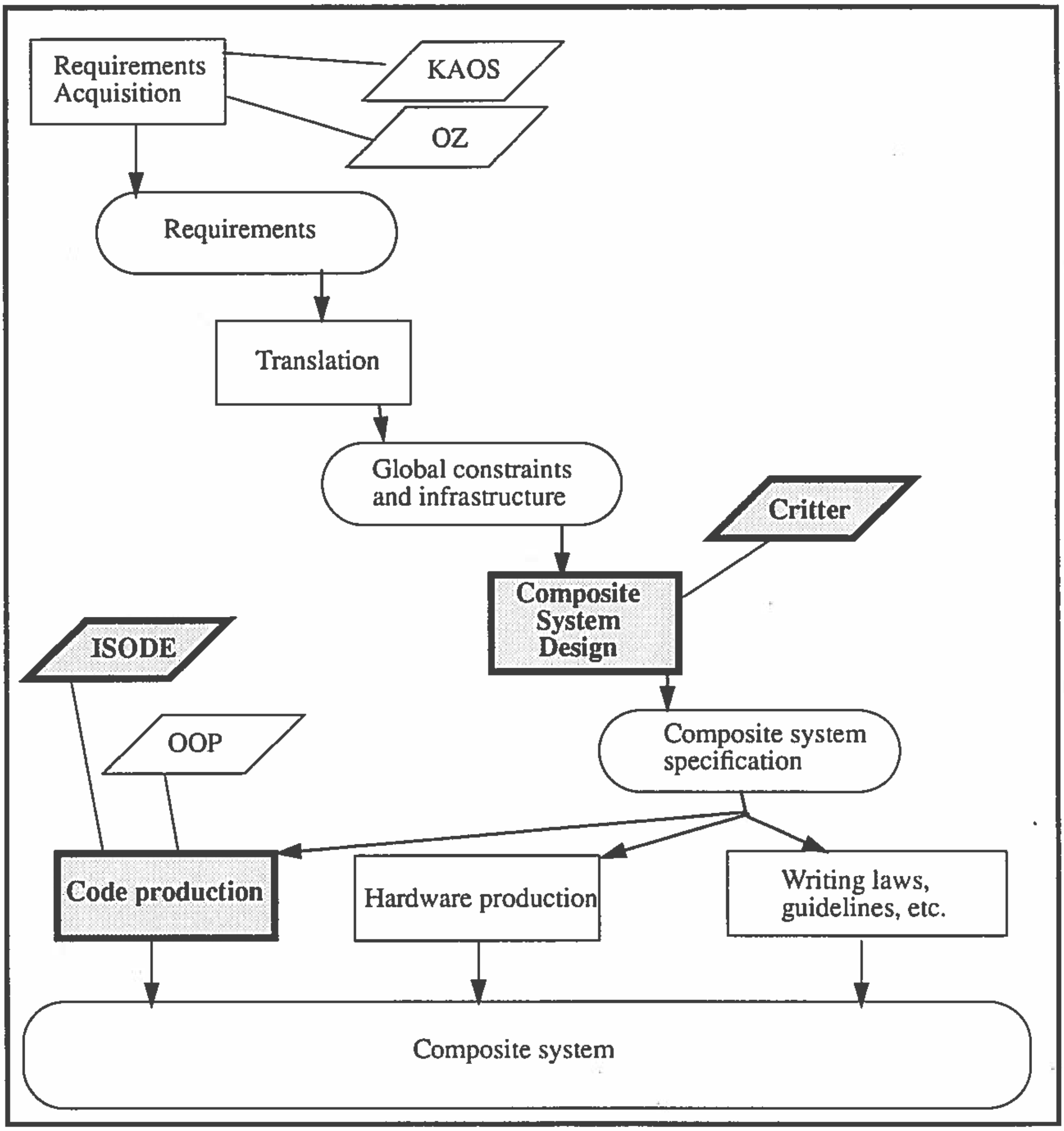
We have included a translation process in Figure 1 to map requirements to a form acceptable by Critter, i.e., a formal statement of global constraints and infrastructure. Clearly this translation process may be a major effort given the lack of consistency in the requirements engineering field, where formality, representations and even definitions may vary from company to company. Nevertheless, we have explored the use of two requirements acquisition tools that minimize the follow up translation process. The first is a tool that attempts to integrate the needs of a diverse set of "stakeholders" in a system. The tool, called Oz [Robinson, 1990], represents the differing values each stakeholder places on system functionality, safety, cost, etc., and allows each stakeholder to state individual, ordered preferences. When conflict occurs among stakeholders' needs, Oz allows an analyst to explore compromises among values and preferences. The output is a formal description of the integrated whole, and is, for the most part, Critter-ready.

The second tool we have used is one for transforming informal statements of need into formal constraints and actions. The system, called KAOS [Dardenne et al., 1991], uses the notion that composite system problems have some standard structure that can be exploited when acquiring requirements. The output of KAOS is easily hand-translated to that needed by Critter. We will return to the KAOS-Critter connection later in the proposal.

The *composite system design* process then decomposes the global constraints and infrastructure into a specification of a *composite system*, a set of components or *agents* which interact to satisfy the global constraints. Finally, the bottom three processes *implement* each agent according to the composite system specification. Typically, implementing an agent involves producing program code which corresponds to that agent's specification, but it might also involve producing hardware, or writing legal statutes or training manuals which prescribe the actions of human workers taking the role of the agent.

Note that there are clearly feedback loops missing in Figure 1. For instance, compromise does not end at the initial requirements level - we will see a need for it in Critter when we begin to question tradeoffs between constraints and design effort. While we have left the feedback links between design level trade-offs and requirements level trade-offs implicit in Figure 1, we are certainly aware of their intertwining.

Figure 1 Critter and its associated development process.



This proposal focuses on the shaded processes of the diagram, and the shaded tools we have used to support them. Critter is described in the next section. Code production tools (e.g., ISODE) for the network applications we propose studying are discussed in section 5[2].

[2] Object oriented programming (OOP) appears to be another interesting match between Critter and code production. For instance, [Wirfs-Brock et al., 1991] discusses an OOP style that incorporates at least informal notions of responsibility. While we will not discuss the Critter-OOP connection further here, we follow it with interest.

3. Results to date

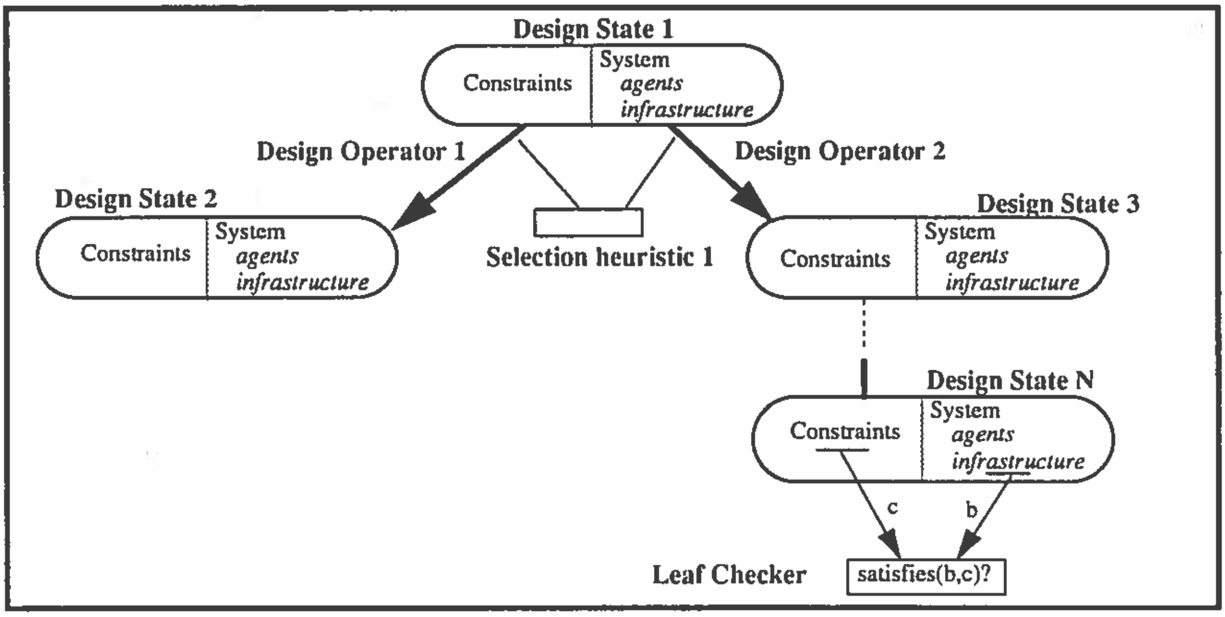
In this section we outline 1) the design model we have developed, and 2) the model's application to two composite system problems. The full, detailed description of the model is found in appendix A; a detailed description of its application is found in appendix B.

3.1 The critter model

The Critter design model (Figure 2) helps a human designer develop a composite system design for some problem using the well-known paradigm of state-based search. Starting from an initial problem state, Critter helps an analyst apply design operators until she arrives at a composite system design state that solves the problem. The Critter model is the sum of the following components:

- a design state representation
- a solution state or leaf-node checker
- a set of move or design operators
- a set of heuristics for selecting design operators
- a search management component that manages the design space, allows browsing, etc.

Figure 2 Critter state-based search model of composite system design



3.1.1 States.

Each design state in our search space represents a single composite system design. A state in Critter has two components: a set of *constraints* on the design, and a *system* that generates behaviors of the design. Constraints are expressed declaratively in terms of system-wide properties, regardless of how the responsibility will eventually be divided among agents in the composite system. They are formal, logical encodings of statements such as "Trains get to their destinations", or

"Trains don't crash" expressed in a temporal logic [Dubois, 1990]. We currently view constraints as having equal utility or value (c.f. Oz in section 2).

The system portion of the state is further divided into two parts. First, there is what we call the infrastructure. This represents the non-agent portion of a system. Examples of an infrastructure are the rail-lines of a railroad system and the communication-lines of a network system. The second part is the set of agents that use the infrastructure of the system and interactions with each other to produce behaviors. In general, an agent is a component of a composite system which can sense a portion of the system's state, make decisions, and perform or prevent actions in parts of the system which the agent controls. An agent in our representation has the following attributes:

1. A set (possibly empty) of the system state it can directly observe.
2. A set (possibly empty) of actions it can perform that affect the system state.
3. A set (possibly empty) of local state information.
4. A set (possibly empty) of responsibilities for achieving one or more constraints or goals of the system[3].

The system portion is represented in a language based on Numerical Petri Nets (NPN) [Wilbur-Ham, 1985] extended to include agents.

3.1.2 Leaf checker.

The overall goal of the Critter search is to bring the system and constraint portions into "consistency" in a single design state (called a "leaf" or "solution" state). A state is consistent when behaviors produced by the system satisfy the constraints. As this implies, the system and constraint components may be misaligned or inconsistent in some states, i.e., in some states of the search, the system may generate behavior that breaks the constraints.

Our model semi-automatically checks design states for consistency using three tools:

1. A planner or scenario generator called OPIE [Anderson and Fickas, 1989]. In general, OPIE attempts to show that a state is *not* consistent by showing that safety or liveness constraints are *not* met. OPIE does this by finding a plan which violates the constraints, in effect producing a counterexample for the constraint. For example, it might prove that the constraint "No two trains are ever in the same block of track" is violated by generating a plan for putting two trains in the same block.
2. A reachability-graph (RG) tool. The tool first produces a reachability graph from a static analysis of the system's NPN, and then allows queries about reachable states. For example, the tool can reply to queries such as, "Is it possible for a train to fail to reach its destination"?. It uses omega values [Huber et al., 1986] to represent infinite plans/behaviors. As with OPIE, these queries can be used to provide existence proofs.

[3] An agent may be *overloaded* in our representation: it may have more than one responsibility in any single composite system, and it may have separate responsibilities in two or more different composite systems. An agent may also *share* responsibility for a single constraint with one or more agents.

3. An NPN simulator. To allow us to test a full blown NPN, we have implemented a tool which runs an NPN on selected test cases. For example, we might run the NPN simulator on a case which attempts to put two trains into the same block of track, in order to validate that this cannot occur. At non-deterministic choice points during the run, alternatives can be either presented to the human designer for selection, or controlled by simple rules in the test case, e.g., always choose transition A over transition B. Typically, the test cases will be chosen from a library designed to expose common design bugs in a specific domain. For instance, a test case for the domain of train scheduling might be one where a message to a train conductor is "dropped" by a busy station operator [Fickas et al., 1991]. The details of this tool, and a semi-automated means of selecting appropriate test cases to feed it, are discussed in [Fickas and Nagarajan, 1988].

For pragmatic reasons, we see a continuing need for all three of these tools. OPIE is not able to operate on the full NPN language, and can be expensive to run. It can only generate finite counterexamples, so it may not be able to produce counterexamples to liveness goals such as "every train gets to its destination". The RG tool, in contrast can represent infinite paths/behaviors. However, it operates on a smaller subset of the NPN language than OPIE, and runs even more slowly. Finally, the NPN simulator, when provided with a suitable set of test cases, can produce counterexamples on a full NPN which neither OPIE or the RG could generate, whether due to resource or to language constraints. However, it does require that the domain be one where test cases can be found and matched, and requires the most human supervision of the three tools. For the problems we have studied to date, two of which we present in this paper, this human supervision has stayed in tolerable bounds for the human designers involved. However, see the discussion of scale issues in section 3.4.

3.1.3 Design operators.

Critter's "search operators" (design operators) are what transform one composite design state into another. They are applied to a non-leaf node N to remedy a deficiency found in N. A design operator may apply to either component of a design state, i.e., the constraints or the system. Further, it can apply to either system component, modifying the infrastructure or the agent set. Because our interest is in composite system problems, our design operators are tailored to multi-agent, distributed-action concerns: creating agents, splitting responsibility for a constraint among agents, establishing communication protocols among agents. Examples of each of these will be seen in our example designs.

3.1.4 Selection Heuristics.

A relatively small number of design operators have accounted for the composite system designs that we have studied. However, these operators are capable of producing a large set of alternative designs, enough so that blind search becomes intractable. We also note that our leaf-checkers are limited in that they cannot evaluate how good or bad a design state is with respect to "nonfunctional" constraints such as cost [Chung et al., 1991]. Consequently, Critter's search process may generate many design states which appear to be "solutions", but which are unimplementable.

We see two means of combatting this. First, we can produce all alternatives of a node N, but attempt to prune bad designs early by heuristic evaluation. At each state we can generate the possible children of that state (leaf or non-leaf), and then attempt to predict which child is more likely to lead to a good design by applying informal or formal domain-specific knowledge. Alternatively, we can attempt to be more selective in the operators we retrieve. The approach is to use a counterexample plan generated by the leaf-checker, as a guide to which design operators could remedy the design's deficiencies. Currently, however, selection heuristics are supplied by the user.

Critter is partially automated. The tools that make up the leaf-node checker are automated, although they require some set up as explained above. The design space is represented in an extended form of IBIS [Conklin and Begeman, 1988] that provides for separate design states. Following IBIS, a design state is represented as a set of issues and subissues (the constraints), and an artifact (an extended form of Petri net that includes agents). Following the IBIS representation, design moves/operators are represented by positions attached to an issue, and design heuristics as arguments attached to a position. In the current implementation, the user manually "applies" design operators by attaching the appropriate positions and updating the agents and infrastructure them using an Petri net editor. The user also must attach any selection heuristics as arguments to the positions. Critter then automatically generates a new design state (a new IBIS state) after the operator has been applied. While there are interesting issues on the use of an IBIS style framework for implementing design tools [Fischer et al., 1989], they are on the periphery of our main points, and hence, we will stay at the level specified in Figure 2 for further discussion.

Instead of discussing the model further here, we will first present two examples, and then reintroduce the topic under the more general heading of scale in section 3.4.

3.2 Applying the Critter model: two examples

We have tested the Critter model described above by a methodology of *rational reconstruction*. In general, this means using our design model on a study problem for which there are already well-documented designs, preferably designs which have evolved over time and which have well-known strengths and weaknesses. We believe the rational reconstruction methodology provides a useful check on the sufficiency of our model, i. e. can we generate interesting, existing designs? Moreover, applying the methodology to well-understood engineering problems helps identify the knowledge our tool would need to perform heuristic evaluation of designs [Feather et al., 1991]. We have used the Critter model to rationally reconstruct several designs; in this section, we summarize two examples, one of which is given in greater detail in appendix B.

A word of motivation on our choice of problems is in order. The first example we present in this paper is a 19th century train management system described in McGean [McGean 1976], henceforth simply the "McGean design"[4]. We arrived at this problem after work on the canonical elevator problem [IWSSD4, 1987] [Feather, 1987a], which in turn derives from [Kemmerer, 1985]. Our switch from the elevator problem was motivated by the following:

- We found it difficult to evaluate the elevator designs that we produced. While there are elevator guidelines, books, and journals published, we failed to find enough detailed design rationale to evaluate our results. We did, however, find intricate and detailed evaluation criteria for train systems [McGean, 1976], including a history of major train accidents and their causes [Shaw, 1961]. We were able to gather enough information to use rational reconstruction as a theory/tool testing method. Both examples in this paper use this approach.

[4] McGean is the chronicler, not the originator of the 1880 design, referred to as a "manual absolute block clearing system." We will call this the McGean design in the interest of brevity.

- Elevator problems are more limited in scale than train problems. In particular, one of our long range interests is an application of composite system design to the Air Traffic Control (ATC) problem. At least some of the current ATC design shares similar features (including misfeatures) with train management systems [Nance, 1986].
- A train system is a more interesting composite system problem. There are more agents involved and more ways to split responsibility among them.

The second problem we have chosen is a simple networking problem, that of email transport. In particular, our target is the reconstruction of the OSI MOTIS (A.K.A. MHS) P1 protocol design. This example has two useful properties:

1. Its design can be validated at a production level. That is, we can implement a MOTIS email transport design leaf node produced by our model as a network application and run it under production conditions (albeit, OSI production conditions). This is a feature of only one of the four canonical IWSSD problems, namely the text editing problem. It is certainly not a feature of the elevator or train problems, at least for our group.
2. MOTIS, and network applications of its ilk, have well studied and formal evaluation criteria [Tanenbaum, 1988]. Many network applications also have a written chronology of their design (see for instance the RFCs maintained by the Internet NIC).

While both examples have positive points, they have negative sides as well. For instance, the train example we have chosen to reconstruct predates computers. Hence, some might view it as an exercise in 19th century transportation engineering. However, we believe this misses the point. Interesting composite systems have human and mechanical agents, as well as computer-based agents. The train example focuses on composite systems constructed from the former. Our MOTIS example focuses on composite systems constructed solely from the latter. Our long range goal is to tackle a problem (e.g., Air Traffic Control, Open Office Systems) that integrates them all. Our research proposal is a further step along this path - it nominates a problem, a distributed group meeting scheduler, for further study that has both software and human agents. The McGean and MOTIS examples lay the foundation for this proposal.

Figure 3 shows the McGean design that is the *target* of our design process. Looking at the final solution from a composite system view, the design designates engineer agents as responsible for entering a block of track between stations only if the signal for that block reads clear (vertical). Station operator agents are given the responsibility of 1) setting the distant/home signal to occupied (horizontal), 2) resetting the signal to clear, and 3) passing information among themselves. Finally, dispatcher agents (not shown in Figure 3) are responsible for making sure trains enter a system in a safe fashion.

Figure 3 Simplified McGean design.

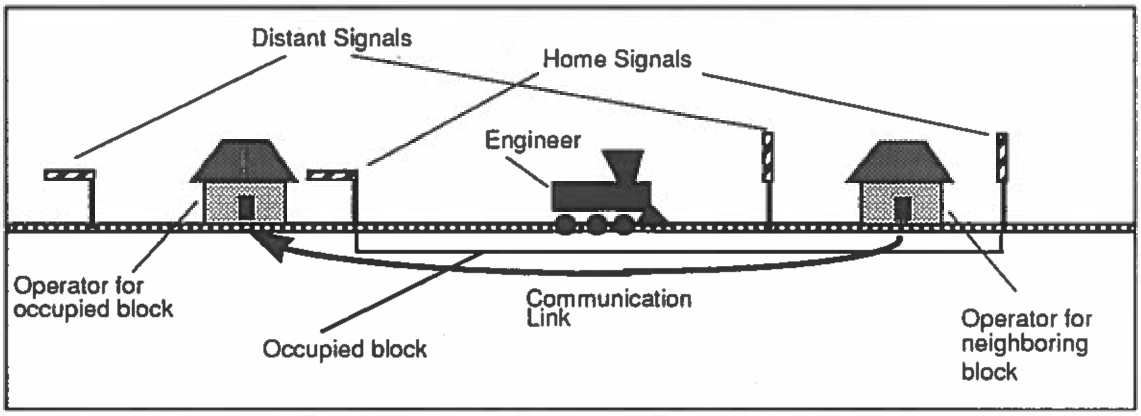
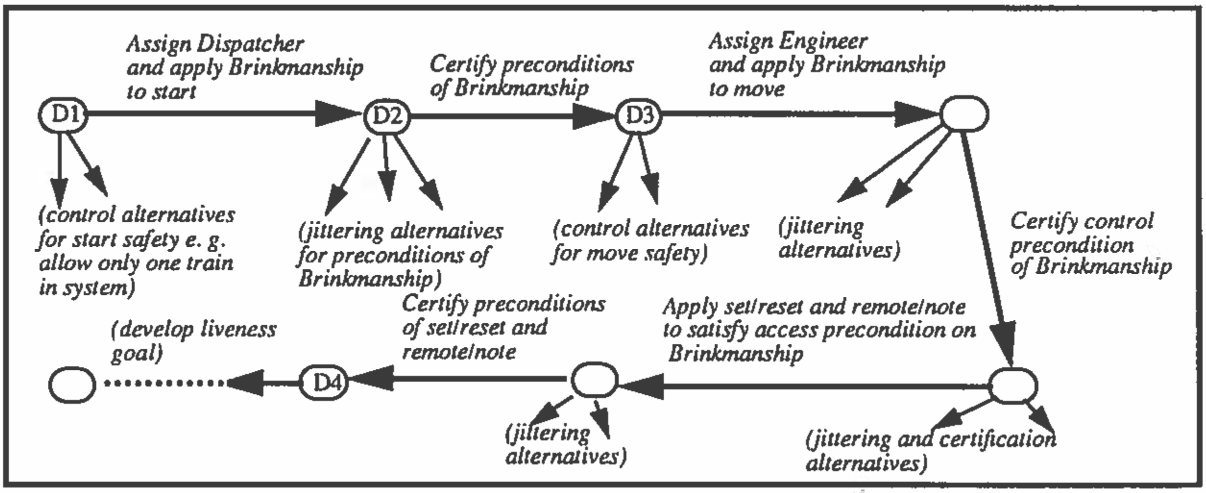


Figure 4 summarizes the first part of the path Critter takes to arrive at the target design in Figure 3. As in Figure 2, arcs coming out of a design state (node) represent alternative design moves out of that node. In the initial state D1, the system has one liveness and one safety constraint: Trains should get to their destinations, and trains should not collide. In the example, we show how these constraints are decomposed into a set of agents and inter-agent protocols which interact to satisfy the constraints.

Figure 4 Excerpt of the McGean example design space


Assign Dispatcher and apply Brinkmanship to start
Certify preconditions of Brinkmanship
Assign Engineer and apply Brinkmanship to move
D1
D2
D3
(control alternatives for start safety e. g. allow only one train in system)
(jittering alternatives for preconditions of Brinkmanship)
(control alternatives for move safety)
(jittering alternatives)
Certify control precondition of Brinkmanship
Certify preconditions of set/reset and remote/note
Apply set/reset and remote/note to satisfy access precondition on Brinkmanship
(develop liveness goal)
D4
(jittering alternatives)
(jittering and certification alternatives)

We offer a summary of our design below; a fuller discussion appears in Appendix B.

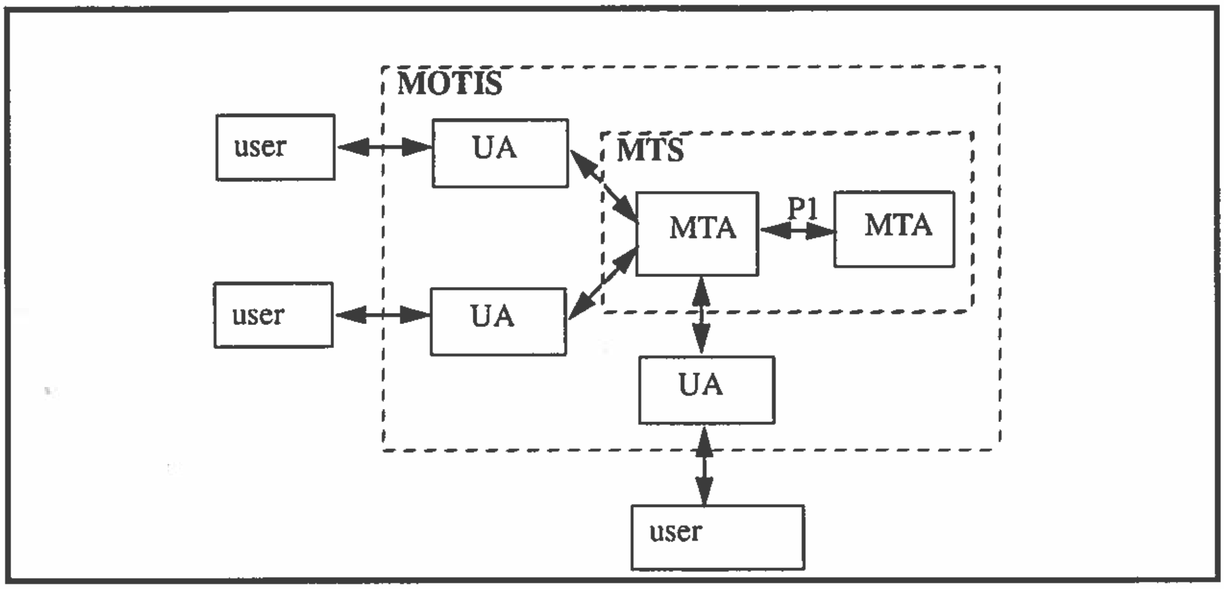
1. The designer uses the planner-based leaf-checker (as described in section 3.1) in state D1 to see whether the safety constraint is met. The planner returns a plan in which two trains collide at start-up. The designer applies the "assign responsibility" and "brinkmanship" operators. These operators are retrieved from a general catalog of operators for assigning responsibility and protecting constraints. Their effect is that a class of agents (called Dispatchers) becomes responsible for the safety constraint when trains start. Dispatchers will act to prevent trains from starting in a block already occupied by another train. This brings us to D2.

2. The designer "certifies" that every train is controlled by a dispatcher, that two trains cannot enter the same block simultaneously if they are controlled by dispatchers. These assertions are necessary if the Brinkmanship operator is to have its intended effect. Also, the designer certifies that a dispatcher knows whether a given block is occupied. In general, many of the design operators in the Critter catalog have *deferred preconditions,* conditions which must be satisfied after applying the design operator. They are typically conditions which are difficult to prove correct by analysis, or which depend heavily on domain-dependent knowledge to validate. By certifying the three conditions, the designer records her assumption that they will hold in the operating domain.
3. In D3, the designer again calls on the planner to check for violations of the safety constraint. The result is a plan in which two trains start out in different blocks, but move to the same block. The designer again uses the assign responsibility and brinkmanship operators, this time introducing the Engineer agent class and requiring them to prevent moving trains from entering occupied blocks. The designer certifies the first two preconditions of brinkmanship (control and sequential entry) but deliberates more carefully on the third, knowledge of block status. In general, it is not possible for an engineer of a train to know or predict the status of the block ahead; blocks are five to fifteen miles long, and trains can fall behind schedule or become disabled. The designer solves this problem by applying the set/reset operator. Set/reset introduces an inter-agent communication protocol; this protocol introduces a third class of agent (called Operator) which will notify Engineer agents of block status. The designer also applies an operator called remote/note which introduces a protocol by which one operator can notify another when a block becomes clear. Finally, the designer certifies the validation conditions for set/reset.
4. Finally, in D4, the designer turns to the liveness constraint. The designer uses the reachability analysis leaf-checker to disprove the constraint. After studying the result, the designer decides to assign responsibility for the liveness constraint to Engineer agents. While there are interesting details of how the final McGean design falls out of this, at a high level it is more of the same: Engineer agents are made responsible for controlling their actions so that the train progresses. In particular, no new inter-agent protocols are added to the system.

Note that the above steps do not reconstruct the *chronology* of the McGean design, which follows roughly the reverse ordering of the steps. While we could have just as easily presented the design in reverse order, we chose not to: a (superficially) accurate order might imply an attention to history that we do not attempt. We are currently satisfied with simply reproducing the responsibility assignments found in the resulting system, rationalized under our model.

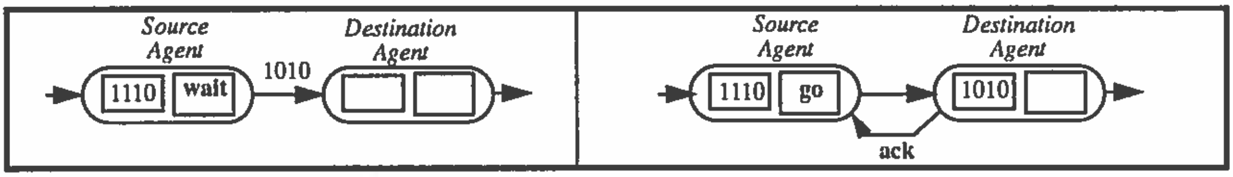
To suggest the generality of the design model, we next summarize a different rational reconstruction. The particular example we will discuss is the design of the flow control of the MOTIS P1 e-mail transfer system discussed in [Tanenbaum, 1988]. The MOTIS model is shown in Figure 5.

Figure 5 MOTIS Messaging system



The P1 flow control protocol, depicted in Figure 6, transfers mail messages over a communication link between "message transfer" agents at different sites on a network. The protocol transfers messages one at a time. As in the McGean example, the flow control aspects of this protocol must satisfy a liveness constraint (get messages to their destination) and a safety constraint (do not send a message until the previous one has arrived).

Figure 6 MOTIS P1 protocol



We can reproduce P1 flow control using composite system operators. We summarize the steps below, using (reusing) the McGean design as a foundation:

1. The designer assigns the safety constraint to a "message transfer" agent at each site. In contrast to the McGean design, the agent responsible for safety in MOTIS/P1 does not follow a message through the system, but is instead associated with a fixed location in the network. This is analogous to having station operators drive trains which are in their blocks.
2. The designer applies the brinkmanship operator to unfold the safety constraint onto each node's "send" operator. The result, as in the McGean design, is that the message transfer agent which sends a message is required to "know" whether the destination site has received that message.

3. The designer introduces the set/reset operator to allow each message transfer agent to track the receive state of its destination agent. On sending a message, the source agent sets a "wait" flag, just as the station operator sets the signal for an occupied block. When the destination agent has received the message, the source agent will clear its wait flag. The designer applies a remote/note operator, causing the destination agent to notify the source agent when it should clear its wait flag.
4. The designer assigns the message transfer agent responsibility for liveness.

The lessons we have learned from the McGean and MOTIS examples are discussed in the remainder of the section.

3.3 Contributions

The McGean and MOTIS examples suggest some of the benefits the model provides:

Semi-Automated analysis. The examples show how formal analysis tools can detect weak points in a composite system design. They point the way towards a formalization of deficiency-driven design.

Reuse. A small set of operators was sufficient to generate both of our example designs. They were reused both within a single design (multiple applications of the same operators), and between designs. This suggests that we may be able to catalog a tractably small set of design operators to generate an interesting range of composite system designs.

Formal derivation of interfaces. The process of unfolding and decomposing constraints and giving pieces of them to agents also generates the inter-agent protocols and interfaces. In the McGean design, the assignment of safety to engineers, combined with their limited access to the status of blocks, derives the need for, and rationalizes, the signalling protocol. In an earlier paper, we used the same techniques to formally specify the user interface of the elevator in our building in terms of responsibility assignments [Doerry et al., 1990].

Documentation and Rationalization. The search space generated by a design problem records the divisions of functions among different agents, and the reasons for introducing those agents. In addition, we can annotate branches with the reasons for keeping or pruning that branch, i.e., we can cache the analysis results produced by our tools at the point they are most useful.

In the next section, we identify what we feel are the most serious limitations of our results to date.

3.4 Limitations: Scale

To assess the limitations of our results, we first ask to what extent they will scale up to real problems faced by software and systems engineers. There are at least two dimensions to this question:

1. Qualitatively, any real world problem, whether train management or electronic mail, raises difficult knowledge representation and reasoning issues. Is our design model up to the challenge?
2. Quantitatively, will the sheer volume of details of a large system overwhelm our analysis efforts?

The problems we have rationally reconstructed to date have allowed us to avoid many qualitative scale issues. However, as we have attempted to tackle larger network problems, a new set of composite system design issues have surfaced:

- *Open composite systems*. In the McGean and MOTIS designs, we could hardwire much of the agent set and the communication links between agents at design time. However, in open composite systems, agents may leave and enter a system at will. Further, responsibilities may shift among agents dynamically. This raises several issues: 1) what strategies exist for dynamic assignment of responsibility, e.g., creating new agents (e. g. process forking in Unix), placing wake-up calls to existing agents (as does `inetd` in Unix), shifting responsibility to active agents (discussed under "fault recovery", below), 2) how can other agents be informed of changing assignments, 3) how can we modify our static theory of responsibility assignment to allow for temporary and/or shifting assignments, and 4) how can agents be found in an open system, i.e., how is agent addressing handled.

- *Agent hierarchies*. In our production of the McGean and MOTIS designs, all agents of a particular class were viewed as having identical abilities. However, in some composite systems applications there is a notion of a "minimal" agent for a class. This leads to differing abilities of agents in a class, all built on top of the minimal set. This leads, in turn, to problems of negotiating among agents to determine what capabilities they have in common, beyond the minimal set, and how they can communicate.
- *Fault recovery.* In the McGean design, as in safety-critical systems in general, we may put extraordinary effort into anticipating and designing out all faults. This makes sense for safety-critical goals. However, in most domains it is impossible to anticipate all the ways a system may misbehave in a production environment. Instead, one designs mechanisms that detect and recover from faults. Using this approach, one must be able to both monitor and reconfigure agents. Hence, agent management becomes a first-class concern during design. We would like to design monitoring and reconfiguration capabilities into an agent from the ground-up (as opposed to tacking it on after the fact as is done with most network management systems today).
- *Agent security and privacy.* There are several issues here. First, agents may wish to restrict read access to a subset of other agents, i.e., limit the amount of information the agent gives out. Second, an agent may wish to restrict "write" access to a subset of other agents, i.e., limit what agents can control it. In either case, agent authorization, authentication, spoofing, etc., become issues.
- *Weakened constraints*. In presenting Critter's search-based design model, we stated that the goal of the search was to make the constraints and the system consistent. In many real-world problems, we may find it necessary to achieve consistency not by changing the system, but by changing the constraints. In particular, the initial constraints may be idealized and open for modification and compromise. For example, the liveness constraint in McGean and MOTIS is unlikely to be met as stated; trains do not always get to their destinations and mail messages do become lost. We need to begin to catalog design operators that can reformulate constraints when it becomes clear that they cannot be met by further system modification. Further, we need to address both the relative and absolute utility or value that constraints carry, and use this information when considering trade-offs.
- *Brownfield constraints*. Some composite system designs have the luxury of starting from scratch with no predefined infrastructure or agents they must fit in to. We refer to these as greenfield problems - we find them quite rare. More typically, one is given an existing composite system to either change or build on. We refer to these as brownfield problems - they are common. While the domain we have chosen could be taken as greenfield -- our treatment of

McGean was in this mold -- we would like to take a browner view. For instance, in the domain of networked applications, there are standard network services (in our terms, agents) provided in both the Internet and OSI domains. Our goal is to use these services in the spirit they were defined, as building blocks. The problem becomes one of recognizing during design when existing components are usable versus when new components must be created from scratch. Given that a decision is made to use an existing network component, a second problem is tailoring that component to the design at hand.

Turning to the question of quantitative scale, our move into the network application domain increases the size of our design operator set, along with the size of the designs we examine. This raises three issues for Critter:

1. How will the growing set of design operators be indexed?
2. How will our analysis tools be affected?
3. How will heuristic evaluation be affected?

We have begun work to address the first two questions. For the first, we are looking at storing operators in an abstraction hierarchy. We have developed an algorithm for adding new operators to the hierarchy automatically. Indexing can be done on a small set of abstract operators. These in turn are refined to concrete operators. This approach is discussed in more detail in [Anderson&-Fickas, 1989]. As for the computation of the index itself, we are looking at using the analysis produced by tools like OPIE to provide clues on what can be done to squash negative scenarios. Hence, if OPIE produces a scenario that shows two trains crashing, we would like to use the steps in that scenario to index into a set of design operators that will change the system so that a train-crash scenario is no longer producable. The research question is forming a scenario-to-design-change mapper. Work that appears to be related includes that on counter-planning [Wilensky, 1983] and plan transformation [Simmons, 1988].

For the second question, that of analysis, we have begun to look at means of caching or reusing analysis results as a design proceeds. Thus, while we might place extreme effort in analyzing deficiencies in state S_i, it seems ludicrous to place the same effort in analyzing state S_{i+1}. Instead, we should look to reusing the S_i analysis in S_{i+1}, tempered by the (typically small) changes between the two states. The research issue revolves around a formalization of this "tempering" notion, and is not unlike more general research into non-verbatim replay [Mostow, 1986].

Addressing the third question is more difficult. The larger designs in the network world will generate larger design spaces, even with well-indexed selection of operators. It follows that we will need techniques to compare alternative designs, and select only those which are most promising for further development. The two examples we have shown illustrate that the designer performs virtually all of the functions of this "heuristic state selection" component in our current model. While we have done some preliminary investigation into the knowledge required to automate this task [Fickas and Helm, 1990], [Feather et al., 1991], [Fickas et al., 1991], we see this problem as too difficult to justify a parallel research effort at present. In practice, this means that the human designer will continue to be the main source of heuristic evaluation until the qualitative issues raised above have been addressed. On the positive side, Critter currently possesses a human-computer collaboration style that supports human-guided evaluation. Particular examples of this can be seen in the use of deferred conditions and certification in appendix B.

3.5 Limitations: Evaluation

Our results to date are based on rational reconstruction. In general, we have taken the standard study problems in the field, selected existing instances for each, and attempted to reproduce the corresponding composite systems. We have found two major limitations with this approach. First, the standard study problems of the field are, realistically, unimplementable. Thus we are left with producing and arguing about designs. One can see the fuzziness of this approach by studying the comparison of twelve library specifications reported by Wing [Wing, 1988]. If any of these specifications were handed to a working library analyst, they would be dismissed as hopelessly inadequate to base a production library system on[5]. Even if the problem is tightly defined (unlike the "library" problem), there remain problems with stopping at the design stage. Perhaps this is most strikingly shown in the current OSI protocol designs. While formal, neatly rationalized, designs exist for the layers of the OSI protocol stack, there was little or no attempt to actually implement and test them in a real environment. In the extreme, Internet researchers have claimed that OSI is *irrelevant* to any practical network application - its designs, no matter how formal and tightly focused, are unimplementable[6].

Our response to the design-without-implementation issue has been to choose a study domain where implementation is possible, that of networked applications. In particular, this domain has well documented, easily accessible tools for helping to implement Internet and OSI applications. Further, it has an active research and development community with good collaboration through conferences, workshops, and newsgroups.

Given a practical domain in which to work, the remaining issue is choice of problems from this domain. In particular, if we are not careful we can end up designing and implementing an ad hoc system that is difficult both to compare to other researcher's work and to reproduce in other labs. In the past, rational reconstruction has been our answer to this problem - by definition, at least one other system exists to compare our work to. As seen in this proposal, we have used this approach on the MOTIS example, one that not only has an existing design rational, but also is eminently reproducable outside of our lab. Our experience with MOTIS further supports our belief in the networked application domain as a fertile one to carry out evaluatable research:

- A well documented infrastructure and easily accessible set of agents pre-exist in both the Internet and OSI areas. This lessens the need for us to develop specialized or ad hoc systems that rely on difficult to duplicate hardware and software in other research labs.
- This same infrastructure and set of agents make interoperability an interesting evaluation criteria. Our designs (and implementations) must fit with standard components and run on standard hardware.
- There are well established standards processes for both areas. If one follows these processes in arguing for a new design (i.e., by submitting a proposed design as a draft, collecting comments, pushing up the levels) then one can expect criticism and comments from the international research *and* industry communities. In some sense, there is a formal evaluation component in place in the networking community. It requires little effort to plug into.

The next section introduces the specific problem we propose studying, that of group scheduling.

[5] We can speak authoritatively on this - ours was one of the twelve and we did hand it to a library analyst.

[6] These claims are made against the transport-layer and below. OSI designs above the transport-layer can make use of the Internet transport protocol TCP/IP by a number of clever means [Rose, 1989].

4. Our proposal

To reiterate, we have two main goals in our research: 1) to extend the composite system design model into more complex domains, and 2) to produce evaluatable results. In a very real sense, the latter has driven the former for us: we first found a domain which met our evaluation goals and then chose a problem that would force extension of our model. The domain we chose was networked applications. Our interest, specifically, is in composite systems we can design above the transport layer on the Internet and above the application layer in OSI.

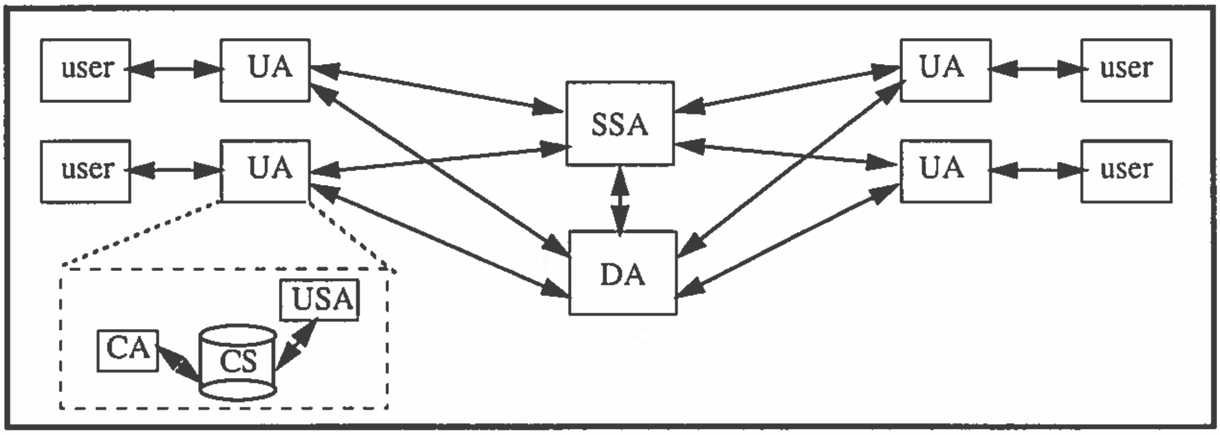
Our choice of a specific problem in this domain is that of group scheduling. We argue below for it as a useful problem for our research. However, we also note that it was one of many applications that we considered from our chosen domain[7]. We suspect most of them would make good future study problems. In other words, this is a rich design domain from a composite system perspective.

4.1 Study problem -- group scheduling support

We propose to develop an architecture to support automatic scheduling of group meetings. People who want a meeting must negotiate the place, time and participants of the meeting; when potential participants are much in demand, this negotiation process can be extremely difficult:

> The problem of finding a mutually acceptable meeting time among several calendars is a pervasive one. As one respondent pointed out, often "It takes a couple of days to catch up with everybody and ask them if they're free on such and such a day." The real-estate saleswoman will "waste a huge amount of time having to do that". According to the flight instructor, whose entire professional life is spent meeting students according to their schedule and hers, "my life is like a jig-saw puzzle in that, you know, everyone's schedule has to dovetail". [Kelley and Chapanis, 1982]

Figure 7. Simple group scheduling architecture.



We will use composite system design techniques to design and rationalize an architecture to support group scheduling. An example of the kind of architecture we might produce is shown in Figure 7. Users interact with the system through a User Agent (UA). The UA in turn is composed of

[7] In particular, the design of a distributed software engineering environment (DSEE) was a strong candidate, but was eventually rejected as too ambitious a jump. However, it remains of interest to us and we suspect that our work on group scheduling will transfer, in part, to the design of a DSEE.

two smaller agents: The user enters and consults personal schedule information via a Calendar Agent (CA), and communicates with the group's shared scheduling agent (SSA) via a User Scheduling Agent (USA). The Shared Scheduling Agent attempts to schedule acceptable meeting times among users, and updates user schedules via their USA's when a time is found. Finally, a Directory Agent (DA) allows UA's and SSA's to look find the network addresses of other agents in the system.

A typical scenario for use of the system is as follows: A user Alexis starts her user agent and requests a meeting with Bill[8]. The user agent relays this request to a scheduling server for Alexis and Bill, along with an initial set of possible meeting times based on Alexis' schedule. The scheduling server contacts Bill's user agent. It asks Bill's user agent to check Bill's calendar and approve one of the possible meeting times or suggest an alternate. When a possible meeting time is found, the scheduler agent relays that time back to Alexis' and Bill's user agents, which report it to Alexis and Bill for final approval.

We will use our composite system design tool, Critter, to formally specify a group scheduling architecture like that above. As in the McGean and MOTIS examples, Critter will derive the different types of agents in the architecture, and the protocols between those agent types. It will document the division of responsibilities among agents. Finally, it will record the design history, including alternatives considered and arguments for and against the chosen architecture and its alternatives.

Our architecture will be high-level and "policy-free" to the greatest extent possible. We will not try to prescribe group scheduling policies and algorithms in detail. Instead, our aim will be to provide mechanisms broad enough to cover at least the policies and algorithms used by popular commercial calendar managers and scheduling assistants.

We will, however, implement a specific meeting scheduler based on the architecture we develop. We believe this is a necessary step in evaluating our design techniques. We will use automated tools when available to generate implementation code from the formal specification. However, automated implementation of specifications is not within the scope of our research, so we will also rely on manual implementation methods where necessary.

4.2 Issues revisited

Our interest is in extending the composite system design model so that addresses the issues summarized in section 3.4. We believe these issues must be tackled if we are to design network applications using our model. Below we revisit the issues, but now instantiated with the group scheduling problem.

- *Open systems*. As in the train problem, we will have to deal with dynamic shifts of responsibility among agents in our group scheduling architecture. Locating agents, whether users or hardware, is more complex for group scheduling support than in the train problem. For instance, a user proposing a meeting may not always know the login name or e-mail address of every other user she would like to attend a meeting; a user is even less likely to know the port identifier of another user's user agent. It follows that our architecture will have to allow agents to be

[8] Note that Alexis and Bill may be in geographically remote locations, e.g., different domains on the Internet. In particular, one test problem we will explore is setting up the PC meeting for a Requirements Engineering conference we are putting on. PC members are scattered throughout the globe, but all have access to the Internet.

addressed by a variety of attributes such as personal names or job titles. Further, in group scheduling, as in the train problem, some agents may be mobile. User agents and schedulers may move from host to host; user agents may even move physically (if they are installed in portable PC's for instance). Thus, we may need to maintain changing logical->physical address mapping information for these agents.

- *Agent hierarchy.* [Kelley and Chapanis, 1982] state that users expect differing levels of functionality from their personal calendars. It follows that our scheduling protocol may have to deal with automated user agents with differing functionality. Similarly, we expect that different groups are also likely to apply different policies and degrees of automation to scheduling meetings. The result is that scheduler agents of similar types are likely to differ both in their capabilities, and in their parameter settings. Therefore, interacting agents in our scheduling protocol will have to determine the set of functions each supports, by inference or by communication ("option negotiation").
- *Fault recovery.* Both human and computer agents in a realistic meeting scheduler design can be expected to fail in a variety of ways. For computer agent failure, we will need to encode standard network management techniques such as SNMP and CMIS. For human agent failure, we must at least anticipate the class of failures to expect, if not the means to recover from them[9].
- *Security and privacy.* The McGean and MOTIS flow control problems did not raise obvious privacy issues, but this is a significant issue for group support. As noted in [Kelley and Chapanis, 1982], professionals typically want to keep parts of their schedules secret, although the degree of privacy expected varies widely from person to person. Group scheduling protocols need to provide "write" protection to prevent unauthorized alteration of user calendars and scheduler databases. Both read and write protection imply the need for authentication schemes in the group scheduling protocol. Without effective authentication, a malicious user might pose as another user's agent and schedule the victim for an unwanted meeting. A malicious user might also "spoof" a user agent into revealing private information by posing as a scheduler agent. We note that group scheduling includes more intricate security threats, chief among these are traffic analysis, and denial of service e. g. by overburdening an agent. We deem these less likely, and will not address them in our design.
- *Weakened constraints.* As in the MOTIS example, rigid liveness constraints are unlikely to be met. Further, rigid security constraints may be found unrealistic ideals, at least in the rather insecure Internet environment.
- *Brownfield constraints.* While we could develop our group scheduling support standard from the ground up, this is clearly not the best development strategy. The Internet and OSI networking worlds have implemented many "agents" (services) which solve parts of the group scheduling support problem. For instance, the OSI X.500 directory service [Rose, 1989] and the Internet DNS [Comer, 1991] solve significant parts of the "open systems" problem of locating agents. We will have to extend our design model to help integrate complex, predefined components like these into new designs.

As the points above suggest, our group scheduling problem includes many well-studied distrib-

[9] This is similar, in the train domain, to anticipating the ways a station may fail and producing a design that will allow such failure to be overcome. The actual means it will be overcome is found in the details of dispatch and routing tables.

uted system problems (such as security, distributed agent addressing, and fault recovery) as subproblems. Our goal is to collect existing, documented solutions of these problems from the literature they have acquired, and encode these solutions as design operators in our model. For example, we may try to develop general design operators for security by formalizing principles which have been stated informally [Moulton, 1986], or formally [Gasser, 1988]. We may also attempt to build operators for agent addressing by abstracting existing protocols *in toto*, such as DNS. We cannot hope to make an exhaustive catalog of design operators, but we will aim for a set which can generate interesting designs beyond the domain of group schedulers.

4.3 Evaluation

Choosing the network application domain allows us to make our research results available in a particularly concrete form. The end products of the proposed research will include:

1. *A formal definition of our architecture*. This will be in our extended NPN language, although we may investigate mapping to formalisms such as SDL [Belina and Hogrefe, 1988]. The definition will be semi-automatically constructed by the search process outlined in section 3.1; we will manually apply our general composite system design operators to generate states, and automatically check for potential solution states using our leaf-checking tools. Heuristic selection will be manual, using techniques we find in the network and groupware literature where possible.
2. *The design history of our architecture*. This will be entered throughout the development process in our extended IBIS tool, and stored in textual outline form. We will record general design operators and selection knowledge we develop as part of this design history.
3. *An implementation of a meeting scheduler using our architecture*. This implementation will run under systems supporting TCP/IP (via BSD sockets) and X Window output devices. Our distribution will include source code for user agents, scheduler agents, or other agents our protocol requires. It will also include ASN.1 data definitions and associated C code for information exchanged between agents. Finally, it will include operating and maintenance documentation. The code of the implementation will be produced manually, or generated by the tools in the ISODE (ISO Development Environment) [Rose, 1989].

These products will allow us to evaluate our research through by three means:

1. We will test the implementation in our own lab, and in our department. We will show our architecture is correct, and that supports the functionality of several leading commercial group schedulers and calendars, either by rational reconstruction or direct interoperability testing with the implementation.
2. We will freely distribute items from 1 to other researchers and interested vendors. This will help other researchers to (a) validate our reported results, and (b) try their own design formalisms and techniques on the group scheduling problem. It will help vendors evaluate our architecture for usefulness, and for interoperability with their products.
3. We will propose our architecture as an Internet "experimental standard". We will take our design through the process currently being defined by the Internet Activities Board (IAB), as discussed in the FTP-able file draft-iab-standardsprocess-00.txt. The strong "implement before

standardize" tradition of the Internet suggests that item 1, at least, is necessary for our proposal to succeed. We believe the documentation of items 1 & 2 will also support the standardization process.

4.4 Proposal summary

We propose studying a problem in software design that stretches the current limits of formal methods research. The problem is highly relevant to the software industry at large, and positive results should have immediate impact.

We our not starting this research from scratch. We have defined and refined our specification and design representations through a set of small but realistic composite system design problems. We have built and tested our tools on these same problems. We have given particular care to defining our design tool, Critter, in a way that is simple, practical, and expandable. The problem that we have chosen builds directly on our previous work.

Our target design is at the right level. The network community accepts discrete event diagrams as the specification of distributed applications. We can make straightforward modifications to bring our NPN representation into conformance. Further, both the Internet and OSI communities use the ASN.1 abstract syntax for describing data. While we will need to add the capability to generate ASN.1 as part of the output of Critter, we believe much of the hard work has been done for us in its definition and standardization. Last, but not least, there are standard "service elements" provided by the protocol stacks of the Internet and OSI that we can use as existing agents in our design. Thus, the design process becomes more of building applications out of stock pieces and less of generating agents from scratch in an ad hoc fashion.

We have chosen a research evaluation methodology that is based on implementable, evaluatable results. In particular, we are especially keen to make our results easily available to other researchers for redesign, reimplementation and testing.

5. Related work.

Our work is closely related to much of the work in formal development of distributed systems, such as that reported in [de Bakker et al., 1989]. Our primary differences are of emphasis and methodology. Our research to date has focused on *deriving* specifications of composite system components (agents), starting from a global statement of behavior. We have relied in particular on Feather's notion of responsibility assignment to support this process, and do not rule out techniques which are not correctness-preserving. Work such as [Abadi and Lamport, 1989] focuses on correct specification of components, and correct *composition* of those components into a globally correct system. We thus see our work as complementary.

Our approach similarly seems to complement research in formal methods in human-computer interaction, such as that described in [Harrison and Thimbleby, 1990]. That work focuses on developing abstract models of human-computer interaction which formally define properties such as consistency and learnability, and implementing interface programs from specifications guided by such models [Runciman, 1990].

Our work on composite systems extends and formalizes that of [Feather, 1987a], which in particular introduced the notion of responsibility assignment. [Dubois, 1990] has provided a formal semantics for responsibility assignment in terms of deontic logic constructs in his ERAE requirements language. The specification language we use to describe "specification states" was strongly influenced by both ERAE and Gist. Our constraint formalism also appears largely equivalent to that developed independently by Castro [Castro, 1990] to describe multiagent specifications.

Our model of design *starts* with a specification of constraints and infrastructure. A natural question is where do these come from? The most general answer is that a requirements engineering process should produce them. We have studied (and been influenced by) a specific requirements tool called KAOS [van Lamsweerde et al., 1990]. KAOS starts with an informal description of the constraints of a composite system, and calls on two powerful techniques to aid a human specifier transform them into a formal representation: 1) a meta-model of composite systems is used to fill in the pieces of a partial description, and 2) previous cases are used to do analogical reasoning [Dubisy et al., 1990]. Both of these approaches combat the notion that any design problem, composite or otherwise, starts from a blank sheet. In [Dardenne et al., 1991], we discuss the linkage between KAOS and Critter on the elevator problem. As noted in appendix D, our collaboration with the KAOS project is continuing on our more recent design studies.

We have discussed composite system design from a software engineering perspective in this proposal. However, as agents become more sophisticated, one begins to see ties to areas of AI. For instance, Bond and Gasser note in [Bond and Gasser, 1986] that

> There has been remarkably little work on Distributed AI that addresses automated problem formulation and decomposition. Correspondingly greater effort has been put into flexible task-allocation mechanisms that are used after a problem has been described and decomposed into subproblems (p. 11)

Our design model can be viewed as a semi-automatic programming system for a subclass of distributed problem solvers, ones made up of purely reactive agents. It defines classes of agents or actors [Hewitt, 1991], allocates tasks (responsibility assignments) for those agent classes, and identifies the abilities each agent class needs in order to achieve some overall system constraints. In this proposal, we begin to touch on more sophisticated DAI concepts, e.g., dynamically changing the abilities of agents [Hudlicka and Lesser, 1987], or dynamically reallocating tasks [Davis and Smith, 1983].

In our initial designs, agents were simple in that they could only react to the current state of the system, and take action according to a pre-enumerated set of rules which do not allow any inference. If a signal fails to set, for instance, an operator in our model cannot infer that another action is necessary, formulate a new plan, and take corrective action. Agents in our model cannot introspect [Morgenstern, 1986] about their abilities, nor reason about their goals or commitments [Cohen and Levesque, 1990], nor about the goals or laws of the system as a whole. Our agents are thus more similar to the "situated automata" of [Rosenschein and Kaelbing, 1986], in that they are simple machines designed to meet more complex global properties. While we have been able to synthesize, analyze, and rationalize an interesting class of composite system designs by using simple agent representation, it is clear that this representation must expand as we tackle more complex problems.

There is a rich set of issues in network applications and management. Instead of attempting to provide a compendium here, we point the reader to three books that provide a broad brush of the field: 1) Computer Networks by Tanenbaum [Tanenbaum, 1988] provides a good vertical view of the protocol stacks of current networks, 2) Internetworking with TCP/IP Volume I by Comer [Comer, 1991] looks at the popular TCP/IP protocols, and 3) The Simple Book [Rose, 1990] and

The Open Book [Rose, 1989] provide a thoroughly readable discussion of SNMP and OSI, respectively. Beyond these books, there is a deep source of knowledge in the Internet Drafts and RFCs.

6. References

[Abadi and Lamport, 1989] Abadi, M., Lamport, L., Composing specifications, in de Bakker, J. W., de Roever, W.-P., Rozenberg, G. (eds.) *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness* (Lecture Notes in Computer Science 430). Springer-Verlag, 1989.

[Anderson and Fickas 1989] Anderson, J., Fickas, S., 1989, A proposed perspective shift: viewing specification design as a planning problem, 5th International Workshop on Software Specification and Design

[de Bakker, et al., 1989] de Bakker, J. W., de Roever, W.-P., Rozenberg, G. (eds.) *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness* (Lecture Notes in Computer Science 430). Springer-Verlag, 1989.

[Belina and Hogrefe, 1988] Belina, F., Hogrefe, D., The CCITT specification and description language SDL, *Computer Networks and ISDN Systems* **16** (1988/89), p. p. 311-341.

[Bond and Gasser, 1990] Bond, A. and Gasser, L., An analysis of problems and research in AI, in Bond, A. and Gasser, L. (eds.) *Readings in Distributed Artificial Intelligence*. San Mateo, CA: Morgan Kauffman, 1988.

[Castro, 1990] Castro, J., Distributed System Specification using a temporal-causal framework (Ph. D. thesis), Imperial College of Science and Technology and Medicine, University of London, Department of Computing, 1990.

[Chung et al., 1991] Chung, K. L., Representation and utilization of non-functional requirements for information system design. In R. Anderson, J. A. Bubenko, Jr., A Sølvberg, (eds.), *Advanced Information Systems Engineering*, Proceedings, Third International Conference CAiSE '91, Trondheim, Norway, May 13-15, 1991. Berlin: Springer-Verlag, 1991, p. p. 5-30.

[Cohen and Levesque, 1990] Cohen, P. R., Levesque, H., Intention is Choice with Commitment, *Artificial Intelligence* **42** (1990) p. p. 213-261.

[Comer, 1991] Comer, D. *Internetworking with TCP/IP, Volume I: Principles, Protocols, and Architecture* (Second Edition). Englewood Cliffs, NJ: Prentice-Hall, Inc., 1991.

[Conklin and Begeman, 1988] Conklin, J., Begeman, M., gIBIS: a hypertext tool for exploratory policy discussion, ACM Transactions on Office Information Systems, Vol. 6, No.4, October 1988.

[Dardenne et al., 1991] Dardenne, A., Fickas, S., van Lamsweerde, A., Goal-directed Concept Acquisition in Requirements Elicitation, *Proceedings of the Sixth International Workshop on Software Specification and Design* (Como, Italy, October 1991), p. p. 14-21.

[Davis and Smith, 1983] Davis, R. and Smith, R, Negotiation as a metaphor for distributed problem solving, *Artificial Intelligence 20*:63-109, 1983 (reprinted in Bond, A. and Gasser, L., An analysis of problems and research in AI, in Bond, A. and Gasser, L. (eds.) *Readings in Distributed Artificial Intelligence*. San Mateo, CA: Morgan Kauffman, 1988).

[Doerry et al., 1990] Doerry, E., Fickas, S., Helm, R., Deriving user interfaces through composite system design, Technical Report CIS-TR-90-15, Department of Computer Science, University of Oregon, Eugene, OR 97403.

[Dubisy et al., 1990] Dubisy, F., van Lamsweerde, A., Requirements Acquisition by Analogy, Internal Report No. 13, KAOS Project, Institut d'Informatique, Facultés Universitaries de Namur, 1990.

[Dubois and Hagelstein, 1988] Dubois, E., Hagelstein, J., A logic of action for goal-oriented elaboration of requirements, in *Proceedings: 5th International Workshop on Software Specification and Design* (Pittsburgh, Pennsylvania, May 19-20, 1989) In *ACM SIGSOFT Engineering Notes* 14(3) (May 1989).

[Dubois, 1990] Dubois, E. Supporting an incremental elaboration of requirements for multi-agent systems, *Proceedings of the International Conference on Cooperating Knowledge-Based Systems*, University of Keele (UK), October 3-5, 1990.

[Feather, 1987a] Feather, M. S. (1987). Language Support for the Specification and Development of Composite Systems. *ACM Transactions on Programming Languages and Systems*, 9(2), 198-234.

[Feather, 1987b] Feather, M.S. The evolution of composite system specifications, *Proceedings of the 4th International Workshop on Software Specification and Design*, Monterey, California (USA), April 3-4, 1987.

[Feather, et al, 1991] Feather, M.S., Fickas, S.F., Helm, B.R., Composite system design: the good news and the bad news, *Proceedings of Fourth Annual KBSE Conference*, Syracuse, 1991

[Fickas, 1985] Fickas S., Automating the transformational development of software, In *IEEE Transactions on Software Engineering*, Vol. 11, No. 11 Nov. 1985

[Fickas and Helm, 1990] Fickas, S., Helm, R., 1990, A transformational approach to composite system specification, TR-90-19, CS Dept., U of Oregon, Eugene, Or., 97403

[Fickas and Helm, 1991] Fickas, S., Helm, R., 1991, Automating composite system design, TR-91-20, CS Dept., U of Oregon, Eugene, Or., 97403 (To appear in *IEEE Transactions on Software Engineering*, 1992)

[Fickas and Nagarajan, 1988] Fickas, S., Nagarajan, P., Critiquing software specifications: a knowledge based approach, *IEEE Software*, November 1988

[Fickas et al., 1991] Fickas, S., Helm, R., Feather, M., When things go wrong: predicting failure in multi-agent systems, Presented at the Niagra Workshop on Intelligent Information Systems, 1991. Available as Technical Report CIS-TR-91-15, Department of Computer and Information Science, University of Oregon, Eugene, OR, 97403, July 1991

[Fischer et al., 1989] Fischer, G., McCall, R., Morch, A., Janus: integrating hypertext with a knowledge-based design environment, in *Proceedings of Hypertext '89* (Pittsburgh, PA, November 1989), New York: ACM, 1989.

[Gasser, 1988] Gasser, M. *Building a Secure Computer System*. New York: Van Nostrand Reinhold Company, Inc., 1988.

[Harrison and Thimbleby, 1990] Harrison, M., Thimbleby, H. (eds.). *Formal Methods in Human-Computer Interaction*. Cambridge, Great Britain: University Press, Cambridge, 1990.

[Hewitt, 1991] Hewitt, C., Open information systems semantics for distributed artificial intelligence, *Artificial Intelligence Journal* Special Issue on Foundations of Artificial Intelligence (January 1991).

[Holzmann, 1991] Holzmann, G. *Design and Validation of Computer Protocols*. Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1991.

[Huber et al., 1986] Huber, P., Jensen, A., Jepsen, L., Jensen, K., Reachability trees for high-level Petri nets, *Theoretical Computer Science* **45** (1986) 262-292.

[Hudlicka and Lesser, 1987] Hudlicka, E., Lesser, V., Modelling and Diagnosing Problem Solving System Behavior, *IEEE Transactions on Systems, Man, and Cybernetics 17*(3):407-419, 1987. (reprinted in Bond, A. and Gasser, L. (eds.) *Readings in Distributed Artificial Intelligence*. San Mateo, CA: Morgan Kauffman, 1988).

[IWSSD4, 1987] Problem set in *Proceedings of the 4th International Workshop on Software Specification and Design*, Monterey, California, April 3-4, 1987, IEEE Computer Society Press.

[Kant, 1979] Kant, E., A knowledge-based approach to using efficiency estimation in algorithm synthesis, *Proceedings of the Sixth International Joint Conference on Artificial Intelligence,* (Tokyo, August 20-23, 1979), p. p. 457-462.

[Kelley and Chapanis, 1982] Kelley, J., and Chapanis, A., How professional persons keep their calendars: Implications for computerization, *Journal of Occupational Psychology* **55**, p. p. 241-256, 1982.

[Kemmerer, 1985] Kemmerer, R., Testing formal specifications to detect design errors, *IEEE Transactions on Software Engineering* Vol. SE-11, No. 1 (January 1985).

[Lamport, 1977] Lamport, L., Proving the correctness of multiprocessor programs, *IEEE Transactions on Software Engineering,* Vol. SE-3, No. 2, March 1977.

[London and Feather, 1982] London, P. and Feather, M., Implementing specification freedoms, *Science of Computer Programming* 2 (1982).

[Maekawa et al., 1987] Maekawa, M., Oldehoeft, A., Oldehoeft, R. *Operating Systems: Advanced Concepts*. Menlo Park, California: The Benjamin/Cummings Publishing Co., 1987.

[McGean, 1976] McGean, T. *Urban transportation technology*. Lexington, MA: D. C. Heath and Company, 1976.

[Meister, 85] Meister, D. *Behavioral Analysis and Measurement Methods*. New York: John Wiley and Sons, 1985.

[Morgenstern, 1986] Morgenstern, L., A first-order theory of planning, knowledge, and action, in Halpern, J. (ed.) *Theoretical Aspects of Reasoning about Knowledge: Proceedings of the 1986 Conference* (Monterey, California, March 19-22, 1986). Los Altos, California: Morgan Kauffman Publishers, Inc., 1986.

[Mostow, 1986] Mostow, J., Why are design derivations hard to replay, In T. Mitchell, J. Carbonell and R. Michaelski (eds.). *Machine Learning: A Guide to Current Research*, pages 213-218. Hingham, MA: Kluwer, 1986.

[Moulton, 1986] Moulton, R. *Computer Security Handbook: Strategies and Techniques for Preventing Data Loss or Theft*. Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1986.

[Nance, 1986] Nance, J. *Blind Trust*. Macmillan, 1986.

[Robinson, 1990] Robinson, W., A multi-agent view of requirements, *Proceedings of the 12th International Conference on Software Engineering*, Nice, France, 1990.

[Rose, 1989] Rose, M., *The Open Book: A Practical Perspective on OSI*. Englewood Cliffs, NJ: Prentice-Hall, Inc. 1989

[Rose, 1991] Rose, M. *The Simple Book: An Introduction to Management of TCP/IP-based Internets*. Englewood Cliffs, N. J., Prentice-Hall, inc., 1991.

[Rosenschein and Kaelbing, 1986] Rosenschein, S., and Kaelbing, L. P., The synthesis of digital machines with provable epistemic properties, In Halpern, J. (ed.). *Proceedings of the Conference on Theoretical Aspects of Reasoning About Knowledge*. San Mateo, CA: Morgan Kauffman, 1986.

[Runciman, 1990] Runciman, C., From abstract models to functional prototypes, in Harrison, M., Thimbleby, H. (eds.). *Formal Methods in Human-Computer Interaction*. Cambridge, Great Britain: University Press, Cambridge, 1990.

[Shaw, 1961] Shaw, Robert B. *Down Brakes: a History of Railroad Accidents, Safety Precautions, and Operating Practices in the United States of America*. London, United Kingdom: P. R. Macmillan Limited, 1961.

[Shoham, 1991] Shoham, Y., AGENT0: A simple agent language and its interpreter, *Proceedings: Ninth National Conference on Artificial Intelligence* (Anaheim, CA, July 14-19, 1991), AAAI press, 1991, p. p. 704-709.

[Simmons, 1988] A theory of debugging plans and interpretations, *Proceedings of AAAI-88*, 1988.

[[Tanenbaum, 1988] Tanenbaum, A. *Computer Networks*. Englewood Cliffs, N. J.: Prentice-Hall, 1988.

[van Lamsweerde et al., 1991] van Lamsweerde, A., Dardenne, A., Dubisy, F., KAOS Knowledge Representations as Initial Support for Formal Specification Processes, Research Paper RP24/91, Facultés Universitaries de Namur, May 1991.

[Wilbur-Ham, 1985] Wilbur-Ham, M. C., Numerical petri nets -- a guide. Telecom Australia Research Laboratories, Report 7791.

[Wilensky, 1983] Wilensky, R. *Planning and Understanding*. Addison-Wesley Publishing Company, 1983.

[Wing, 1988] Wing, J., A study of 12 Specifications of the library problem, *IEEE Software*, July 1988.

[Wirfs-Brock et al., 1991] Wirfs-Brock, R., Wilkerson, B., Wiener, L., Designing Object-Oriented Software, Prentice Hall, 1991

[Yakemovic and Conklin, 1990] Burgess Yakemovic, K. C., Conklin, J., Report on a development project use of an issue-based information system, *CSCW 90 Proceedings*, (October 1990).

Appendix A: The Critter Model

In this section, we discuss the Critter model and the representations we use in more detail. This material is summarized in section 3.1.

A.1 States

Each design state in our search space represents a single composite system design. A state in Critter has two components: a set of *constraints* on the design, and a *system* that generates behaviors of the design. The overall goal of the search is to bring the two into "consistency" in a single state -- the behaviors produced by the system should satisfy the constraints. As this implies, the system and constraint components may be misaligned in any one state, i.e., the system may generate behavior that breaks the constraints. Be aware that this is a twist on most state-based problem solving where the "goals" of the system are the "goals" of the search. The goals of our system are the constraint portion of the design state. The goal of the search is to find a match between the two state components:

1. The constraining portion of a design state consists of a set of *constraints*, i.e., constraints on behavior. Constraints are expressed declaratively in terms of system-wide properties. They are formal versions of statements such as "Trains get to their destinations", or "Trains don't crash".
2. The generative or system portion of a design state denotes a set of possible behaviors. A behavior is a sequence of events that could occur in the composite system. In our model, the system is represented by a specification in a language we have adapted for composite system specification as explained below.

The problems that we have focused on can be characterized as liveness-safety problems [Lamport, 1977]. Typically, they involve constraints of two general forms:

liveness: $\forall B_i \;\; \exists p \in B_i$ s. t. **satisfies**$(p, C_{achievement})$

safety: $\forall B_i \neg \exists p \in B_i$ s. t. **satisfies**(p, C_{unsafe})

Here, B_i is a behavior produced by the system, p is some intermediate point or state of affairs of that behavior, $C_{achievement}$ is a desirable condition such as getting something delivered, getting a message through, configuring the system in a particular way, and C_{unsafe} is a condition we wish to avoid such as a race condition, a security breach, or a lost message. In toy worlds we may be able to drop one class of constraints (such as the liveness constraints) almost entirely, or pretend that all constraints can be met strictly. In the real world there is often an intricate balance between the two classes.

The system portion of a design state is further divided into two parts. First, there is what we call the infrastructure. Examples of an infrastructure are the rail-lines of a railroad system and the communication-lines of a network system.

The system portion also includes a set of agents. Agents use the infrastructure of the system and interactions with each other to produce behaviors. In general, an agent is a component of a composite system that can sense a portion of the system's state, make decisions, and perform or prevent actions of the system that the agent controls. *Assigning responsibility* for a constraint to a class of agents requires that all agents in that class limit their actions so the constraint is achieved. Informally, only those agents responsible for a constraint are expected to limit their own behavior to ensure satisfaction of that constraint. For example, if a train engineer/agent is solely responsible for keeping her train from colliding with another, then she must limit her actions accordingly, and in particular, cannot rely on other agents to constrain their actions to avoid collision.

An agent class in our representation has the following attributes:

1. A set (possibly empty) of the system state an agent can directly observe.
2. A set (possibly empty) of actions an agent can perform that affect the system state.
3. A set (possibly empty) of local state information.
4. A set (possibly empty) of responsibilities for achieving a constraint or goal of the system. An agent may be *overloaded* in our representation: it may have more than one responsibility in any single composite system, and it may have separate responsibilities in two or more different composite systems. It may also *share* responsibility for a constraint with another agent in the same system or a different system.

Both the constraining and system portions are represented in notations we have adapted from existing work in formal specification. No existing notations had exactly what we need, so we have followed the approach of Feather [Feather, 1987a], who shows how to extend the Gist language to support composite system design. We have adopted less powerful languages than Gist, however, so that we can use the automated reasoning techniques we introduce in later portions of the paper.

Our constraint language is a style of modal, temporal logic roughly similar to that of that of the ERAE language [Dubois and Hagelstein, 1988], and is very similar to the "temporal-causal logic" independently developed by Castro [Castro, 1990]. It is possible to express both safety and strong liveness requirements in the language using **always, ifEver, eventually, never** [Lamport, 1977].

The language for the system portion can be viewed, alternatively, as a subset of Gist [London and Feather 1982] or a superset of Numerical Petri Nets (NPN) [Wilbur-Ham, 1985] that adds the notion of agents.We will use the Petri net view in this paper for presentation purposes.

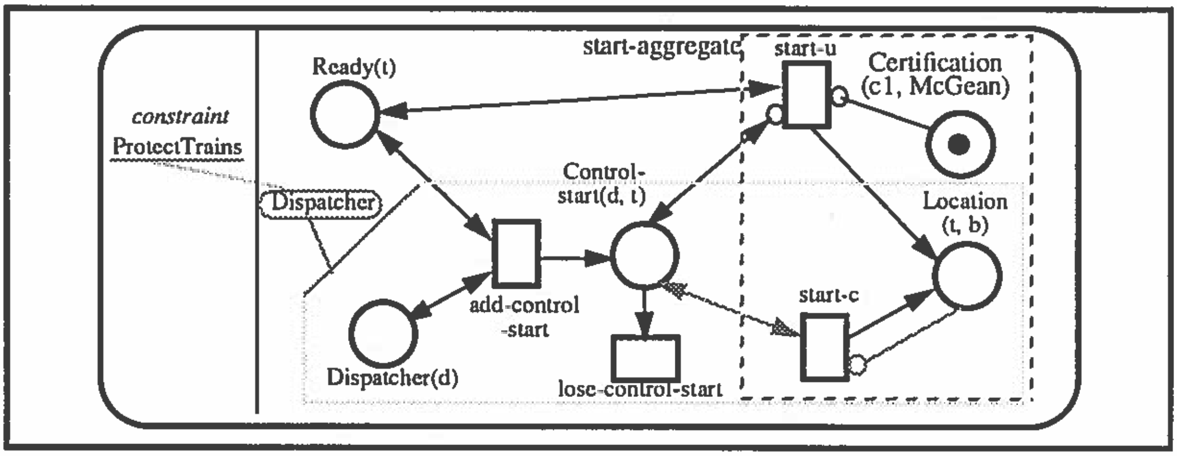
Figure A.1 gives an example of our presentation style for design states. We use the left portion to represent the constraints and the right portion to represent the system. Links between the two halves represent responsibility assignments. In Figure A.1, such a link can be seen between the constraint ProtectTrains (defined more formally in section B.1) and a dispatcher agent.

The system description uses an extended NPN that differs from a standard Petri net in the following ways:

1. Transitions (drawn as solid boxes) may have predicates associated with them. We do not include the predicates in the graphical representation, but explain them as necessary in the text. Double-headed arcs attached to a transition indicate that the transition uses, but does not modify or consume, a corresponding token.
2. Places (drawn as open circles) contain structured tokens; we will view tokens as records in our examples. They will be represented graphically as either solid circles (see Figure A.1) or representative icons (seen in later figures). Dot notation will be used to refer to the fields of a token where necessary.
3. A not-arc is included. It is drawn as an arc with a circle on its destination end. If a transition has a not-arc coming into it, it is disabled from firing whenever *there is* a token at the arc's source.

4. Some arcs represent potential links - they are drawn as gray arcs. Two such arcs, one a plain arc and one a not-arc, can be seen attached to the start-c transition in Figure A.1. Their use is discussed in the first example (section B.1).
5. Agents annotate the net. In general, portions of the net representing either local state of an agent, or things that the agent can sense and control, are bounded by dotted lines and labelled by the associated agent.
6. Dashed lines bound pieces of the net that we wish to treat as a whole for transformation purposes. In Figure A.1, we might want to view the two start transitions as an aggregate if we are about to make a change to the system state.

Figure A.1 Example design state (taken from Figure B.3).



A.2 Leaf Node Checkers

A solution state or leaf node in our search process is a composite system design where the behaviors produced by the system portion (a Numerical Petri Net) satisfies all of the constraints in the constraint portion. We need a way to test for satisfaction to identify potentially successful designs. We use three automated reasoning tools to test a state for leafhood:

Analysis Tool 1: A planner or scenario generator called OPIE [Anderson and Fickas, 1989]. In general, OPIE attempts to show that a state is *not* congruent by showing that safety or liveness constraints are *not* met. OPIE does this by finding a plan that violates the constraints, in effect producing a counterexample for the constraint. For example, it might prove that the constraint "No two trains are ever in the same block of track" is violated by generating a plan for putting two trains in a block. As we discuss below, we believe generating counterexamples to constraints, rather than verifying constraints are met, can give the designer useful guidance on the operators to apply to a state if it is not a leaf (solution).

However, in addition to testing leafhood, we also use OPIE to show that a set of constraints *could* be met if agents cooperated in the right way. OPIE does this by producing a plan that satisfies the set of constraints, effectively proving the existence of a correct behavior. As with counterexamples, an existence plan may provide insight into what operators to apply to a non-leaf state to transform it into a solution.

Analysis Tool 2: A reachability-graph (RG) tool. The tool first produces a reachability graph from a static analysis of the NPN, and then allows queries about reachable states. For example, the tool can reply to queries such as, "Is it possible for a train to fail to reach its destination"?. It uses omega values [Huber et al, 1986] to represent infinite plans/behaviors. As with OPIE, these queries can be used to provide existence proofs.

Analysis Tool 3: An NPN simulator. To allow us to test a full blown NPN, we have implemented an NPN simulator that "runs" an NPN forward. At non-deterministic choice points, alternatives can be either presented to the human designer for selection, or controlled by simple rules, e.g., always choose transition A over transition B. For example, we might run the NPN simulator on a test case that attempts to put two trains into the same block of track, in order to validate that this cannot occur. The details of this tool, and a semi-automated means of selecting appropriate test cases to feed it, is discussed in [Fickas and Nagarajan, 1988]. It should be noted that this tool was initially tested in the library domain where few standards of automation exist. It has proven more usable in the networking problems we have studied (and will discuss shortly) where standards are tight. In essence, the problematic "matching problem" discussed in [Fickas and Nagarajan, 1988] has been made tractable.

For pragmatic reasons, we see a continuing need for all three of these tools. OPIE is not able to operate on the full NPN language, and can be expensive to run. Further, it can only generate finite counterexamples. Thus, counterexamples that may require an infinite loop, such as those to disprove a liveness goal "every train gets to its destination", are problematic. The RG tool, in contrast can use of omega values to represent infinite paths/behaviors. However, it operates on a smaller subset of the NPN language than OPIE, and runs even more slowly. Finally, the NPN simulator, when provided with a suitable set of test cases, can produce counterexamples on a full NPN that neither OPIE or the RG could generate, whether due to resource or to language constraints. However, it does require that the domain be one where test cases can be found and matched, and requires the most human supervision.

Even with all of these analysis tools, there is typically a significant need for human intervention in proving constraints, either in simplifying the system representation to one that is handled by OPIE or the RG tool, or in producing the right test cases for and evaluating the results of the NPN simulator. For the problems we have studied to date, two of which we present in this paper, this intervention has stayed in tolerable bounds for the human designers involved.

A.3 Search operators.

Search operators (henceforth, design operators) are what transform one composite design state into another. They are applied to a non-leaf node N to remedy a deficiency found in N. A design operator may apply to either component of a design state, i.e., the constraints or the system. Further, it can apply to either system component, modifying the infrastructure or the agent set.

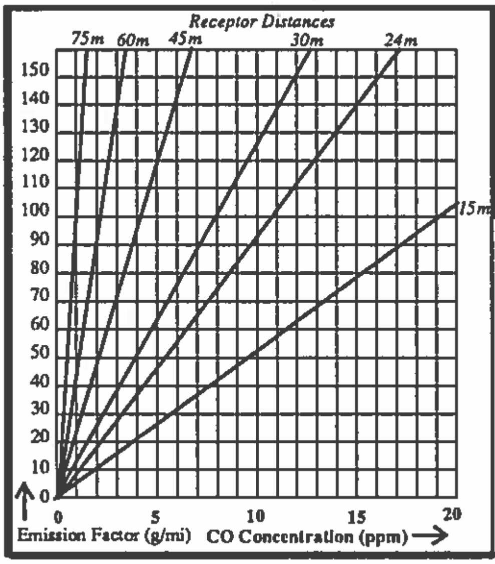
Because our interest is in composite system problems, our design operators are tailored to multi-agent, distributed-action concerns: creating agents, splitting responsibility for a constraint among agents, establishing communication protocols among agents. Examples of each of these will be seen in our example designs.

The actual representation of operators has taken a transformational appearance: a design operator has a left-hand-side pattern that matches a piece of NPN (or constraint), a left-hand-side precondition that establishes the validity of the transformation, and a right-hand-side action that computes a replacement pattern (for example, see Figure B.2 in appendix B). The Glitter system has been used as a model for this representation. As with the Glitter system, computing preconditions uses a mixture of special-purpose provers and human verification. A good example of the latter is seen in the application of the *brinkmanship* operator in section 3.2 (and appendix B), which assigns a last-ditch type of control for constraint G to agent A. However, the correct application of the operator requires that A have access to parts of the system state which could violate G. This access precondition is usually highly problem-dependent and difficult to prove automatically. Our style is to allow the operator to be applied without proof of access, but to flag the access precondition as one that is still open (seen as grayed arcs in Figure A.1). The designer may either assert that the precondition holds or do further design to make it hold (known as jittering in Glitter).

A.4 Heuristic evaluation.

There are a relatively small number of operators that account for the composite system problems that we have studied. However, they are capable of producing a large set of alternative designs, enough so that blind search becomes intractable. We currently rely on the user to provide the necessary domain knowledge to guide the search. At the same time, we are actively exploring means to remove the user from this role, or at least minimize her involvement. There are at least two approaches worth pursuing. First, we can produce all alternatives of a node N, but attempt to prune bad designs early by heuristic evaluation. In essence, we attempt to apply domain-specific knowledge to predict which child is more likely to lead to a good design. To show the difficulty of this approach, consider an early composite system design we studied, that of the seemingly simple design of a highway-railroad crossing. Figure A.2 shows a portion of the design evaluation criteria we found for this design.

Figure A.2 Rail-highway selection models.I



School bus safety indicators

1. # of school bus crossings
2. # of students on the bus during crossing
3. # Train operations during crossing times
4.% of crossings by buses at-grade
5. # Historical Bridge/Bus Incidents
6. Alternative route delay

Aesthetic impacts

Insignificant: The proposed improvement would create visual characteristics similar to those that currently exist in the landscape.
Low: The improvement would introduce additional visual characteristics into the landscape that would be evident but would not necessarily attract attention.
Moderate: The improvement would introduce visual characteristics that would be noticeably different from existing visual elements.
High: The improvement would visually dominate the landscape and would cause substantial change in the visual character of the landscape.

Accident prediction factors (excerpt)

Category	Constant.	Exposure Factor	Main Tracks Factor	Day Thru Trains Factor
Passive	0.002268	$((c * t * 0.2) / 0.2)^{0.3334}$	$e^{0.2094mt}$	$((d+ 0.2) / 0.2)^{0.1336}$
Flashing	0.003646	$((c * t * 0.2) / 0.2)^{0.2953}$	$e^{0.1088mt}$	$((d+ 0.2) / 0.2)^{0.0470}$

The four criteria are taken from [Taggart et. al., 1987]. They include graphical models for computing carbon monoxide emissions, guidelines for evaluating aesthetic impacts, checklists of special concerns such as emergency vehicle and school bus movements, and statistical models to predict accidents at crossings with different protocols and signals.

In general, to genuinely evaluate design alternatives, we have to evaluate the disruption a system will create in its environment. To automate this, we must find a way to both represent the type of knowledge in Figure A.2, *and* integrate it into a design tool such as Critter. While we have done some preliminary investigation into the knowledge required to automate this task [Fickas and Helm, 1990], [Feather et. al., 1991], [Fickas et. al., 1991], it remains an open problem.

A second approach is to replace generate-and-test with analyze-and-select. Analyzing the deficiencies in the current state may allow us to be more selective in the operators we retrieve. Our approach is to use a counterexample plan as a guide to assigning credit and blame. Our early results in using plans to assist operator retrieval are discussed in [Anderson & Fickas, 1989]. However, as we note in section 3.4, deficiency-driven control is an important and still open problem in automated design. In practice, this means that the human designer will continue to be the main source of heuristic evaluation in Critter for the foreseeable future.

Appendix B: McGean Example

In this section, we will use Critter to design a train management system. Before starting the example, we note that much of our motivation for work on the Critter model stems from a belief in *interactive* tools to support an *incremental* design process. One of our early attempts of interactive tool building, the Glitter system, was judged promising from a purely formal sense, but seriously flawed in a practical sense. At its heart it was overly complex in its implementation, and crude in its interaction with a human designer. Our solution has been to simplify our current tool, Critter, by using a standard, well-understood paradigm, that of state-based search. We have also provided deferred preconditions and certification as an aid to formal verification.

The reader may wish to refer back to Figure 3 on page 9, which shows the McGean design that is the target of our design process. This design attempts to prevent collisions of trains by allocating regions of track of 5 to 15 mile lengths (called blocks) to a single train at a time. Each block has a single station associated with it. When a train T enters a block B (by passing or stopping at the station for B) the operator of that station sets a signal that block B is now occupied. No other trains may enter B until the signal is cleared. The operator clears the signal when the next operator down the line sends a signal back that the train T has been spotted (and hence, has vacated block B).

The occupied/clear signal is actually broken into two components, a "distant" signal and a "home" signal. Both are always in the same state. The distant signal is set out far enough from the station to guarantee that a train can stop before reaching the station. The home signal is in front of the station. In essence, the distant signal tells a moving train when to start stopping, and the home signal tells a stopped train when to start moving. If a train sees a clear distant signal then it will see a clear home signal (see the constraint of one train per block at a time).

If one looks at the final solution from a composite system view, the design designates engineer agents as responsible for entering a block of track between stations only if the signal for that block reads clear (vertical). Station operator agents are given the responsibility of 1) setting the distant/home signal to occupied (horizontal), 2) resetting the signal to clear, and 3) passing information among themselves. Finally, dispatcher agents (not shown in Figure 3) are responsible for making sure trains enter a system in a safe fashion.

B.1 The initial state

The initial state of the train example is shown in Figure B.1. This starting state represents the train management system that immediately preceded that of McGean - a schedule-based system that relied on clever timing of trains to avoid collisions. As noted in [Shaw, 1967], the schedule-based design had its problems, causing an unacceptable number of accidents. This, along with the invention of telegraph, sparked an interest in finding a new design.

More generally, we view the initial state as the infrastructure that is already in place when design commences (or what we referred to as brownfield constraints in section 3.4). Typically, the infrastructure represents the piece of the system that has the greatest modification cost, and hence, is the piece one tries to design around, or in a more positive light, on top of.

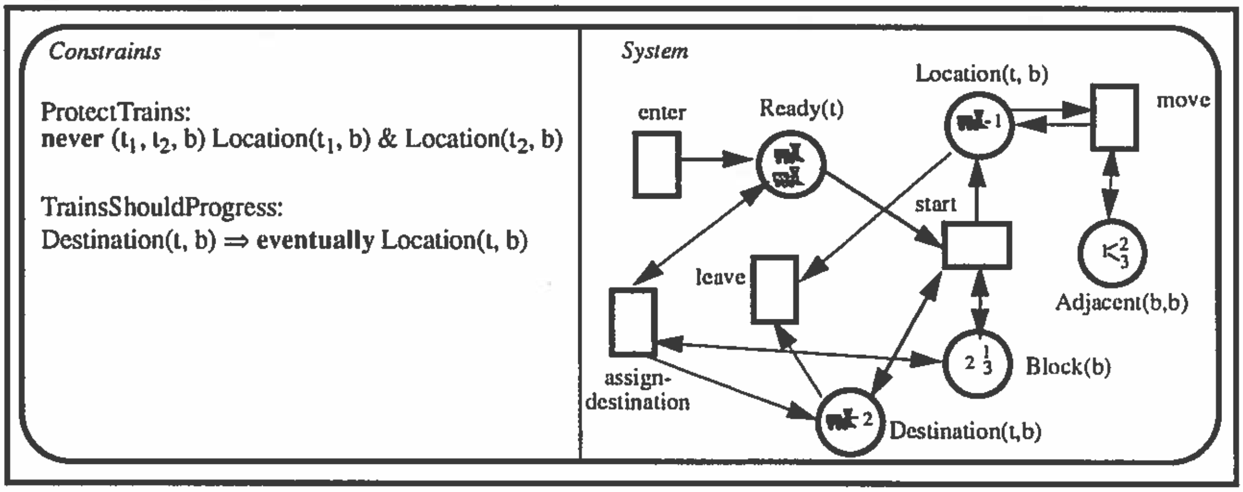
As for the actual construction of the initial state, we assume that it has been hand-engineered by the designer, possibly assisted by a system such as the KAOS [van Lamsweerde et al.] assistant discussed in section 5.

As seen in Figure B.1, we will use an enhanced version of an NPN (Numerical Petri-net [Wilbur-Ham, 1985]) to represent both infrastructure and agents of a system. The details of our state representations can be found in Figure A.1. While we typically start with a simple system description, the design process tends to increase complexity quickly. Hence, we will elide or abstract views of the system state for clarity.

The initial system portion appears at right, the constraint portion at left. We can paraphrase the specification of the system portion in this figure as follows:

- Trains can be created. The transition enter (box at upper left of the system portion) fires non-deterministically to introduce a train token t on the place labeled Ready(t). The Ready place denotes the relation "Train t is ready to start".
- A train t for which Ready(t) holds can be assigned a destination block by the transition assign-destination.
- If a train t is Ready and has a destination, it can start (transition start can fire). Starting puts t at a location block, denoted by the place Location(t, b).
- Trains for which Location(t, b) holds can move by firing the transition move, which places them at a new location adjacent to their original location.
- Trains at their destination location can leave the system by firing the leave transition.

Figure B.1 Initial state of the McGean example.



Constraints

ProtectTrains:
never (t_1, t_2, b) Location(t_1, b) & Location(t_2, b)

TrainsShouldProgress:
Destination(t, b) ⇒ **eventually** Location(t, b)

There are two constraints in state D1: two trains should never be at the same location (safety); trains should eventually get to their destination (liveness).

Safety constraints are handled by reformulating a constraint of the form

safety: $\neg \exists\, B_i, p \in B_i$ s. t. **satisfies**(p, C_{unsafe})

as its negation:

not safe: $\exists\, B_i, p \in B_i$ s. t. **satisfies**(p, C_{unsafe})

If OPIE can find a plan (i.e., a behavior) to achieve C_{unsafe} then the safety constraint is violated. As we will discuss below, the plan that led to a violation may contain important information. In particular, analyzing the "bad" plan itself gives rise to a deficiency-driven design process: we can limit the search for constraint satisfaction to design operators which could prevent the bad sequence of events from occurring.

We negate constraints in the same way to analyze liveness constraints:

liveness: $\forall B_i \; \exists p \in B_i$ s. t. **satisfies**(p, $C_{achievement}$)

becomes

not live: $\exists B_i$ s. t. $\forall p \in B_i \neg$ **satisfies**(p, $C_{achievement}$)

However, as noted, OPIE cannot prove the "not live" condition in all cases, so we must rely on the reachability graph or NPN simulator tools.

Finally, a strong liveness constraint may be weakened during the search process[10]:

liveness: $\forall B_i \; \exists p \in B_i$ s. t. **satisfies**(p, $C_{achievement}$)

could be weakened to

possibility: $\exists B_i, p \in B_i$ s. t. **satisfies**(p, $C_{achievement}$)

In other words, we may decide that we can live with a system which gives us at least one way to reach $C_{achievement}$, even though not all behaviors do. We can use OPIE to look for plans that support these types of weakened liveness constraints.

We start by asking if the design state of Figure B.2 is consistent. We must determine whether the system-side NPN generates behaviors (scenarios) that can violate one or both constraints, e.g., two trains move to the same location, a train loops endlessly between two locations. We will start by looking at the safety constraint that specifies a state that must be maintained indefinitely:

(D): Find a counterplan to ProtectTrains in D1 given Block(b1)

The human designer, D, asks OPIE to find a sequence of events that violates ProtectTrains. OPIE automatically generates a scenario in which two trains enter the system at the same location, a violation of the ProtectTrains constraint. A formatted form of that scenario is given below.

(O): The constraint ProtectTrains is violated by scenario S1 in state D1:

1. given Block(b1);
2. enter(t1|Train) produces Ready(t1);
3. enter(t2|Train) produces Ready(t2);
4. start (t1, b1) produces Location (t1, b1);
5. start (t2, b1) produces Location (t2, b1);

Violation: ProtectTrains in D1

The above scenario implies that we are not at a leafnode of the design. In particular, the `satisfies?` predicate of Figure 1 returns false given `b` is bound to the scenario above and `c` is bound to the ProtectTrains constraint. From this point, we need to choose among what design operators are available for transforming D1 into something closer to a solution. The designer has three alternative (but complementary) methods to address the negative scenario S1:

1. Modify the infrastructure. The designer could decide to provide separate arrival points for each train, making it impossible to generate the "crash on arrival" scenario of S1.

[10] To reiterate, given that we are not at a leaf node, two classes of search operators are available: 1) the system can be changed to produce different behaviors, ones that fit the constraints, *or* 2) the constraints can be changed to fit the behaviors produced by the system. Weakening a constraint is an example of the second.

2. Weaken the safety constraint. Some past train systems took this approach by allowing more than one train in a block [Shaw, 1961].
3. Assign responsibility for the safety constraint to a class of agents. This requires agents in the class to control their actions so that the constraint is met.

The designer chooses the third method, and introduces a new agent class (subsequently called "dispatcher") that controls the arrival of trains into the system. The designer assigns instances of the dispatcher agent (henceforth, "dispatchers") to control trains so that they do not collide. But now the designer must define how dispatchers will protect the constraint. Using scenario S1 as a guide, the designer decides that a dispatcher should control the choice of starting location for a train. This is done by instantiating a standard operator, called brinkmanship, that maintains a constraint by manipulating transitions so that some but not all the constraint conjuncts are allowed to become true at the same time. Stated another way, the agent prevents transitions that are the last step in breaking the constraint[11]. In our example, this means never allowing a train to start at the same location as another train.

The abstract brinkmanship operator is shown in Figure B.2 in NPN form. The *pattern* at top specifies what must hold to apply the operator initially. Given that an agent A is responsible for a constraint G, applying brinkmanship requires the analyst to:

1. Factor G into a "controlled" condition P1, and a "brink" condition P2, such that P1 & P2 violate G. In this example step, P1 and P2 are both Location(t,b).
2. Identify a transition T that can produce P1 when P2 is true, violating the constraint G. The agent will control T so that it cannot fire when P2 is true, preventing T from pushing the system over the "brink" by producing P1 & P2.

   Based on scenario (1), the analyst selects the "start" transition for the role of T in our example step.
3. Select a "client" object class C that the agent A will manipulate to control T.

   The client class in this application of the operator is trains; the analyst reasons that the start time of trains can be manipulated to prevent their collision.

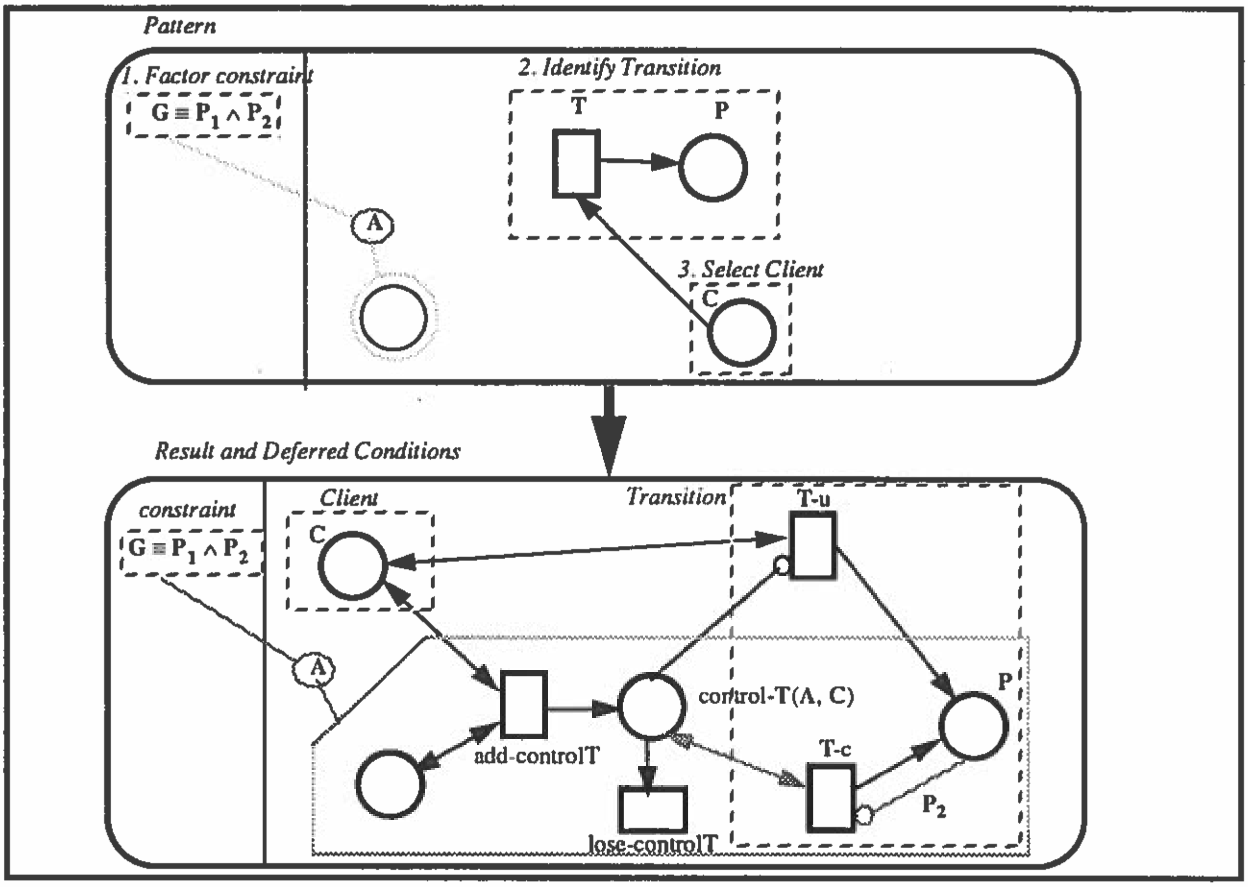
When the pattern conditions have been filled in, the system will apply the brinkmanship operator. It replaces the transition T with two separate transitions: T-c, a version of T that is under the control of agent A; T-u, a version of T that is uncontrolled. T-u acts as a type of reality check, representing the behavior of the system if the control regime fails. We will return later to the designer's handling of Tu in the McGean design.

Finally note that the grey arcs in the bottom half of Figure B.2 represent *deferred preconditions* on the operator; after the brinkmanship operator has been applied, the analyst must verify that the controlling agent (the dispatcher in this case) can:

1. Control train starts, and
2. Test whether another train is at the location of a train about to enter.

[11] Other control strategies for maintaining constraints include 1) never allowing a specific condition in a constraint to ever be true (e.g., never allowing trains in the system), and 2) ensuring that two conditions in a constraint are mutually exclusive (never allowing more than one train in the system).

Figure B.2 Brinkmanship operator.



The designer may either assert that the precondition holds or do further design to make it hold (known as jittering in Glitter).

Critter, like Glitter, includes deferred preconditions in operators for practical reasons:

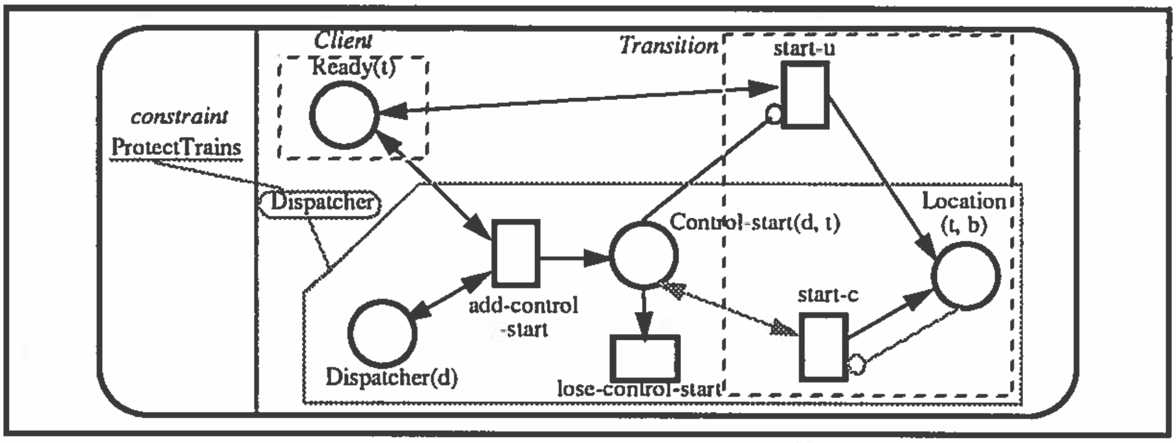
1. Deferring preconditions improves problem-solving efficiency. Typically, deferred preconditions will be conditions that are difficult to prove formally, or that require considerable additional design effort ("jittering") to satisfy. If Critter requires the analyst to prove all preconditions of an operator immediately, and the analyst later backtracks over that operator, much of the proving effort will have been wasted. Moreover, the analyst may be able to satisfy a deferred precondition serendipitously, when working on some other subproblem of the design [Feather, 1987a].

2. It can improve usability for the analyst. Proving some preconditions of an operator may require substantial effort from the analyst. If forced to prove these preconditions immediately, the analyst may lose his "train of thought" and the thread of the development. Also, as we discuss below, many of the deferred preconditions of the operator will be proven by the analyst inspecting the design and "certifying" that they are true. In general, it may be difficult for the analyst to assess these preconditions without actually applying the operator and seeing its effects.

Returning to our example step, the designer asserts that the dispatcher has direct control over trains and direct access to the status of blocks (i.e., the designer changes the grey arcs, representing deferred preconditions, to black ones). Figure B.3 shows the parts of the design state affected by the brinkmanship operator and verification of its preconditions. The start transition is split into a controlled and an uncontrolled version. The controlled version has a not-arc to test the status of blocks, and a control place representing dispatchers controlling trains. We can paraphrase Figure B.3 as follows:

> Trains normally arrive (start-c) under the control of a dispatcher (Dispatch-control). Using start-c, a train is not allowed to enter at the same location as an existing train. If no dispatcher is in control of a train then it is possible for it to enter unrestricted via the transition start-u.

Figure B.3 System state D2 (excerpt): Application of brinkmanship operator to start transition.



In terms of the transformation schema in Figure B.2, the start transition plays the role of T, and Location(t, b) plays the role of P. The constraint ProtectTrains plays the role of G; it is factored into $\mathbf{P_1} = \exists$(t1, b) Location(t1, b) and $\mathbf{P_2} = \exists$(t2, b) Location(t2, b). In the resulting system, start-c and start-u correspond to T-c and T-u, respectively, while Control-start takes the role of the control place Control-T.

B.2 The use of certification

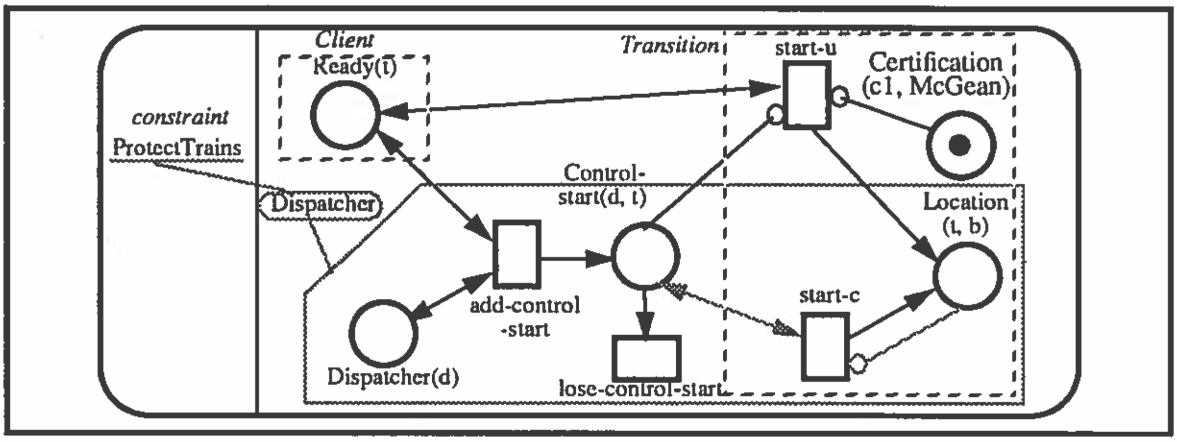
Have we now foiled the D1 scenario? Only if we can count on start-c as always being chosen (i.e., that control is always reliably created and never lost). But D2 leaves a loophole: if the critic can remove all Control-start(d,t) tokens then it can again produce something like scenario S1 using start-u. In at least one view, this can be seen as a reliability question: how likely is it that the controlling agent (dispatcher) will lose control (i.e., that lose-control-start will fire)? If the designer decides that this is a problem, then at least two modifications are possible: weaken the ProtectTrains constraint to allow an infrequent loss of control; lessen the odds of disaster by firing the add-control-start several times, generating more than one Control-start relation for each train, i.e., create back-up dispatchers for each train.

There is yet another reliability problem with the current specification state as seen in Figure B.3: there is no guarantee that control will be created "in time". That is, there is nothing preventing the sequence train-enters; start-u; add-dispatch-control. In other words, a train enters before a dispatcher can be assigned.

The McGean design does not 1) address either of these reliability problems, or 2) suggest that the

ProtectTrains constraint was weakened. Hence, we will allow the designer to certify that uncontrolled entry (start-u) is impossible. Our model provides a meta-operator for such certification, one that adds an explicit record of the designer's claim. The result is shown in Figure B.4. The certification token, c1, can be paraphrased as "all scenarios involving uncontrolled train arrival have been considered and dismissed as either implausible or infeasible to avoid".

Figure B.4 Application of certification operator to uncontrolled start.



The certification operator can clearly be misused: the designer could eliminate all nasty behavior from the net by erroneously certifying that it is implausible or too costly to deal with. While we expect that a certification token will include data to back it up, there is no way to formally check this. Hence, the reference to "McGean" in Figure B.4 could as just as easily have been "because I said so."

In our experience, we do not run into such misuse. In particular, certification is used almost exclusively when our tools lack the knowledge to reason formally about the design. It provides two crucial pieces of information: 1) it records that a negative outcome has been considered, and 2) it often points to a body of relevant domain knowledge that we currently lack a formal representation for. As with the earlier Glitter system, we maintain a list of certification tokens generated during design as a type of demand-driven guide to useful extensions to our model and tools

B.3 A split of responsibility

The designer again asks OPIE if the ProtectTrains constraint is satisfied:

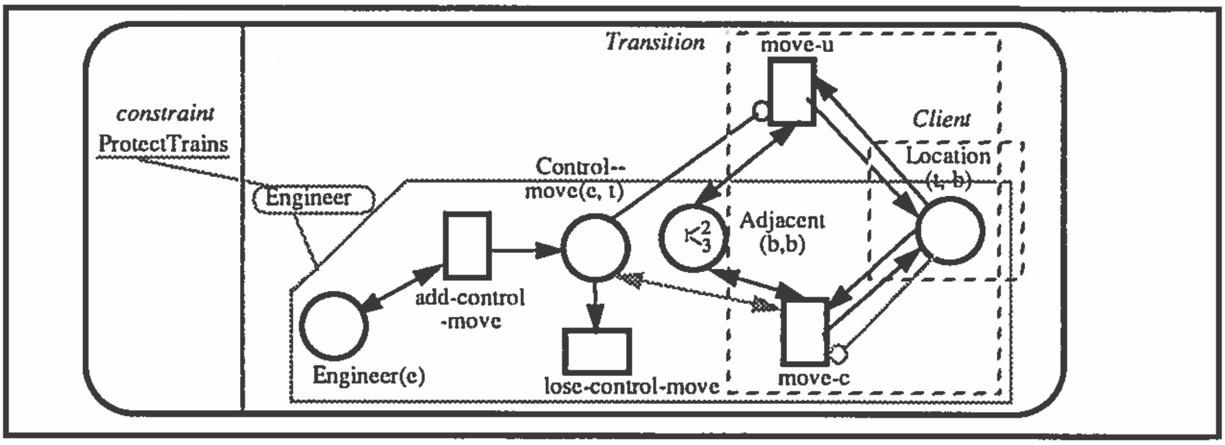
(D): Verify ProtectTrains in D2 given Block(b1) and Block(b2) & adjacent(b1, b2)

OPIE generates a scenario in which two trains enter the system safely, but still end up at the same location. Instead of crashing on start, they crash by moving into the same intermediate block.

(O): The constraint ProtectTrains is violated by scenario S2:
1. given Block (b1);
2. given Block (b2);
3. given Adjacent(b1, b2);
4. enter(t1|Train) produces Ready(t1);
5. create-dispatcher(d1|Dispatcher) producesDispatcher(d1);
6. add-dispatch-control(d1, t1) produces Control-start(d1, t1);
7. start-c(t1, b1) produces Location(t1, b1);
8. move(Location(t1, b1), b2) & Adjacent(b1, b2) produces Location(t1, b2);
9. enter(t2|Train) produces Ready(t2);
10. create-dispatcher(d2|Dispatcher) produces Dispatcher(d2);
11. add-dispatch-control(d2, t2) produces Control-start(d2, t2);
12. start-c(t2, b1) produces Location(t2, b1);
13. move(Location(t2, b1), b2) & Adjacent(b1, b2) produces Location(t2, b2);
Violation: ProtectTrains in D2

The step the designer takes to counter the S2 scenario is similar to that taken to counter S1: the brinkmanship operator is applied. Figure B.5 shows the result of this development. To paraphrase, the brinkmanship operator introduces a new move-c transition that can only occur when (1) an engineer is controlling the train, and (2) the preconditions of brinkmanship are met. As before, the new system retains a move-u transition that can fire for uncontrolled trains. The combination of the brinkmanship applications to the dispatcher and the engineer gives us a *sequential split* of responsibility, a common joint problem solving approach: break the problem into pieces (temporal in this case) and assign separate agents to each piece.

Figure B.5 System state D3 (excerpt): Application of brinkmanship to train movement



As with dispatch, we have reliability problems to address in Figure B.5. Can a train move before an engineer is assigned? Perhaps more realistically in the transportation domain, can an engineer (driver, pilot) be lost/incapacitated? If either question is a concern then we may need to introduce a fail-safe mechanism to *disable* uncontrolled movement (as in the "dead man's switch" on power lawnmowers, trains), or an understudy mechanism to *recover* from it (such as "co-pilots" on large ships, planes). Adding more sophistication, we may decide to consider an auto-pilot mechanism that allows temporary but anticipated loss of control.

Since no discernible design effort was made to handle these reliability issues in the McGean design, we will again allow the designer to certify that it is not a concern by applying the certification operator to block move-u (not shown in Figure B.5).

B.4 Deriving agent interface requirements.

As can be seen in Figure B.5, the brinkmanship operator has introduced deferred preconditions (grey arcs) between the move-c transition and the Location place, and between the move-c transition and the Control-move place. When faced with such arcs in the earlier dispatcher case, the designer simply blackened them, stating that it was expected that a dispatcher had direct access to the vacancy of entry blocks. To blacken the grey arcs here requires answers to the following questions:

1. Can the engineer directly control the move-c transition?
2. Does the engineer know the train's current location?
3. Does the engineer know if the track adjacent is empty (required by the not-arc on move-c in Figure B.5)?

The designer decides that engineer agents will have direct access to both the location of the train they control, and to the "throttle" itself, and blackens the appropriate arcs. While this seems like common sense knowledge if one uses on-board humans as engineer agents, one can imagine vehicles that are controlled remotely by an engineer (e.g., unmanned spacecraft), that require sophisticated two-way communication devices to bring about control.

The last question asks if engineers can "see" into adjacent blocks? This was not realistic in McGean's domain: blocks were 5 to 15 miles long and engineers could not directly sense the entire length of blocks adjacent to them. Fortunately, a standard solution is available: specify another agent to act as middleman, and gather the vacancy information for the engineer and pass it along to her. To select this joint problem-solving protocol, the designer will apply an operator we call set/reset that is defined to handle just such problems[12].

Figure B.6 shows the effects of this decision on the design state. As with other operators we have seen, the application of set/reset generates further design questions:

1. Can the designer guarantee that each block will be controlled at all times, or at least, whenever it is occupied?
2. What agents will control the various actions?

The designer certifies the first question, but deliberates more carefully on the second. The set-reset operator called for *two* disjoint sets of agents (A1 to control "reset" and A2 to control "set") to cover blocks, but the designer decides that the same agent that manages "set" (sets the home signal) can also manage "reset" (clear the signal), and applies a merge-agent operator to agent classes. While this is clearly more economical (and follows the McGean design), it also introduces a risk. In a particular implementation of the agent class (now called "Operator"), an agent may not have sufficient time to carry out both of its responsibilities (set and reset) for all of the blocks it is tracking -- it may be *overloaded*. In the McGean design, this is partially addressed by assigning only one block to each Operator. Our designer will likewise add this capacity constraint here.

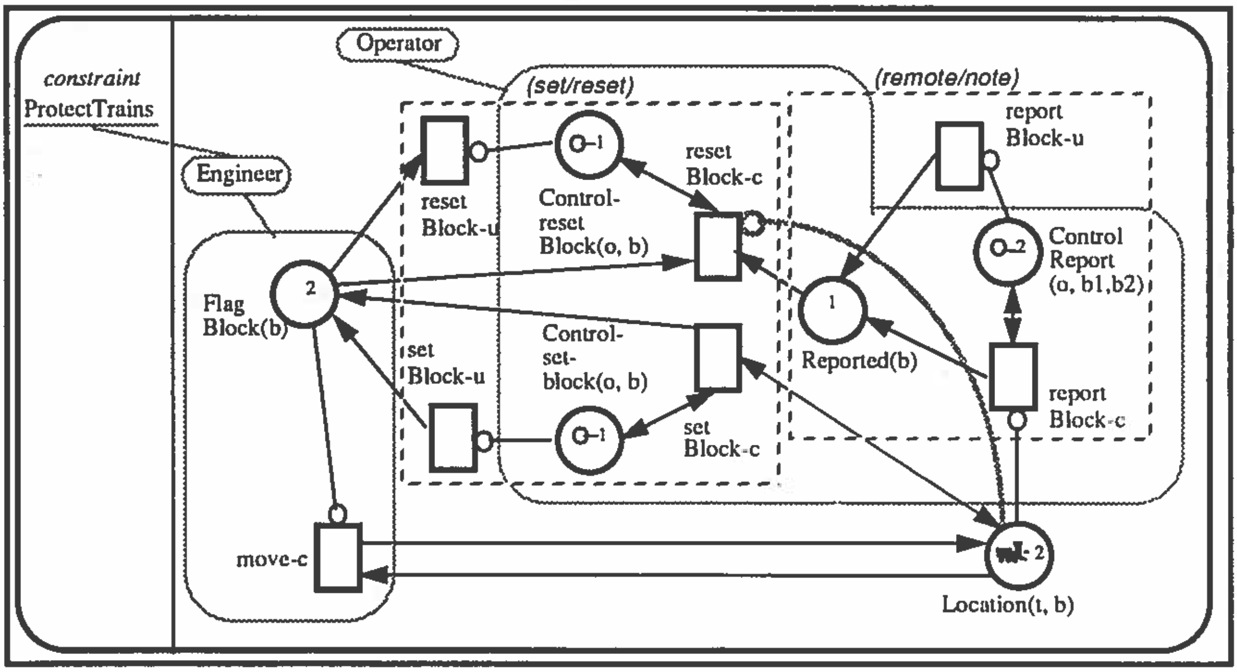
[12] The set/reset operator is modeled after the communication and synchronization primitives typically found in concurrent programming languages and operating systems, distributed and otherwise [Maekawa et al.,1987].

In addition to reliability and interference questions, installing the set/reset operator raises access issues:

3. Can each agent sense whether a block it controls is occupied? Can it sense when the block is clear?

The designer decides that the operator can directly sense that a train is in its block, simply by seeing the train pass by the station. The second access issue is represented by the virtual arc entering the "reset-Block-c" transition in Figure B.6. As with engineers, an operator would have to see into the adjacent block (to see a train leaving the operator's block) in order to determine whether the block it controls is, in fact, empty. This is not practical given the length of blocks. The solution adopted here is to replace the virtual arc by application of an operator called "remote/note" (on right of Figure B.6). This operator assigns an agent to monitor an entity, and report when the entity changes from one state to another. The recipient consumes ("notes") the report when it acts on it. In this context, the reporting agent (A3) generates a report when a train enters the block it is monitoring (and hence, leaves the adjacent block, making it vacant). An operator that receives a clear event then resets the signal for its block.

Figure B.6 System state D4 (excerpt) with set/reset and remote/note operator applications



As with set/reset, the report/note operator raises issues of control, reliability, and interference. The designer dismisses all except the interference issue: as with set/reset, the designer merges the functions of the reporting agent A3 with those of the operator (merged from A1 and A2). In addition to monitoring their "own" blocks, operators now will monitor the blocks from which trains arrive. When a train enters a block, that block's operator will notify the previous operator that her block is now clear. The previous operator will receive the report and reset her signal.

Section 3.2 summarizes the remainder of the development from this point.