Open access • Journal Article • DOI:10.1109/32.142870

Knowledge representation and reasoning in the design of composite systems

— Source link

Stephen Fickas, B.R. Helm

Institutions: University of Oregon

Published on: 01 Jun 1992 - IEEE Transactions on Software Engineering (IEEE)

Topics: Design process, High-level design, Iterative design, Design education and Computer-automated design

Related papers:

- Goal-directed requirements acquisition
- Language support for the specification and development of composite systems
- Representing and using nonfunctional requirements: a process-oriented approach
- A proposed perspective shift: viewing specification design as a planning problem
- Four dark corners of requirements engineering

Share this paper:

View more about this paper here: https://typeset.io/papers/knowledge-representation-and-reasoning-in-the-design-of-2o9ugkfddw

Knowledge Representation and Reasoning in the Design of Composite Systems

Stephen Fickas and B. Robert Helm

Abstract— **Our interest is in the design process that spans the gap between the requirements acquisition process and the implementation process, in which the basic architecture of a system is defined, and functions are allocated to software, hardware, and human agents. We call this process composite system design. Our goal is an interactive model of composite system design incorporating deficiency-driven design, formal analysis, incremental design and rationalization, and design reuse. We discuss knowledge representations and reasoning techniques for the product (composite system) that we are designing, and for the design process, which support these goals. To evaluate our model, we report on its use to rationally reconstruct the design of two existing composite systems.**

Index Terms—**Automated analysis, composite systems, knowledge-based design, rational reconstruction, software specification.**

I. Introduction

OUR group's historical interest has been in knowledge-based software development (AI and SE). However, in the course of our research, we found that we could not formally explain or reproduce the features of standard software designs, such as those of elevator controllers and library databases [24] 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 in the domains we studied [16]; they focused on policies and concerns that 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 [13]. 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 an interactive design model, called Critter, for producing composite systems. The model is founded on an earlier design tool called Glitter [15], which used automated problem-solving techniques to assist a human in implementing a formal specification. 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 [24]), and some containing software components solely (e.g., email transport systems). Our work on Critter is tied to the special issue as follows.

Software engineering goal: We wish to address the stage between the requirements acquisition process and the implementation process, in which the basic architecture of a system is defined, and functions are allocated to software and hardware components, or to the users. For the design process, in particular, we are interested in representations that promote deficiency-driven design, formal design analysis, incremental design and rationalization, and design reuse.

Knowledge content: There are two types of knowledge we must address in designing composite systems: 1) the knowledge of the artifact (composite system) that we are designing, and 2) the knowledge we use in designing the artifact. Our knowledge of an artifact is a formal model of its behavior and the constraints it lives under. For design knowledge, we have focused on the knowledge necessary 1) to decompose a global specification into specifications of its components, and 2) to modify the global specification so that realistic implementations can be found. We introduce the notion of a minimally restricted design step as a means to effectively play out a complex design. We also discuss the form of analysis our tools produce. In particular, we want our analysis tools to output not only "correct design" or "incorrect design," but the plans/counterexamples/scenarios that led to such a judgement. Our goal is to use this information in formulating the next step in the design process.

Knowledge representation and reasoning: We are most concerned with representing the design process. In contrast, for representing the artifact (i.e., the design state) we have chosen off-the-shelf formal languages (a form of temporal logic to represent design constraints, and a form of Petri net to represent artifact behavior). Our representation of the design process is based on a traditional AI paradigm, that of state-based search. It has two parts: 1) a representation of the design operators, heuristics, and analysis tools necessary for a design (search) to incrementally progress, and 2) a representation of the search manager, the component that keeps track of alternative search states and paths, allows browsing, exploration, etc. Our focus is on the first component, which Section III discusses in more detail.

We will argue for the strength of our knowledge content and knowledge representation choices through two examples in Sections IV and V. We will discuss the weaknesses of our choices, in the context of these two examples and other larger

Manuscript received October 1, 1991; revised February 11, 1992. Recommended by M. Jarke and A. Borgida. This work was supported by the National Science Foundation under Grant CCR-8804085.

The authors are with the Department of Computer Science, University of Oregon, Eugene, OR 97403-1202.

IEEE Log Number 9200172.

0162-8828/92$03.00 © 1992 IEEE

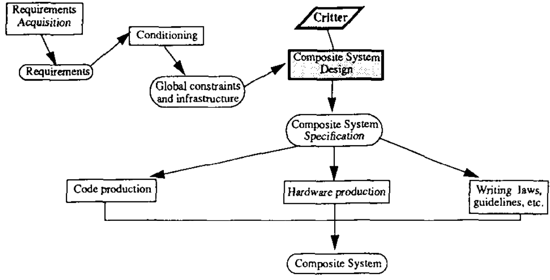


Fig. 1. Critter and its associated development processes.

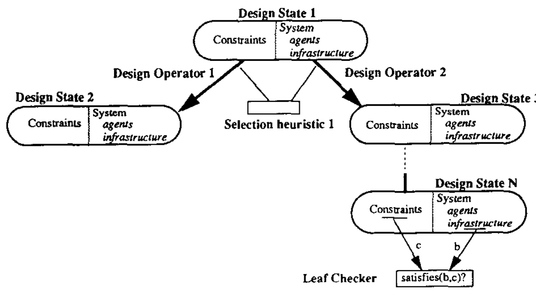


Fig. 2. Critter state-based search model of composite system design.

examples we have begun, in Section VI. Section VII discusses work related to the Critter model.

II. Our Place In The Lifecycle

Fig. 1 places Critter in the more general system lifecycle we envision.

To briefly summarize the figure, the requirements acquisition process involves analysts acquiring 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.

We have included a conditioning process in Fig. 1 to map the requirements produced by requirements acquisition tools to a form acceptable by our design tools. Clearly, this conditioning 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 organization to organization.

The composite system design process, the center of our research interest, decomposes formally specified global constraints and infrastructure into a specification of a composite system, a set of components or agents which interact to satisfy the global constraints. For example, the input to composite system design might be a formal version of the following:

Infrastructure = Internet
Constraint 1 = given an email message of form M, deliver M to its recipient
Constraint 2 = given an email message of form M, notify the sender when it is delivered
. . . .

The output of the composite system design process would be the specification of a set of email agents running on the Internet that took responsibility for satisfying the set of constraints. The informal SMTP specification (the standard email system on the Internet) is an excellent real life example of what we are trying to produce formally. The focus of this paper is on a tool, Critter, that supports the interactive design of such systems.

The bottom three processes implement each agent class according to the composite system specification. A single composite system specification may require that each implementation process be applied: producing software, acquiring or manufacturing hardware, and writing legal statutes or training manuals which prescribe the actions of human workers taking the role of an agent. The first example we describe, that of a train management system, has forced us to look, at least superficially, at the hardware and rule-writing processes. Our second example, that of an email transport system, is principally composed of software.

The next section introduces composite system design in more detail, and the Critter tools that support it.

III. The Critter Model

The Critter design model (Fig. 2) helps a human designer develop a composite system design using the paradigm of state-based search. Starting from an initial problem state, Critter helps an analyst apply design operators until he 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.

We discuss the knowledge content, representation, and reasoning tools of each of these components in turn.

3.1. Design States

Each design state in our search space represents a single composite system design. A state in Critter has two components:

1) 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.
2) 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 into each other."

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.

The problems that we have focused on can be characterized as liveness-safety problems [28]. Typically, they involve constraints of two general forms.

Liveness: For each behavior B, find some state p in B s.t. p satisfies $C_{achievement}$.

Safety: For each behavior B, no state p in B should satisfy C_{unsafe}.

In toy worlds we may be able to drop one class of 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. An agent class in our representation thus 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.

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. 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 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 [12], 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 [10], and is very similar to the "temporal-causal logic" independently developed by Castro [5]. It allows expression of both safety and strong liveness requirements [28].

The language for the system portion can be viewed, alternatively, as a subset of Gist [30] or a superset of Numerical Petri Nets (NPN) [46] that adds the notion of agents. We will use the Petri net view in this paper for presentation purposes.

3.2. Leaf Node Checkers

The overall goal of the Critter search is to bring the system and constraint portions into "consistency" in a single design state. A consistent state is called a "leaf" or "solution" state. Our model semi-automatically checks design states for consistency (leafhood) using three tools.

Analysis Tool 1: A planner or scenario generator called OPIE [2]. 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 nonleaf state to transform it into a solution.

Analysis Tool 2: An NPN simulator. We have implemented an NPN simulator that "runs" an NPN forward. At nondeterministic 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 to show that a safety constraint is not satisfied in the current state. The details of this tool, and a semi-automated means of selecting appropriate test cases to feed it, is discussed in [16].

Analysis Tool 3: 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 [23] to represent infinite plans/behaviors. As with OPIE, these queries can be used to provide existence proofs.

3.3. Search Operators

Search operators (henceforth, design operators) are what transform one composite design state into another, and may apply to either the constraint or the system part of the state. They are applied to a nonleaf node N to remedy a deficiency found in N. Hence, we loop through the se-

quence analysis⇒deficiency⇒remedy in Critter until the analysis tools report success.

Because our interest is in composite system problems, our design operators are tailored to multi-agent distributed-action concerns. Among the types of design operators we use are the following.

- Introduction of agent classes. There are operators for introducing new types of agents, or for splitting existing types. For example, an operator might split a class of "protocol entities" into "clients" and "servers."
- Assignment and merging of responsibility. As described above, agents can be given responsibility for constraints. They can share their responsibilities with one or more other agents. They can also be overloaded; each agent can have multiple responsibilities.
- Communication and synchronization among agents. Critter includes operators which introduce simple communication and synchronization protocols (such as request–reply interactions) between agents.
- Weakening of constraints. 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. In these cases, a constraint may be weakened to make illegal behaviors legal ones.

3.4. Heuristic Evaluation

There is a relatively small number of design operators that are able to produce the composite systems 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. From our study of composite system design heuristics [14],[17], we suspect that the human designer will continue to be the main source of heuristic evaluation in Critter for the foreseeable future.

3.5. Implementation

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 [7] that provides for separate development states. Each development state represents the current design state and the current set of design issues.

Design 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 using a Petri net editor. The user also must attach any selection heuristics as arguments to the positions. Critter then automatically generates a new development state (a new IBIS state) after the operator has been applied. In this way, the entire design space is maintained, allowing a designer to move to any node and explore alternatives. While there are interesting issues on the use of an IBIS style framework for implementing incremental design tools [18], in this paper we will focus on the more general model of composite system design.

IV. Applying The Critter Model: The McGean Design

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 that have evolved over time and that 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 automate the now manual heuristic evaluation of designs. We have used the Critter model to rationally reconstruct several designs, two of which we will summarize in the remainder of the paper. The success criterion for our rational reconstructions is the ability to model the original infrastructure and reproduce the final responsibility assignments through the use of our design operators.

A word of motivation on our choice of the two study problems is in order. The first example we present in this paper is a train management system described in McGean [32]. The system, known as "manual absolute block clearing," originated in the 19th century; we will refer to it as the "McGean design" for brevity.

We arrived at this problem after work on the canonical elevator problem [12],[24], which in turn derives from [25]. 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 [32], including a history of major train accidents and their causes [42].
- A train system is a more interesting composite system problem. There are more agents involved and more ways to split responsibility among them.
- This same domain has been studied for its connection to network protocols by [22]. This leads cleanly into our second example.

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

- 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.

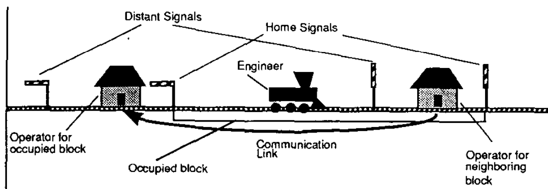


Fig. 3. Simplified McGean design.

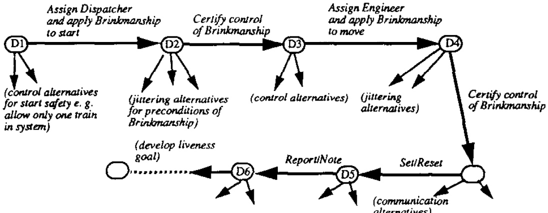


Fig. 4. Excerpt of the McGean example design space.

- MOTIS, and network applications of its ilk, have well-studied and formal evaluation criteria [44]. Many network applications also have a written chronology of their design (see, for instance, the RFC's maintained by the Internet NIC).

Taking up the McGean design first, Fig. 3 shows the target of our design process. This design attempts to prevent collisions of trains by allocating regions of track of 5- to 15-mi lengths (called blocks) to a single train at a time. Looking at the final solution from a composite system view, the design designates Engineer agents as responsible for entering a block between stations only if the signal for that block reads clear (vertical). Station Operator agents are given the responsibility of 1) setting the distant and home signals for a block to "occupied" (horizontal) when a train enters a block, 2) resetting the signal to clear when notified that a train has left a block, and 3) notifying other station operators when a block becomes clear. Finally, Dispatcher agents (not shown in Fig. 3) are responsible for making sure trains enter a system in a safe fashion.

Fig. 4 summarizes the first part of the path Critter takes to arrive at the target design in Fig. 3; it roughly represents the IBIS form of search tree we maintain as part of Critter. As in Fig. 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 that interact to satisfy the constraints.

4.1. The Initial State

Critter's search process requires an initial design state. We rely on the analyst to produce this initial state for the McGean design, guided by some considerations we describe in this section. This section also introduces the notation we use for design states throughout the paper.

The initial state of the train example is shown in Fig. 5.

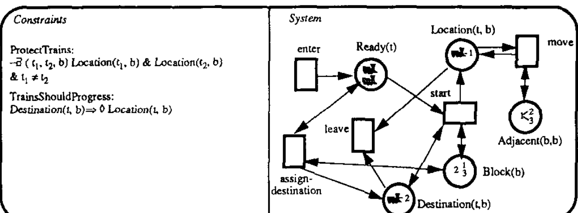


Fig. 5. Initial state $D1$ of the McGean example.

We use an enhanced version of an NPN [46] to represent both infrastructure and agents of a system. The initial constraint portion appears at left, the system portion at right. 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). We can paraphrase the specification of the system portion in this figure as follows.

- Trains can be created. The transition (drawn as a box) enter in the upper left corner can fire nondeterministically, since it has no input arcs; when it fires, it introduces a train token t (drawn as a train icon) on the place (drawn as a circle) labeled $\mathrm{Ready}(t)$ at the end of the arrow. The Readyplace denotes the relation "Train t is ready to start."
- A train t for which $\mathrm{Ready}(t)$ holds can be assigned a destination block by the $\mathrm{assign}-\mathrm{destination}$ transition. The arc between the transition$\mathrm{assign}-\mathrm{destination}$ and $\mathrm{Ready}(t)$ is doubled-headed. This means that the transition needs a token (i.e., a train t) from $\mathrm{Ready}(t)$ to fire, but the token is simply replaced in $\mathrm{Ready}(t)$, unmodified, on firing. In contrast, the move transition changes a train's location; hence, we have drawn two separate arcs between move and $\mathrm{Location}(t, b)$.
- If a train t is Ready, has a Destination, and has an available Block to start in (shown as blocks 1,2,3 in the Block place), it can start (transition start can fire). Starting puts t at a location block b, denoted by the place $\mathrm{Location}(t, b)$.
- Trains for which $\mathrm{Location}(t, b)$ holds can move to Adjacent blocks (shown as block numbers connected by lines) 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.

The initial state $D1$ 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 [42], 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 later refer to as brownfield constraints). 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 is the output of a requirements acquisition process, possibly aided by a system such as the KAOS [45] assistant discussed in Section VII.

4.2. Deficiency-Driven Composite Design

Our general style of design is deficiency-driven. We use the automated analysis tools of Critter to identify deficiencies, i.e., behaviors of the system that violate the constraints. We then apply design operators to overcome those deficiencies. This section illustrates this process and our representations in more detail.

The designer uses the planner-based leaf-checker OPIE (described in Section 3.1) in state $D1$ to see whether the safety constraint is met. OPIE cannot verify systems written in the full NPN language, but only a subset that is roughly equivalent to the language of STRIPS [2]. We rely on the designer to restrict the system to this subset.

OPIE returns a plan in which two trains collide at start-up.

> The constraint ProtectTrains is violated by scenario $S1$ in state $D1$:
> 1. given $\mathrm{Block}(b1)$;
> 2. $\mathrm{enter}(t1|\mathrm{Train})$ produces $\mathrm{Ready}(t1)$;
> 3. $\mathrm{enter}(t2|\mathrm{Train})$ produces $\mathrm{Ready}(t2)$;
> 4. $\mathrm{assign}-\mathrm{destination}(t1, b1)$ produces $\mathrm{Destination}(t1, b1)$
> 5. $\mathrm{assign}-\mathrm{destination}(t2, b1)$ produces $\mathrm{Destination}(t2, b1)$
> 6. $\mathrm{start}(t1, b1)$ produces $\mathrm{Location}(t1, b1)$;
> 7. $\mathrm{start}(t2, b1)$ produces $\mathrm{Location}(t2, b1)$;
>
> Violation: ProtectTrains in $D1$

To eliminate a negative scenario such as $S1$, the designer can choose among three complementary strategies.

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 $S1$.
2) Weaken the safety constraint. Some train systems take this approach by allowing more than one train in a block [42].
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 option. Critter has several operators that could accomplish this.

1) Never allow a specific condition in the constraint to be true (e.g., never allow trains in the system).
2) Ensure that two conditions in the constraint are mutually exclusive (never allowing more than one train in the system).
3) Manipulate transitions so that some but not all the constraint conjuncts are allowed to become true at the same time.

The designer chooses 3), whose general class we call brinkmanship. The set of brinkmanship operators modify the system so that an agent assigned to the constraint prevents transitions that are the last step in breaking the constraint. The outcome of this in our example is that a designated agent will never allow a train to start at the same location as another train.

Before we describe the brinkmanship operator in detail, we describe Critter design operators in general. Design operators take a cooperative form.

- The operator contains the framework of a composite system design strategy, for instance, the means of dividing responsibility among agents in a particular setting. This is represented by a matching pattern that establishes the right setting and a replacement pattern that introduces the concept into the design. A design operator may also introduce domain assumptions and failure modes. These are discussed in more detail in the next section.
- The human designer 1) supplies important pieces that fill out the matching and replacement patterns, 2) either confirms each assumption the operator makes, or decides that an assumption is uncertain enough to call for further remedial design, and 3) attempts to mitigate any troublesome behavior introduced by the operator. All of these actions typically require the designer to supply domain-specific knowledge to the design process.

In summary, the designer uses her knowledge of the domain to select operators and guide their application to the current state. The operators add the necessary components to the system description and track open assumptions for the designer. In this way, our design operators are much like the skeleton plans of the MOLGEN system that addressed the strategic aspects of a design, relying on the user to supply the domain-specific details and fix the plan when problems were discovered [19].

The particular brinkmanship operator we will employ matches on constraints of the form

$$\neg\exists(x, y, z)\ P(x, z)\ \wedge\ P(y, z)\ \wedge\ x \neq y$$

where P is a relation (or place in NPN form), x, y, z are vectors, and there is an implicit "for all behaviors" qualifier on the front. This pattern, along with the agent A assigned to the constraint, is shown as part of the brinkmanship operator in Fig. 6. The result of applying the operator is a new system where a "brink transition" (T in the figure) is controlled by the agent A.

The application of the brinkmanship operator is interactive.

1) The designer factors the constraint into a "controlled" condition and a "brink" condition. In our example, P is the Location relation, and the controlled and brink conditions are both partial instantiations of P.
2) The designer chooses an agent class A to be responsible for the factored constraint. In this case, the designer chooses a train Dispatcher agent to be the responsible, hence controlling, agent.
3) The designer identifies a transition T that can produce the controlled condition when the brink condition is true, violating the constraint. The agent responsible for the constraint will control T so that it cannot fire when the brink condition is true, preventing T from pushing the system over the "brink" by producing $P(x, z) \wedge P(y, z)$. In our example T is the start transition.

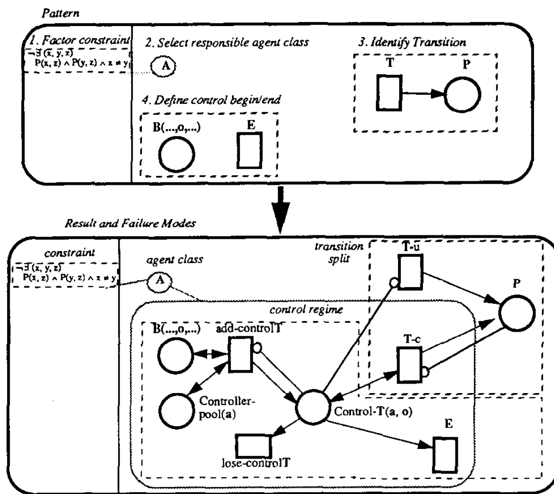


Fig. 6. Brinkmanship operator.

4) The designer defines how control begins and ends. The designer must identify some place that triggers an agent in A to take control. The designer also identifies a condition under which that agent can release control. In our example, a Dispatcher agent takes control of a train $t1$ when $\text{Ready}(t1)$ is true. The dispatcher releases control of $t1$ when start fires.

As can be seen in Fig. 6, the result of applying the operator is that the T in the top pattern is split into two parts, one that is controlled $(T - c)$ and one that is uncontrolled $(T - u)$. The controlled transition $T - c$ has a new piece of subnet associated with it that allocates (and potentially loses) control. $T - c$ also has a not-arc (represented as a line ending in a circle) attached to it from the place P. This represents the brink check: if the brink condition exists already (e.g., there is another train at the same starting location), then block any firing of $T-c$ that would cause the control condition to become true (e.g., a second train would end up at the same starting location).

Fig. 7 shows the parts of the McGean design state affected by the brinkmanship operator. Places and transitions drawn with short, nondirected arcs designate elision; we have left off either incoming or outgoing arcs that have no bearing on the current figure. We can paraphrase Fig. 7 as follows.

> Trains ready to start are assigned a dispatcher. More than one dispatcher can be assigned to control the same train, either simultaneously or after loss of control. However, a single train cannot be under the control of the same dispatcher more than once simultaneously.
>
> Trains under the control of at least one dispatcher do not start until their starting block is clear. If no dispatcher is in control of a train then it is possible for the train to start unrestricted via the transition $\text{start}-u$ (including cases where a train starts before control can be assigned).

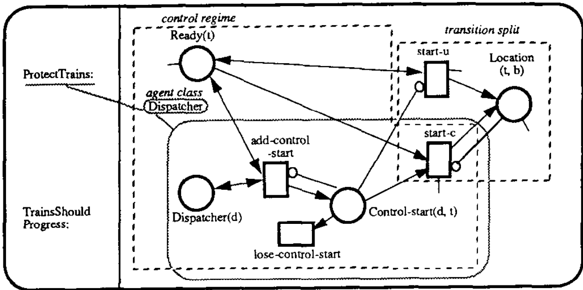


Fig. 7. System state $D2$ (excerpt): Application of brinkmanship operator to start transition.

The brinkmanship operator has also introduced a set of domain assumptions (not shown in the figures), and has added two failure mode transitions ($\text{start}-u$ and $\text{lose}-\text{control}-\text{start}$). Dealing with these is the next step in the McGean development.

4.3. Validating Operator Application: Dismissal and Certification

Critter design operators do not guarantee a provably correct design by themselves. This is a major and conscious departure from the work in formal transformation systems. Instead of concentrating on tightly restricted correctness-preserving operators, we have focused on general operators that incorporate knowledge to validate their use in a particular problem setting. The next two sections illustrate and defend this strategy.

Critter attempts to validate the changes an operator makes to the specification by two means. First, design operators include domain assumptions, conditions that must hold in the design domain for the operator to be effective. When an operator is applied, Critter records its domain assumptions as open issues. For instance, in Fig. 7, the domain assumptions of brinkmanship are that dispatchers can be found, that they can gain control of a train, that they have direct access to the start transition, and that they can see or sense that a starting location is clear or occupied. Each of these assumptions will remain open until actually validated by the designer.

Second, design operators include failure modes, representing behaviors that have been found to cause trouble for the systems where the operator was applied. For example, the $\text{lose} - \text{control} - \text{start}$ and $\text{start}-u$ transitions of Fig. 7 represent undesirable behavior which may occur if the control regime for starting trains fails.

The designer can address open domain assumptions by dismissing them, or by adjusting the composite system to satisfy the assumptions. The McGean design we are reproducing assumes that all of the new control components (transitions, arcs, and places) added by the brinkmanship operator are valid. We assume the designer certifies them as such (not shown in Fig. 7).

Similarly, the designer can address failure modes in a design by certifying they will not occur, or by redesigning the system to tolerate them. The McGean design we are reproducing does

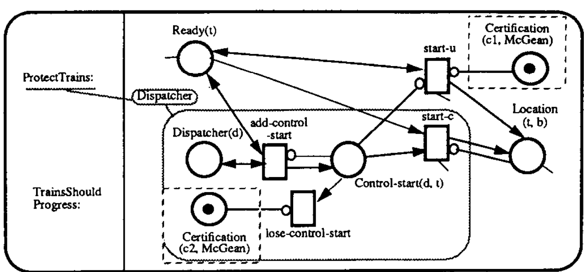


Fig. 8. Design state $D3$: Application of certification operator.

not address either the lose − control − start or the start−u transition, i.e., there are no new agents in the target design that are introduced to overcome either problem. Hence, we will allow the designer to certify that uncontrolled entry (start−u) and loss of control (lose-control-start) can be ignored from a composite system design viewpoint.

The result, design state $D3$, is shown in Fig. 8. The certification token, $c1$, can be paraphrased as "all scenarios involving uncontrolled train arrival have been considered and dismissed as either preventable at the implementation level or unlikely to occur in the real world." The certification token $c2$ works in a similar fashion for loss of control. The certification tokens provide a hook where the designer can insert data backing their claims, such as standards, or court cases setting limits on what negative impact must be considered in designing artifacts. If these standardsf later become false, or if we want to experiment in a hypothetical world where they are false, the certification tokens can be removed and the corresponding transitions will become unblocked.

In general, dismissal and certification act as an escape when our tools lack the knowledge to reason formally about the subsequent implementation process or the domain in which the system will reside. They provide two crucial pieces of information: 1) they record that a negative outcome has been considered, and 2) they often point to a body of relevant domain knowledge for which we currently lack a formal representation.

The previous step also illustrates the philosophy underlying Critters design operators. Applying an operator got us a little bit further toward a "safe" design, but was not provably correct, even with respect to the local goal for which it was applied. We have de-emphasized provably correct design in Critter for several reasons. First, provably correct operators would have a lengthy set of operator preconditions that a) might require a large up-front theorem proving effort, and b) would narrow operator application to a small class of settings, and hence, require a large subgoaling or jittering effort before the operator is ever attempted. In our experience with Glitter, both of these problems led to much wasted design effort—it wasn't until an operator/transformation had actually put a concept in place that it could adequately be judged. Second, validation of an operator frequently relies on domain-specific knowledge unavailable to Critter. Again, we prefer to postpone this validation process until after the operator has put the concept in place within the system.

In summary, our approach is to use a minimal set of preconditions to get a general design strategy (as represented by an operator) into play, including stereotypical issues and bugs associated with the strategy. We then use analysis tools and the human designer to point to places where further design is necessary. In essence, we promote a style of design that allows a designer to "test drive" a strategy before committing enormous energy into making it work at a detailed level.

4.4. Reuse of Operators: Splitting Control

Critters operators are intended to provide reusable strategies for composite design. The next step of the McGean rationalization suggests how reuse occurs within a particular design problem, and how an incremental approach can rationalize features of a multi-agent design.

The design state $D3$ includes an agent class, Dispatcher, which will ensure that trains do not violate the safety constraint at start-up. However, design state $D3$ is not a leaf node—OPIE generates a scenario $S2$ 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 (firing the move transition in Fig. 5) into the same block.

The steps the designer takes to counter the $S2$ scenario are similar to those taken to counter $S1$: the same brinkmanship operator is applied (Fig. 9). As with start−c, the new move−c transition can only occur when an agent (Engineer) is controlling the train. As before, the new system retains both a move−u transition that can fire for uncontrolled trains, and a transition for losing control.

The combination of the brinkmanship applications to Dispatcher and Engineer gives us a sequential split of responsibility, a common cooperative problem solving approach: break the problem into pieces (temporal in this case) and assign separate agents to each piece. However, this division of labor was constructed incrementally, by a sequence of deficiency-driven steps. The design history thus rationalizes the sequential division of labor in terms of a specific goal (ProtectTrains), problematic scenarios ($S1$, $S2$) encountered during design, and the design steps taken to address them.

4.5. Validating Operator Application: Adjusting the Design

When the domain assumptions or failure modes of an operator cannot be certified away, the designer adjusts the design to address them. The next development sequence shows how this adjustment activity can naturally rationalize supporting services such as inter-agent communication protocols within a design.

Given the second application of brinkmanship, the designer is left with the task of verifying the domain assumptions as they now apply to engineers controlling the movement of trains.

1) Can the Engineer directly control train movement (move−c), i.e., does she have direct access to the "throttle"? While the answer obviously seems to be yes if one uses on-board humans as Engineer agents, vehicles such as unmanned spacecraft are controlled remotely by an engineer, and require sophisticated two-way communication devices to bring about control.

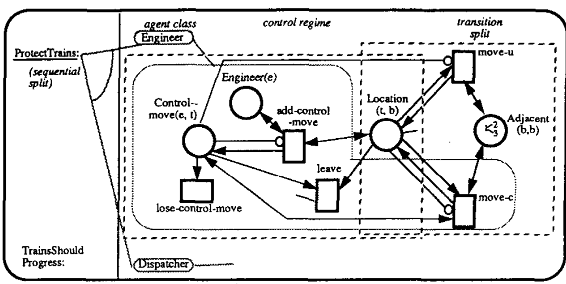


Fig. 9. System state $D4$ (excerpt): Application of brinkmanship to train movement.

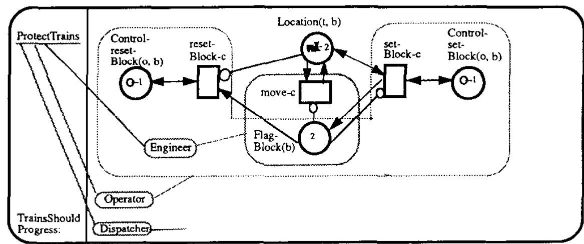


Fig. 10. System state $D5$ (excerpt)—Result of set/reset.

More down to earth, some recent transportation systems rely on cooperation between an onboard computer agent and a remote human agent to control vehicles.

2) Can the Engineer directly access whether an adjacent block is occupied? (Is the not-arc between move$-c$ and Location valid?) The sensing technology of the day was human vision.

As before, the Brinkmanship operator also introduces two potential failure modes.

1) Can uncontrolled movement (move$-u$) occur?
2) Can loss of control (lose $-$ control $-$ move) occur?

The history of train system design has shown that all of these questions has eventually required attention. However, the design we are reconstructing only addresses the second question (access to an adjacent block); hence, as with Dispatchers, we will mark the arc between Control $-$ move and move$-c$ as valid, answering question 1. We will also certify that loss of engineer control and uncontrolled train movement can be ignored, certifying "yes" to questions 3 and 4.

This leaves us with question 2—can Engineers "see" into adjacent blocks, i.e., is the not-arc between move$-c$ and Location valid? This was not realistic in McGeans domain: blocks were 5 to 15 mi long and Engineers could not directly sense the entire length of blocks adjacent to them. Fortunately, a standard composite system solution is available: specify other agents to act as intermediaries, manipulating a flag or "spin lock" that relays the vacancy information to the Engineer. To select this joint problem-solving protocol, the designer applies an operator from a class we call set/ reset. As with brinkmanship, the designer must guide the operator to the appropriate location in the current state, in this case, to the not-arc that blocks movement in Fig. 9. Fig. 10 shows the effects of this decision on the design state: the not-arc between move$-c$ and Location is replaced with a more elaborate mechanism for maintaining a flag on the block.

Besides *pinpointing* the location of application for the operator, the designer must specify which agent classes will control the actions of setting and resetting. Set/reset calls for two 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 $-$ Block$-c$ (sets the home signal) can also manage reset $-$ Block$-c$ (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—overloading may lead to overcommitment. In the McGean design, this is partially addressed by assigning only one block to each Operator.

Applying the set/reset operator raises its own domain assumptions and failure modes (not shown in the figure). Two of the domain assumptions are of particular interest here.

1) Can the controlling agent sense whether a block it controls is occupied? (Is the arc between set $-$ Block$-c$ and Location valid?)
2) Can the controlling agent sense when the block is clear? (Is the not-arc between Location and reset $-$ Block$-c$ valid?)

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 question is similar to one that surfaced with Engineers: an Operator would have to see into the adjacent block (to see a train leaving the Operators block) in order to determine whether the block it controls is, in fact, empty. Again, this is not practical given the length of blocks. The solution adopted here is to replace the not-arc between reset $-$ Block$-c$ and Location by the application of an operator from the class we call "report/note."

The result is shown in Fig. 10. With the guidance of the designer, a report/note operator splices in an agent that monitors a state, and reports when the enabling condition is true, in this case that no train is in the preceding block. The recipient consumes ("notes") the report when it acts on it. In Fig. 10, the recipient is the adjacent Operator, and she uses the report to reset the signal for her block.

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.

There are a number of remaining issues in state $D6$, generated by the application of set/reset and report/note. Can we certify that loss of control and uncontrolled actions (not shown in Fig. 10) can be ignored? Can we handle the race conditions between movement and setting/reporting? Can we control report $-$ Block so that it does not produce redundant

reports? Should we change the arc between report – Block and Location from a not-arc to a positive arc to better reflect the McGean protocol?

More design effort is needed to address these problems if we are to accurately reproduce the McGean design. However, no new Critter components are highlighted, so we will omit the remaining design steps necessary to complete development of the safety constraint.

Before leaving our development of the safety constraint, we note that we have highlighted OPIE as the tool to check for violations of this constraint. However, the NPN simulator tool is also available, and at times may be more effective. The NPN simulator allows the designer to guide its analysis. Interactive analysis can often quickly expose faults which would take OPIE much longer to discover, albeit automatically. However, the NPN simulator requires that we have a library of test cases for the train domain. The designer must help adapt these test cases to the current design and then guide the simulation. These issues are discussed in more detail in [16].

4.6. Liveness

We summarize the remainder of the McGean development, which is guided by violations of the liveness constraint. This section briefly indicates how such violations can be discovered. It also illustrates the precarious balance between safety and liveness in composite system design problems. In general, modern transportation designs trade off these two classes of constraints in complex ways that we do not address in the McGean design [37].

In the McGean design, the analyst uses the reachability analysis leaf-checker (RG) to disprove the liveness constraint. The RG tool, unlike OPIE, can generate an infinite counterexample in which C never becomes true, disproving a constraint that C eventually is satisfied. However, the RG tool generates what is effectively an exhaustive reachability graph for the system, whereas OPIE generates selected behaviors guided by goals. The RG tool also operates on a smaller subset of the NPN language than OPIE.

We can see one way in which liveness can fail by looking back at Fig. 9, in which Engineers were assigned control of trains after being started by Dispatchers. This hand-off error may lead to a scenario in which a train starts, but no Engineer is ever assigned to control it, thus preventing the train from moving to its destination. In essence, designing to meet the safety constraint has introduced a problem with liveness.

The designer could address this problem in a number of ways. She could patch the application of Brinkmanship so that Engineers are assigned to trains before the trains start; this would lead to a "harbor pilot" form of control in which two agents (a Dispatcher and an Engineer) are responsible for a vehicle to a certain point, at which point a single agent takes over. Instead, the designer keeps the current division of labor, which more closely resembles "valet parking"; one agent passes the vehicle to another.

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. This assumes that we can realistically certify away loss of control as we did in all design subsequent to this state. If any of these assumptions change, e.g., if we remove $c2$ in Fig. 8, not only might safety constraints be in jeopardy, but liveness constraints as well; only controlled trains can move in the McGean design.

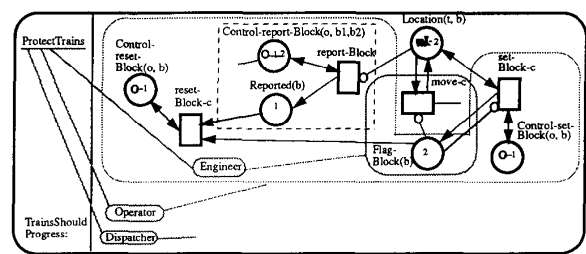


Fig. 11. System state $D6$ (excerpt)—Insertion of report/note.

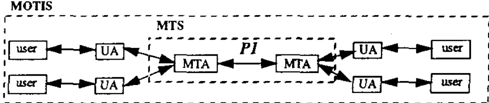


Fig. 12. MOTIS messaging system and P1 protocol.

V. Applying The Critter Model: The MOTIS Design

To suggest the generality of our 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 [44]. The MOTIS model is shown in Fig. 12.

The P1 flow control protocol, depicted in Fig. 13, 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).

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 doing away with engineers, and having station operators drive trains that are in their blocks.
2) The designer applies the brinkmanship operator to unfold the safety constraint onto each nodes "send" operator. The result, as in the McGean design, is that the message transfer agent that sends a message is required

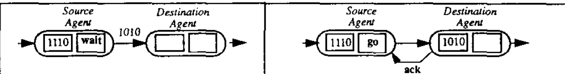


Fig. 13. MOTIS P1 flow control protocol.

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.

This completes our two rational reconstruction examples.

VI. Conclusions

We have argued for the benefits of the Critter model in the preceding sections. Returning to our original software engineering goals, Critter supports deficiency-driven design, formal design analysis, incremental design and rationalization, and design reuse. Design knowledge is split between the human designer and Critter: Critter supplies knowledge of composite system design strategies and concepts; the designer supplies domain specific knowledge to validate these strategies and concepts in a particular application domain.

Our focus in Critter has been to build an effective interactive design tool. This has led us to a representation of operators that enforce minimal applicability conditions. The human designer can quickly test an idea using Critter, and then if satisfied, commit further design effort to verify and validate the new design. We have also argued for incremental design as an effective means to control a complex design problem. Using this style, a problem is chipped at gradually until it is finally solved.

We have evaluated the tractability of Critter's reasoning techniques on a handful of rational reconstruction problems. While we have demonstrated the sufficiency of Critter on these problems, they also exposed limitations of Critters reasoning techniques that prevent its use on larger software engineering problems. In particular, our deficiency-driven style of design requires heavy use of the analysis tools to guide application of operators. Further, our minimally restricted design operators rely on analysis tools to point to places where cleanup design is necessary. Consequently, efficient use of analysis becomes critical. One way to use analysis more efficiently is to cache or reuse analysis results as a design proceeds. For example, rather than generating a scenario like $S2$ from scratch, an analysis tool might create it by adding a move step and changing objects in the previous scenario $S1$. In general, having analyzed deficiencies in state S_i, the system may reuse the S_i analysis in S_{i+1}, tempered by the (typically small) changes between the two states. For planners such as OPIE, "tempering" techniques have been developed from work on plan reuse and transformation [21], [43]. However, considerable research is required to apply these techniques to analysis reuse.

We have also evaluated the coverage of Critters design knowledge base, and the expressiveness of its artifact representation. We can produce the high-level architecture of simple composite systems with a reusable set of design operators. However, we have also tried to extend Critter to more challenging problems in the network application domain of the MOTIS problem. In the process, we have identified several classes of composite system design issues Critter does not yet address. For example:

- 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 network 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 functionality. When two agents need to communicate, they may need to negotiate to determine what capabilities they have in common beyond a minimal set. Our agent representation must be extended to provide agents with a crude internal model of their capabilities so they can participate in such negotiation protocols, as do peers in the TELNET [36] and the OSI Session Layer [38] protocols.
- Agent security and privacy. Our reconstructions of McGean and MOTIS did not force us to address the security and privacy issues of these domains [20]. In network applications, we frequently need to limit the abilities of agents to read information and reconfigure the system. Agent authorization and authentication become issues. In general, communication becomes a more complicated business than that portrayed in the two examples in this paper.
- Predefined agents. Our examples started with minimal infrastructure and no agents to build on. In the domain of networked applications, there are standard network services (in our terms, standard agents) provided in both the Internet and OSI domains. These domain-specific composite system building blocks need to be cataloged just as the domain-independent composite system design operators [27].
- Fault recovery. In the McGean design, as in safety-critical systems in general, we may put extraordinary effort into anticipating and designing out faults. However, in many domains it is impossible to anticipate and design out all faults. In these domains, one frequently designs mechanisms that detect and recover from unanticipated faults. Network application design is one of these domains. Thus, to represent devices managed by network management protocols such as SNMP [39], our agent representation must include hooks for remote monitoring and control.

To address these issues, we have to extend both our artifact representation, and the synthesis and analysis knowledge we apply to the design process. The research challenge is to incorporate this knowledge while retaining the advantages of Critter demonstrated in this paper.

VII. Related Work

Our work shares many of the goals of research in formal development of distributed systems, such as work reported in [3]. In general, we wish to support development of multiagent systems which satisfy globally specified constraints. At the same time, we note two contrasts between our approach and others.

The first contrast is in design methodology. A designer using our approach starts from a global statement of properties, and attempts to decompose these constraints into specifications of individual agents, guided by considerations of access to information and control. In this respect, our approach is similar to that advocated by [41]. Work in formal development of distributed systems, such as that of Abadi and Lamport [1], the "constructive" approach of Kramer, Magee, and Finkelstein [27], and the earlier work on CCS [33], frequently emphasizes a different style of design: the designer composes prespecified components or agents into a system, and verifies the system against global constraints. In addition, our composite system approach does not preserve correctness: global constraints may be dropped or changed during design, and design operators may not always solve the problems for which they were applied. The approaches of Abadi and Lamport, Kramer *et al.*, and CCS emphasize correct composition against a stable set of constraints. We see the two styles as complementary: our approach focuses on an initial phase where the components of a composite system are identified and functions allocated to them; work on correctness-preserving composition provides techniques for verifying that the behavior of the final composite system design is satisfactory.

The second contrast we see with formal distributed systems work is in research methodology. Many researchers in this area rely on problems with a pedigree within computer science to demonstrate and validate their formalisms and techniques. Thus, [4], [26], and [31] each use the "dining philosophers" problem in an example of their techniques. Well-known computer science problems like this make it easier to understand and compare competing design approaches and formalisms. On the other hand, we want to design complex systems of interacting human, hardware, and software agents; we cannot easily predict how useful a design approach will be based on an exercise with the dining philosophers or distributed sorting. We believe we can better validate formal methods by rationally reconstructing existing composite systems, and by applying the methods to solve industrial-size problems (such as the collision avoidance problem of [29]).

Our interest in designing multi-agent systems also overlaps that of distributed artificial intelligence. However, in our example designs, agents were simple in that they could only sense the current state of the system, and react according to pre-enumerated rules that do not allow any inference. If a signal failed to set, for instance, an operator in the McGean design could not infer that another action is necessary, formulate a new plan, and take corrective action. Agents in our designs cannot make inferences about their abilities [34], nor about their goals or commitments [6], nor about the goals or laws of the system as a whole. Our agents are thus more similar to the "situated automata" of [40], in that they are simple machines designed to meet more complex global properties.

Our work on composite systems extends that of Feather [12], which in particular introduced the notion of responsibility assignment. Dubois [11] 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 language integrates temporal logic and Petri nets, a combination adopted independently by Castro [5] 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 [45]. 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 [9]. Both of these approaches combat the notion that any design problem, composite or otherwise, starts from a blank slate. In [8], we discuss the linkage between KAOS and Critter on the elevator problem. Our collaboration with the KAOS project is continuing on our more recent design work on distributed network applications.

Acknowledgment

The authors wish to thank the anonymous reviewers and the editors for their comments and assistance. Jeff Kramer and Martin Feather also made useful comments on earlier versions of this paper.

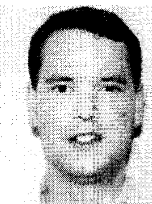
References

[1] M. Abadi and L. Lamport, "Composing specifications," Digital Systems Research Center, *Rep. 66*, Oct. 1990.
[2] J. S. Anderson and S. Fickas, "A proposed perspective shift: Viewing specification design as a planning problem," in *Proc. 5th Int. Workshop Software Specification and Design*, pp. 177–184, 1989.
[3] J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, Eds., *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*. Berlin: Springer-Verlag, 1989.
[4] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens, "METATEM: A framework for programming in temporal logic," in J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, Eds., *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*. Berlin: Springer-Verlag, 1989, pp. 94–129.
[5] J. Castro, "Distributed system specification using a temporal-causal framework," Ph. D. dissertation, Department of Computing, Imperial College of Science and Technology and Medicine, Univ. London, 1990.
[6] P. R. Cohen and H. Levesque, "Intention is choice with commitment," *Artificial Intelligence*, vol. 42, pp. 213–261, 1990.
[7] J. Conklin and M. Begeman, "gIBIS: A hypertext tool for exploratory policy discussion," *ACM Trans. Office Information Syst.*, vol. 6, pp. 303–331, Oct. 1988.
[8] A. Dardenne, S. Fickas, and A. van Lamsweerde, "Goal-directed concept acquisition in requirements elicitation," in *Proc. 6th Int. Workshop Software Specification and Design*, pp. 14–21, 1991.
[9] F. Dubisy and A. van Lamsweerde, "Requirements acquisition by analogy," *Int. Rep. 13*, KAOS Project, Institut d'Informatique, Facultés Universitaries de Namur, 1990.
[10] E. Dubois and J. Hagelstein, "A logic of action for goal-oriented elaboration of requirements," in *Proc. 5th Int. Workshop Software Specification*

and Design, published as *ACM SIGSOFT Engineering Notes*, vol. 14, pp. 160–168, May 1989.
[11] E. Dubois, "Supporting an incremental elaboration of requirements for multi-agent systems," in *Proc. Int. Conf. Cooperating Knowledge-Based Systems*, pp. 130–134, 1990.
[12] M. S. Feather, "Language support for the specification and development of composite systems," *ACM Trans. Programming Languages Syst.*, vol. 9, pp. 198–234, Nov. 1987.
[13] ——, "The evolution of composite system specifications," in *Proc. 4th Int. Workshop Software Specification and Design*, pp. 52–57, 1987.
[14] M.S. Feather, S. Fickas, and B. R. Helm, "Composite system design: The good news and the bad news," *Tech. Report CIS-TR-91-12*, Dept. Comp. Info. Sci., Univ. Oregon, 1991 (to appear in *Proc. Fourth Annual KBSE Conf.*, Syracuse, NY, Oct. 1991).
[15] S. Fickas, "Automating the transformational development of software," *IEEE Trans. Software Engineering*, vol. SE-11, no. 11, pp. 1268–1277, Nov. 1985.
[16] S. Fickas and P. Nagarajan, "Critiquing software specifications: a knowledge based approach," *IEEE Software*, Nov. 1988.
[17] S. Fickas, B. R. Helm, and M. S. Feather, "When things go wrong: Predicting failure in multi-agent systems," *Tech. Rep. CIS-TR-91-15*, Dept. Comp. Info. Sci., Univ. Oregon, 1991 (presented at the Niagra Workshop on Intelligent Information Systems, Niagra, NY, July 1991).
[18] G. Fischer, R. McCall, and A. Morch, "Janus: Integrating hypertext with a knowledge-based design environment," in *Proc. Hypertext 89*. New York: ACM, 1989, pp. 105–117.
[19] P. Friedland and Y. Iwasaki, "The concept and implementation of skeletal plans," *Rep. KSL 85–6*, Stanford Knowledge Systems Laboratory, 1985.
[20] M. Gasser, *Building a Secure Computer System*. New York: Van Nostrand Reinhold, 1988.
[21] K. J. Hammond, *Case-Based Planning: Viewing Planning as a Memory Task*. Boston, MA: Academic, 1989.
[22] G. Holzmann, *Design and Validation of Computer Protocols*. Englewood Cliffs, NJ: Prentice-Hall, 1991.
[23] P. Huber, A. Jensen, L. Jepsen, and K. Jensen, "Reachability trees for high-level Petri nets," *Theoretical Computer Science*, vol. 45, pp. 262–292, 1986.
[24] Problem set in *Proc. 4th Int. Workshop on Software Specification and Design*, 1987, pp. 52–57.
[25] R. Kemmerer, "Testing formal specifications to detect design errors," *IEEE Trans. Software Engineering*, vol. SE-11, pp. 32–42, Jan. 1985.
[26] B. Kramer, "Prototyping and formal analysis of concurrent and distributed systems," in *Proc. 6th Int. Workshop on Software Specification and Design*, 1991, pp. 60–66.
[27] J. Kramer, J. Magee, and A. Finkelstein, "A constructive approach to the design of distributed systems," in *Proc. 10th Int. Conf. Distributed Computing Systems*, May 1990.
[28] L. Lamport, "Proving the correctness of multiprocessor programs," *IEEE Trans. Software Engineering*, vol. SE-3, pp. 125–143, Mar. 1977.
[29] N. G. Leveson, M. Heimdahl, H. Hildreth, and A. Ortega, in *Proc. 6th Int. Workshop Software Specification and Design*, pp. 31–41, 1991.
[30] P. E. London and M. S. Feather, "Implementing specification freedoms," in *Readings in Artificial Intelligence and Software Engineering*, C. Rich and R. Waters, Eds. Los Gatos, CA: Morgan Kaufmann, 1986, pp. 285–205. (Originally appeared in *Sci. Computer Programming*, vol. 2, pp. 91–131, 1982.)
[31] J. Loyall, S. Kaplan, and S. Goering, "Abstraction and composition in D-specifications of concurrent systems," in *Proc. 6th Int. Workshop Software Specification and Design*, pp. 52–59, 1991.
[32] T. McGean, *Urban Transportation Technology*. Lexington, MA: D. C. Heath, 1976.
[33] R. Milner, *A Calculus of Communicating Systems*. Berlin: Springer-Verlag, 1980.
[34] L. Morgenstern, "A first-order theory of planning, knowledge, and action," in *Theoretical Aspects of Reasoning about Knowledge: Proc. 1986 Conference*, J. Halpern, Ed. Los Gatos, CA: Morgan Kaufmann, 1986, pp. 99–114.
[35] J. Mostow, "Why are design derivations hard to replay?" in *Machine Learning: A Guide to Current Research,* T. Mitchell, J. Carbonell, and R. Michaelski, Eds. Hingham, MA: Kluwer, 1986, pp. 213–218.
[36] J. B. Postel and J. K. Reynolds, "Telnet protocol specification," Internet Request For Comments 855, May 1983.
[37] W. Robinson, "A multi-agent view of requirements," in *Proc. 12th Int. Conf. Software Engineering*, pp. 268–276, 1990.
[38] M.T. Rose, *The Open Book: A Practical Perspective on OSI*. Englewood Cliffs, NJ: Prentice-Hall, 1989.
[39] ——, *The Simple Book: Management of TCP/IP-based Internets*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1991.
[40] S. Rosenschein and L. P. Kaelbing, "The synthesis of digital machines with provable epistemic properties," in J. Halpern, Ed., *Theoretical Aspects of Reasoning about Knowledge: Proc. 1986 Conf.*, pp. 83–98, 1986.
[41] A. V. Shankar and S. S. Lam, "Construction of network protocols by stepwise refinement," in J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, Eds., *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*. Berlin: Springer-Verlag, 1989, pp. 669–695.
[42] Robert B. Shaw, *Down Brakes: A History of Railroad Accidents, Safety Precautions, and Operating Practices in the United States of America*. London: P. R. Macmillan, 1961.
[43] R. Simmons, "A theory of debugging plans and interpretations," in *AAAI-88: Proc. 7th Nat. Conf. Artificial Intelligence*, pp. 94–99, 1988.
[44] A. J. Tanenbaum, *Computer Networks*. Englewood Cliffs, NJ: Prentice-Hall, 1988.
[45] A. van Lamsweerde, A. Dardenne, and F. Dubisy, "KAOS knowledge representations as initial support for formal specification processes," Facults Universitaries de Namur, Research Paper RP24/91, May 1991.
[46] M. C. Wilbur-Ham, "Numerical petri nets—A guide," Telecom Australia Research Laboratories, Report 7791, 1985.

Stephen F. Fickas received the B.S. degree in mathematics from Oregon State University, the M.S. degree in computer science from the University of Massachusetts, and the Ph.D. degree in computer science from the University of California, Irvine, CA.

He currently heads the Kate Project at the University of Oregon, and his interests are in formal models of the requirements and specifications process.



B. Robert Helm received the B.S. degree in computer science and mathematics from the University of Puget Sound, WA, in 1986 and the M.S. degree in computer science from the University of Oregon, Eugene, OR, in 1991.

He is currently pursuing Ph.D. research on formal analysis and transformation of software requirements at the University of Oregon.