Scare Tactics: Evaluating Problem Decompositions Using Failure Scenarios

B. Robert Helm and Stephen Fickas

Department of Computer and Information Science
University of Oregon
bhelm@cs.uoregon.edu   fickas@cs.uoregon.edu

Abstract

Our interest is in the design of multi-agent problem-solving systems, which we refer to as *composite systems*. We have proposed an approach to composite system design by decomposition of problem statements. An automated assistant called Critter provides a library of reusable design transformations which allow a human analyst to search the space of decompositions for a problem.

In this paper we describe a method for evaluating and critiquing problem decompositions generated by this search process. The method uses knowledge stored in the form of *failure decompositions* attached to design transformations. We suggest the benefits of our critiquing method by showing how it could re-derive steps of a published development example. We then identify several open issues for the method.

Introduction

Our group is interested in the design of *composite systems*, ones that encompass multiple agents cooperating in an ongoing activity [Fickas & Helm, 1992][1]. We arrived at this interest while studying the processes of software development. Systems analysts in the domains we studied [Fickas and Nagarajan, 1988] focused on policies and concerns which cut across human, hardware and software components. In composite system design, 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, called Critter, to help a human designer create a composite system design [Fickas and Helm, 1992].

Figure 1 shows the place of composite system design within the more general system lifecycle we envision. We view the design process of a system as composed of four phases:

1. **Acquisition.** The designer acquires an initial, informal statement of the problem in terms of text descriptions and diagrams.
2. **Formalization.** The designer creates an initial formulation of the problem in terms of *system* and *constraints*. The initial system formally describes a minimal set of assumptions about possible behavior of the target system.

The constraints formally describe the desirable behavior in terms of the initial system.
3. **Composite system design.** Given the formulation of the problem as initial constraints and system, the designer uses Critter to build a formal specification of a *composite system* for the problem. A composite system is a set of interacting, reactive components called *agents*. Each agent is associated with a set of *responsibilities*, constraints which the agent's behavior must satisfy. If all agents behave according to their responsibilities, the composite system will solve the desired problem.
4. **Implementation.** The agents of the composite system are implemented in the appropriate "technology" according to their specifications. This could mean producing software or manufacturing hardware. It might also involve writing legal statutes or training manuals describing the responsibilities of humans playing the role of an agent.

Figure 1.   Context of composite system design.
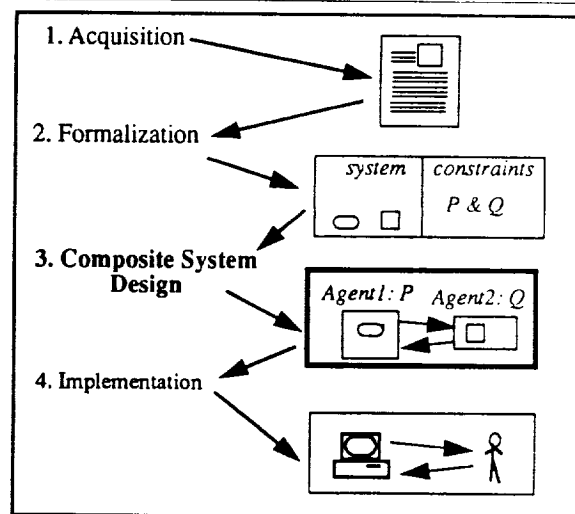
In designing a library circulation system, for example, the designer first acquires an informal statement of assumptions about the system, and constraints such as "Library patrons get the books they want" and "Every book is accounted for". The designer formalizes the system and constraints. The designer then uses Critter to design a composite system rep-

resenting the library. This formally specifies the responsibilities of agents such as the online catalog ("Report catalog entry if book title found"), the antitheft devices ("Sound alarm when magnetized book passes through the gate"), and even the library patrons ("Look in the online catalog if the book title is known"). Finally, the library agents are implemented. For the online catalog and antitheft device, this would involve writing or acquiring software and hardware. "Implementing" the library patron implies writing regulations and guidebooks to inform patrons of their role.

We have begun to formalize an approach to phase 3, composite system design, by decomposing problem statements. The designer incrementally decomposes the global constraints in the initial problem statement into the conjunction of more manageable subconstraints. The designer then *assigns responsibility* for these constraints to particular agents. For example, the designer of a library system could decompose the global constraint "Library patrons get the books they want" into "Library patrons can find the books they want" and "Library patrons can get the books they find." The patron and the online catalog agents are assigned responsibility for the former constraint; the patron and the library·staff agents are assigned responsibility for the latter. [Feather, 1987] illustrates the approach by an informal example.

Critter includes a library of formally-represented composite system design tactics, and a suite of tools for automated evaluation and critiquing of the designs generated. To incorporate the decomposition method into Critter, we need to (1) identify and formalize general tactics for decomposing problem statements, and (2) identify knowledge which Critter could use to critique problem decompositions.

This paper focuses on the latter problem, that of critiquing problem decompositions. We illustrate a method for generating critiques, by showing how it rationalizes specific steps in a published development example [Feather, 1987]. In that example, Feather informally derived an elevator system design from the global constraints of never unnecessarily delaying passengers, and never moving passengers further from their destination. The development was guided by Feather's intuitions of the problem, and his domain knowledge. We show how we can capture some of this knowledge, in the form of a library of *failure scenarios*. We then discuss the research issues raised by this example.

Our work addresses the workshop in two respects:

1. We propose general techniques for evaluating problem decompositions in multi-agent systems. These techniques may find use beyond our interests, in formulating problems for multi-agent planning or for distributed AI systems.

2. The evaluation approach we propose in this paper requires techniques for storing and using compiled abstractions, specifically abstract plans. This workshop may identify research we can apply to our approach.

Searching for decompositions

In this section, we outline the Critter composite system design model, and its support for synthesizing problem decompositions.

Critter treats composite system design as search in a state space (Figure 2). A typical search algorithm has the following components:

* A state space representation
* A set of search operators for moving between states
* A solution checker which recognizes satisfactory states.
* A heuristic evaluator which identifies promising states.
* A search manager which maintains a record of states visited and operators applied.

Figure 2   Composite design search.
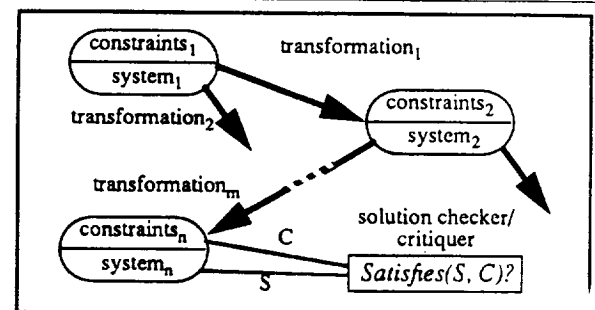
Each state in Critter's search space represents a c[om]plete composite system design for the problem at ha[nd]. The "search operators" which move from state to state are design transformations stored in Critter's knowledge base. The solution states in the search are acceptable composite system designs -- Critter provides critiquing tools to help identify these.

The last two components, heuristic evaluation and search management, are beyond the scope of our research at present. For heuristic evaluation, we rely on the human designer. Our studies of composite system design heuristics [Feather, Fickas, and Helm, 1991] [Fickas, Feather, and Helm, 1991] suggest that this task will have to remain with the designer in the foreseeable future. Support for human evaluation of design operators is the focus of other research [Johnson and Feather, 1991]. As for searc[h] management, Critter is implemented using an extended form of IBIS [Conklin and Begeman, 1988] that provides for separate design states. Critter provides functions for searching and backtracking in this space. In our current

implementation, new states are generated by hand-simulating operator application using an editor.

In the remainder of this section, we discuss Critter's support for the first three search components: state representation, transformations, and solution checking.

Design states

Figure 3 informally represents an initial state for the elevator design problem we use to illustrate our design approach. A state (hereafter "design state") in Critter's design search space has two parts:

1. *System.* The system portion defines the space of possible behaviors of the current composite system design.
2. *Constraints.* The constraint portion of a design state defines the subset of possible behaviors which are viewed as legal or desirable.

The system portion of a design state represents the space possible behaviors of the composite system. It specifies a set of objects, a set of primitive relations, and a set of actions which can add or delete object tuples from the relations. The system is thus similar to a planning domain for a STRIPS-like planner.

Relations and actions in the system portion are also labelled by *agents*. Agents in our model are simple reactive components. A relation labelled by an agent can be sensed by that agent; an action labelled by an agent can be controlled by that agent.

A behavior is a sequence of actions, each action labelled by its controlling agent. A prefix of a behavior represents the intermediate state of the composite system during its operation; to avoid confusion with design states, we will refer to execution states of the composite system as "points." As with planning domains, the system portion is non-deterministic; more than one action may be possible at a given point.

The system portion in Figure 3 includes two classes of agents, an elevator and set of passengers. Each passenger controls its own actions of entering and exiting elevators. A passenger can sense which floor it is on, and whether or not it is in a given elevator. Passengers also have a destination (not shown in the figure), which they know. The unique elevator controls its action of moving from floor to floor. It also can wait at a floor (not shown in the figure). The elevator can sense whether it is on a given floor.

The constraint portion of a design state is composed of a set of constraints. Each constraint is a predicate which is true or false for each behavior generated by the system portion. A constraint may refer to either relations or actions in the system portion.

The constraint portion of Figure 3 includes two constraints:

1. NeverFurther: Elevator passengers should not move further from their destinations.

2. NoDelays: Passengers should not be unnecessarily delayed. This means that at each point in the elevator's behavior, it must either move, take on, or drop off passengers, unless no passengers exist.

Agents in the system portion can be assigned *responsibility* for constraints. If an agent has been assigned responsibility for a constraint, that agent must act to satisfy the constraint. The agent must control its actions so that all of the behaviors it generates satisfy the constraint, regardless of the actions of other agents. We call a constraint which is the responsibility of some agent an "assigned constraint."

The legal behaviors of a composite system design are all sequences of actions which can be generated by the agents in its system portion, and which satisfy all of the constraints and responsibility assignments of the constraint portion.

Figure 3   Initial state of the elevator problem.
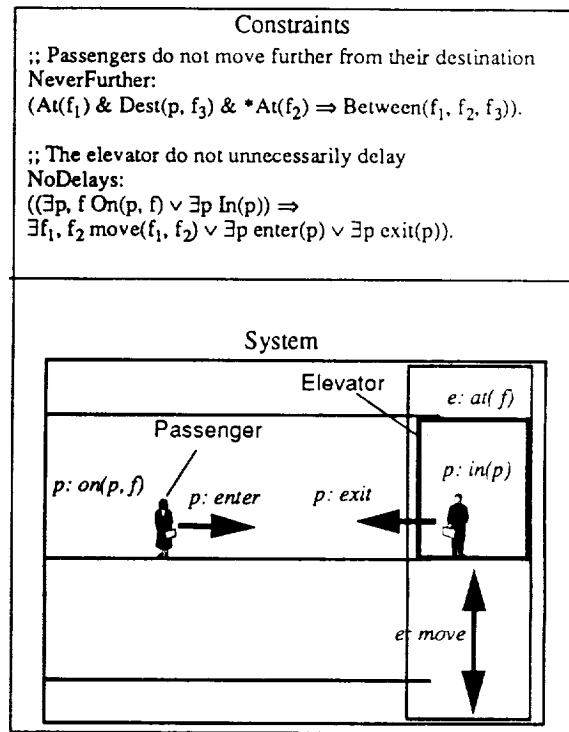
We have represented the system portion informally, which is adequate for the purposes of this paper. Critter represents the system portion of design states are expressed in a Numerical Petri Net [Wilbur-Ham, 1985] notation, extended to include agents.

The constraints are written in a linear-time, quantified temporal logic extended to include constructs for responsibility assignment [Dubois, 1990]. For the most part, the constraint notation used in this paper is simply the predicate calculus, except on the following points:

- Variables app ·ing in a constraint are universally quantified unle otherwise indicated.
- Actions appear as predicates in constraints. The expression $move(f_1, f_2)$ in the NoDelays constraint, for instance, states that "The elevator moves from f_1 to f_2 at the current point of the system's behavior." Ordinary predicates are capitalized to distinguish them from actions.
- Temporal logic operators reference future and past points in the system's behavior. The only construct we use in this paper is the * operator, which denotes the next point. Thus, the expression $At(f_1)$ & $Dest(p, f_3)$ & $*At(f_2)$ can be read "The elevator is at f_1 and passenger p has destination f_3 and at the next point the elevator is at f_2."
- Constraints can include responsibility assignment operators. [Feather, 1987] and [Dubois, 1990] give a formal semantics for this construct; we use it informally throughout.
- The notation $C[t/t']$ denotes the constraint C with all occurrences of t replaced by t'. Thus, the expression $NeverFurther[p/p_1]$ denotes the NeverFurther constraint with all occurrences of p replaced by p_1.

Design transformations

Critter has a library of design transformations that function as operators in the search for an acceptable composite system design. Each design transformation has a pattern which matches against parts of an existing design state, a result which generates in a new design state, and a list of conditions called domain assumptions that must hold for the transformation to apply (we do not discuss domain assumptions in this paper). We will represent the pattern and result of transformations as Prolog-like clauses.

Transformations are applied interactively. The human designer selects a transformation to apply, and matches the pattern of the transformation to components of the current design state. The system then generates a new design state incorporating the result of the transformation.

In design by problem decomposition, most of the transformations applied are of the following form:
pattern: constraint(C).
result: constraint(C_1 & ..., & C_n).

C in the pattern is a constraint. The transformation generates a new state where C is replaced by a new constraint C_1 & ... & C_n that entails C. This in turn may be decomposed into subconstraints.

When the designer judges that the constraints have been decomposed into sufficiently simple subconstraints, she assigns responsibility for each of the subconstraints to a single agent. As described above, assigning responsibility for a constraint C to an agent requires that agent to limit its

actions so that C is met, regardless of the actions of oth... agents in the system.

Finally, the designer applies transformations to unfold the assigned constraints onto the preconditions of actions in the system portion. The designer may also have to use low-level design editing transformations to change the details of actions and relations in the system.

Our main interest is in the transformations for decomposition of constraints and assignment of responsibility. As an example, one class of decomposition transformation used in this paper is *Zone Defense*. Intuitively, Zone Defense decomposes a constraint by

1. Selecting an object.
2. Dividing the object's lifetime into "zones", and
3. Splitting the con... into subconstraints based on the "zone" the object .

More formally, gi... constraint C and a universally quantified variable v ...ve decompose C into subconstraints based on poss... ates of objects to which v can be bound. The applica... of Zone Defense to the Never-Further constraint of the ...evator problem is as follows:
pattern: constraint(NeverFurther), uv(p, NeverFurther).
result: constraint((p_1)

$\exists f$ enter(p_1, f) $\Rightarrow$ NeverFurther[p/p_1]
&
$\exists f_1, f_2$ move(f_1, f_2) $\Rightarrow$ NeverFurther[p/p_1]
&
$\exists f$ exit(p_1, f) $\Rightarrow$ NeverFurther[p/p_1]
&
($\exists f$ enter(p_1, f) $\vee$ $\exists f_1, f_2$ move(f_1, f_2)
$\vee$ $\exists f$ exit(p_1, f))).

Intuitively, to ensure that passengers never move further from their destination, we can ensure that the constraint holds ... en the passenger enters an elevator, when an elevator r...ves, and when the passenger exits the elevat...r.

Having broken NeverFurther into more managc...nie subconstraints, the designer can next assign responsib...ity for one of the subgoals to the elevator. The only action the elevator controls are "move" and "wait", so we separate these subconstraints of the decomposition, and assign them to the elevator with the *Limit Each Action* transformation. The instantiation of this transformation on the move action reads as follows:
pattern: constraint((p_1)

$\exists f_1, f_2$ move(f_1, f_2) $\Rightarrow$ NeverFurther[p/p_1]
agent(elevator)).
result: constraint((p_1)
responsible(c1,
$\exists f_1, f_2$ move(f_1, f_2) $\Rightarrow$ NeverFurther[p/p_1]),
age... elevator)).

This tr... ...rmation requires that the elevator control each move a... ...1 so that NeverFurther holds, regardless of the actions ...t the passengers. Unfortunately, there is no way for both NeverFurther and NoDelays to be met if Never-

Further is assigned to the elevator as shown here. Two passengers going in different directions can enter the elevator and leave the elevator no choice but to either violate NeverFurther or NoDelays. We discuss this example further below.

Detecting solution states

A solution state in Critter's search is a design state where the system portion does not generate any behaviors which violate the constraints in the constraint portion. Critter includes analysis tools to help the analyst identify solution states. In [Fickas and Helm, 1992], we discuss several of these analysis tools and trade-offs between them. In this paper, we will discuss mainly the OPIE planning tool [Anderson and Fickas, 1989]. The system portion of a design state is effectively a planning domain. OPIE is a planner which shows that a design state is not a solution by producing a plan incorporating actions from the system portion for violating one or more constraints. We refer to such a plan as a *failure scenario*.

For example, to show that the initial elevator design state in Figure 3 is not a solution state, OPIE can generate a plan for violating the NeverFurther constraint from an initial point supplied by the analyst (+ indicates a relation added, - indicates a relation deleted):

Initial. On(p, 1), At(1), D(p, 2);
1. enter(p, 1): -On(p, 1) +In(p);
2. move(1, 3): -At(1) + At(3);

>>Violation of NeverFurther <<

At(1) & In(p) & D(p, 2) & *At(3)
& ¬ Between(1, 3, 2)

This illustrates the general style of solution testing in Critter; we focus on identifying classes of behaviors or scenarios which violate the constraints, rather than verifying that the constraints are met. In the next section, we discuss some of the benefits of this approach. We also identify some of its limitations, and suggest how to address those limitations in design by decomposition.

Critiquing with failure scenarios

Critiquing composite system design states by failure scenarios offers two benefits for design:
1. *Diagnosis.* A scenario is a specific behavior of the system which violates a constraint. The designer can use this behavior to diagnose the problems of the current design state and identify potential solutions.
2. *Validation.* The system portion of a design state is effectively a model of what is possible in the design domain. If a scenario generated from that model is

counterintuitive or unlikely in the domain, this is a hint that the model is too weak.

Our goal is to gain these benefits for design by problem decomposition. In this section, we suggest an approach to critiquing problem decompositions, and demonstrate the approach by showing how it could reproduce steps taken in a published composite system design derivation.

Synthesizing an approach

Planning over the system portion is not necessarily the best way to generate failure scenarios for decompositions, or for composite system designs in general. The planner cannot tell how likely, or how important a failure scenario it generates is. Consequently, it generates many scenarios with marginal value for design. More seriously, a designer can miss important failure scenarios in a design problem by "naive specification" of the problem. The planner relies entirely on the information in the design state to generate critiques. This knowledge may be incomplete or incorrect with respect to the design domain. The designer can exclude a particular failure, even a common one, by not including actions in the system portion which allow the planner to generate that failure. For example, the designer of a library can miss the possibility of books being stolen, by not encoding a "steal book" action in the initial design state.

A critic with domain knowledge can focus more quickly on serious problems, and can recognize problems even in naive specifications. We describe a domain-specific critic called SKATE for library design in [Fickas and Nagarajan, 1988]. SKATE has a case base of 1) library designs, 2) constraints they meet or violate, and 3) failure scenarios for those designs. Given a proposed design and a set of constraints, SKATE retrieves designs from its case base that match features of the proposed design, and that violate the proposed constraints. It then runs failure scenarios from the retrieved designs to demonstrate the problems. Given a library design including unrestricted checkout of books, for instance, and a constraint "users have a large selection of books to choose from", SKATE retrieves a design case with unrestricted checkout. It then executes a stored failure scenario of a "run" on the library, in which unrestricted checkout is used to strip the shelves bare.

SKATE's case base points it directly to well-known library failure scenarios, avoiding the problem of generating marginally useful scenarios. SKATE also avoids the problem of naive specifications. The failure scenarios SKATE generates are not restricted to using the actions and relations specified in the proposed library design. They can also include "environment" actions such as stealing or destroying books, which a designer might not specify but which are known to cause problems in the library domain.

SKATE, however, suffers from a limited ability to match designs against cases. In general, it is hard to match the features of one arbitrary specification to another [Robinson, 1990]. SKATE requires the user to manually map features of the proposed library design into features used in SKATE's case base. This task is onerous and error-prone; important critiques can be missed by user mistakes in the mapping process.

One solution proposed by Fickas and Nagarajan is to integrate matching more closely with the process of producing designs. They suggested that the proposed design be generated by domain-specific editor, equipped with a collection of library components appearing in the case base. In effect, this limits the designer to producing designs SKATE knows how to critique.

Based on these considerations, we propose the following approach which integrates the approaches of OPIE and SKATE:

1. We will use Critter's transformation library in place of the case base of SKATE. Each decomposition transformation has an attached set of failure scenarios representing its typical defects. Critter thus plays the role of the domain-specific editor proposed by Fickas and Nagarajan.

2. Critter matches failure scenarios when it applies a transformation. Matching is simpler, compared to SKATE, because the instantiation of the transformation itself guides the matching process.

3. Critter critiques a design state using the OPIE planner. OPIE produces plans by specializing and refining previously matched failure scenarios.

This approach addresses the problem of marginally useful scenarios by storing a library of typically useful scenarios on transformations, and using these scenarios to focus the planner. Our study of failures in multi-agent systems [Fickas, Feather, & Helm, 1991] suggests that we can find such characteristic failure scenarios for problem decompositions. The approach also addresses the naive modelling problem by allowing failure scenarios to introduce new relations and actions into the design state being critiqued. As in SKATE, these "environment" components represent knowledge of well-known problems that crop up in multi-agent systems.

To illustrate this approach, we next show how critiques generated this way could anticipate two design steps which occurred in the composite system design development described in [Feather, 1987].

Focusing on a decomposition failure

Recall that Feather's elevator design problem had two initial constraints:

1. Passengers should never move further from their destination (NeverFurther).

2. Passengers should not be unnecessarily delayed (NoDelays).

From the constraint that passengers never move further from their destination, the designer in Feather's example "chooses the implication" that passengers in the same elevator must be travelling in the same direction. We show how a failure scenario can focus the planner to reproduce this design step.

Earlier we showed a development step which assigned the NeverFurther goal to the elevator. This step used a transformation called Limit Each Action. As noted above, this assignment requires the elevator to satisfy NeverFurther for all combinations of passengers and floors, regardless of prior actions of the passengers involved. Critter can generate an interesting counterexample to this constraint using a scenario attached to the Limit Each Action transformation. The attached scenario is called "incompatibility conspiracy". The abstract incompatibility conspiracy scenario requires that:

1. There are two agents in the system portion whose state can affect the truth of the constraint assigned by the transformation.

2. These two agents can act to reach a state S where an application of the action A will fail to satisfy the constraint for either one agent, or for the other. For the assigned constraint C and limited action A, we can compute the conditions on the state S the conspiring agents must reach. Specifically, we regress $\exists a1, a2 \neg (C[a1] \& C[a2])$ through the action A.

Instantiating the scenario on the application of Limit Each Action, we get a goal of generating a state where:

- There are two passengers in an elevator on a floor f_1
- The two passengers have destinations f_3, f_4
- No floor f_2 exists such that Between(f_1, f_2, f_3) & Between(f_1, f_2, f_4)

It remains for the planner, OPIE, to try to extend this minimal "scenario" into a plan. This requires a considerable effort on OPIE's part. If such a plan can be found, however, it provides a motivation for the requirement that passengers only enter the elevator with compatible passengers -- passengers travelling in the same direction.

Using an abstract failure scenario thus allowed the planner to recognize a critical deficiency, one which Feather deduced informally in his example.

Critiquing a naive communication model

In another step of Feather's development of the elevator problem, passengers have been assigned to enter the elevator when a suitable one arrives at the passenger's floor. The elevator has been assigned to take passengers to their destination when they enter. From this, the designer in Feather's example derives the constraint that the passengers communicate their presence on entering the elevator. We show how an abstract failure scenario could lead a

designer to this communication protocol, by introducing environment actions and relations which cause a stereotypical breakdown of communication.

The starting point for this development is the NoDelays goal, which requires that the elevator must either move or load and unload passengers when any passenger is present. The designer applies a macro-transformation called *Sequential Split* to the NoDelays goal. This transformation combines a Zone Defense operator with responsibility assignment. It subdivides the task of moving passengers into sequential zones, based on the status of the passenger. In particular, the designer uses Sequential Split to make passengers responsible for NoDelays when the elevator arrives at a floor. Responsibility passes sequentially to the elevator once the passenger enters. The instantiated version of Sequential Split expresses this formally:

pattern: constraint(

On(p, f) & At(f) $\Rightarrow$ NoDelays),

agent(p), agent(elevator).

result: constraint(

On(p, f) & At(e, f) $\Rightarrow$ Responsible(p, enter(p, f))

&

(In(p) & At(f$_1$) $\Rightarrow$

Responsible(elevator, $\exists$ f$_2$ move(f$_1$, f$_2$)))))

agent(p), agent(elevator).

Note that the requirement that the elevator moves, coupled with the NeverFurther constraint, ensures that the passenger will eventually arrive at its destination.

Our studies of transportation system failures suggest that sequential decompositions, while common, frequently fail due to "hand-off errors". In one hand-off failure scenario, for instance, the agent responsible for the second half of a sequential decomposition fails to pick up where the first agent leaves off, because it does not recognize it has become responsible. Translating this to the current problem, the elevator may fail to move, because it does not recognize that the passenger has entered and thus handed off responsibility for NoDelays.

This sequence of events is encoded as an abstract scenario attached to Sequential Split. Instantiated with the Sequential Split transformation above, it asks the planner to expand a sequence of states where:

1. $\exists$p, f On(p, f) & At(f);

2. In(p) & At(f) & –ElevatorResponsibleForMove

Note that the abstract scenario introduces a new binary relation ElevatorResponsibleForMove. This relation represents the elevator's internal model of the condition that activates its responsibilities. The failure scenario also introduces actions for asserting and deleting this relation. As with SKATE scenarios, abstract scenarios in Critter can add environment actions and relations to the design state for use in generating critiques. In this example, the new components allow OPIE to generate a plan in which a passenger enters the elevator, but the elevator does not

recognize this (ElevatorResponsibleForMove is false), and so does not move.

Environment components introduced by attached scenarios allow OPIE to avoid the naive modelling problem. They force the designer to consider behavior which is typical for a class of problem decompositions, even if the designer has neglected to include components which support such behavior in the initial design state.

Returning to our example, the designer acknowledges the scenario, and designs a communication protocol to prevent it. The passenger becomes responsible for notifying the elevator when it enters the elevator. The elevator will acknowledge the handoff. This can be implemented by a familiar interface: passengers hit a button on entry to the elevator, and the button lights in response.

The handoff failure scenario thus produces and rationalizes an interface component developed in the Feather example. This step also shows how a failure scenarios incorporating environment components can expose naive assumptions about inter-agent communication, and lead to more realistic agent interfaces as a result.

Conclusions and Issues

We have proposed an approach to composite system design based on problem decomposition. To evaluate designs generated by the approach, we have proposed a method of scenario-based critiquing using compiled knowledge of typical failures of problem decompositions. Our method combines the approaches of earlier plan-based and case-based design critics we have developed. It addresses the problem of matching cases which stymied the case-based critic. It also helps solve the problems of unfocused search and naive modelling which were the principle drawbacks of the plan-based critic.

There remain numerous open research issues for the approach. Two issues in particular may be of interest to this workshop.

First, can we store scenarios on transformations which are specific enough to be more useful than simply running the planner? For example, the incompatibility conspiracy scenario was extremely general, and costly to instantiate. OPIE could possibly find the associated plan just as quickly by directly analyzing the design state. One rejoinder is that the transformation associated with the incompatibility conspiracy scenario, responsibility assignment, is too general to have useful scenarios associated with it. Increasing the grain size of transformations, and placing scenarios only on the large-grained transformations, might give better results on evaluation, but at a cost of increasing the size of Critter's transformations and complicating their application. The research issue: how can we evaluate the trade-off between more effective evaluation knowledge, versus more general problem decomposition methods?

Related to the issue of transformation versus scenario grain size is the question of combining multiple failure scenarios. For example, consider the step which split the NoDelays goal. In that step, we applied Sequential Split, which combined three smaller transformations (Zone Defense and two responsibility assignments). The result was tested by scenarios stored on Sequential Split. Suppose instead we had applied the three primitive transformations. How should we merge the separate stored failure scenarios into a combined scenario; or, alternatively, how can we decide which of the scenarios is the most important to run?

Related Work

Our work extends and formalizes that of Feather [1987], who proposed the concept of responsibility assignment and informally demonstrated a development methodology based on decomposition and assignment of constraints. [Dubois, 1990] developed a constraint formalism, and a development methodology incorporating responsibility assignment, which has influenced our own work.

The decomposition design process can be viewed as a multi-agent extension of "operationalization" [Mostow, 1983]. Mostow's FOO and BAR systems designed problem-solving programs by decomposing and weakening constraints until they were expressible in terms of easily computable functions. The problem-solving systems we are designing, however, incorporate a broad range of social, hardware, and software systems. Consequently, it is difficult to state a compact operationality criterion for a given design problem. We rely on the human analyst to judge operationality. Similarly, constraint violations in our design problems may have consequences ranging from trivial to life-threatening. Weakening and approximating constraints therefore is much more problematic; we do not attempt to address it with our current research.

[Steier and Kant, 1985] argue for the importance of execution in designing algorithms. Our style of critiquing is motivated by similar considerations. The approach we propose grows out of our previous work on case-based [Fickas and Nagarajan, 1988] and planner-based [Anderson and Fickas, 1989] critics. [Dubois and Hagelstein, 1988] propose a slightly different approach to critiquing: derive implications by forward inference over the constraints, and present them to the user for validation. A critic using this approach requires knowledge to decide which deductions to make; abstract failure scenarios provide our method with this guidance.

Critter's critiquing task is similar to that of failure critics in planning systems such as CHEF [Hammond, 1989].The failure critics of CHEF attempt to steer CHEF's planner away from two types of failures:

1. Planning failures. These occur when the planner generates a plan that does not meet its goals, due to a false move by the part of the planner e. g. misordering two interacting steps.
2. Expectation failures. These occur when the planner generates a plan which does not meet its goal when executed in the environment of interest. Expectation failures arise when the planner's knowledge of its domain is incomplete or incorrect.

CHEF includes mechanisms for learning new failure critics from past planning or expectation failures. It also automatically indexes failures to planning moves that avoid those failures, and to moves which repair those failures.

In Critter, the "planner" is the user, and the "planning moves" are the transformations in Critter's library. The failure scenarios on a transformation identify both planning failures and expectation failures which could arise from using that transformation.

Critter does not, however, automatically learn failure scenarios from failures when they are encountered, due to the generality of its transformation library. CHEF was designed to operate within a fairly specific task domain (its example domain was Szechuan cooking). Consequently, it did not have to be too "finicky" in its choice of failures to learn [Minton, 1990]. In contrast, we hope to reuse Critter's knowledge base across diverse domains, such as transportation systems, network applications, and resource management systems. This makes it more difficult to automatically decide whether a given failure scenario is worth storing, and at what level of abstraction it should be stored. Our initial focus is thus on automated reuse of handpicked failure scenarios; learning the scenarios from previous design effort is a topic for future work.

Similarly, Critter does not automatically index from failures to avoidance or repair transformations. The "plans" (formal specifications) that Critter produces are allowed to contain more complex operators -- iterative, conditional, and uninstantiated operators, for example -- than the plans of CHEF. This makes it harder to explain a failure, assign blame for the failure to specification components, and index through those components to relevant transformations. For the present, we rely on the designer to perform indexing, but view it as an important area for future research.

References

[Anderson and Fickas, 1989] J. S. Anderson and S. Fickas, "A proposed perspective shift: viewing specification design as a planning problem," in *Proc. 5th International Workshop on Software Specification and Design* (Pittsburgh, PA, USA). Los Alamitos, CA: IEEE Computer Society, 1989, p. p. 177-184.

[Conklin and Begeman, 1988] J. Conklin and M. Begeman, "gIBIS: a hypertext tool for exploratory policy discussion," *ACM Trans. Office Information Systems*, Vol. 6, No.4, p. p. 303-331, October 1988.

[Dubois, 1990] E. Dubois, "Supporting an incremental elaboration of requirements for multi-agent systems," in *Proceedings: International Conference on Cooperating Knowledge-based Systems* (Keele, UK), 1990. Springer-Verlag: 1990, p. p. 130-134.

[Dubois and Hagelstein, 1988] E. Dubois and J. Hagelstein, "A logic of action for goal-oriented elaboration of requirements," in *Proc. 5th International Workshop on Software Specification and Design* (Pittsburgh, PA). Published as *ACM SIGSOFT Engineering Notes*, Vol. 14, No. 3, p. p. 160-168, May 1988.

[Feather, 1987] M. S. Feather, "Language support for the specification and development of composite systems," *ACM Trans. Programming Languages and Systems*, Vol. 9, No. 2, p. p. 198-234, November 1987.

[Feather, Fickas, and Helm, 1991] M.S. Feather, S. Fickas, and B. R. Helm, "Composite system design: the good news and the bad news," in *Proc. Fourth Annual KBSE Conference* Syracuse, NY, October 1991.

[Fickas and Helm, 1992] S. Fickas, B. R. Helm, "Knowledge representation and reasoning for the design of composite systems," *IEEE Transactions on Software Engineering*, Vol. 18, No. 6, June 1992, to appear.

[Fickas, Feather, & Helm, 1991] S. Fickas, B. R. Helm, and M. S. Feather, "When things go wrong: predicting failure in multi-agent systems," Dept. of Comp. and Info. Sci., Univ. of Oregon, Tech. Report CIS-TR-91-15, 1991 (Presented at the Niagara Workshop on Intelligent Information Systems, Niagara, NY, July 1991).

[Fickas and Nagarajan, 1988] S. Fickas, P. Nagarajan, "Being suspicious: critiquing problem specifications," in *Proceedings: AAAI-88 The Seventh National Conference on Artificial Intelligence* (St. Paul, Minn), 1988. AAAI Press, 1988, p. p. 19-24.

[Hammond, 1989] K. J. Hammond, *Case-Based Planning: Viewing Planning as a Memory Task*. Boston, MA: Academic Press, Inc., 1989.

[Johnson and Feather, 1991] W. L. Johnson, M. S. Feather, "Using evolution transformations to construct specifications", in M. Lowry and R. McCartney (eds.), *Automating Software Design*. AAAI Press, 1991.

[Minton, 1990] S. Minton, "Quantitative results concerning the utility of explanation-based learning," *Artificial Intelligence*, Vol. 42, p. p. 363-392. Also in J. W. Shavlik and T. G. Dietterich (eds.), *Readings in Machine Learning*. San Mateo, CA: Morgan Kaufmann, 1990, p. p. 573-587.

[Mostow, 1983] J. Mostow, "A problem solver for making advice operational," in *Proceedings: AAAI -83*. Morgan Kaufman, 1983, p. p. 279-283.

[Robinson, 1990] W. Robinson, "A multi-agent view of requirements," in *Proc. 12th International Conference on Software Engineering* (Nice, France), 1990, p. p. 268-276.

[Steier and Kant, 1985] D. Steier, E. Kant, "The roles of execution and analysis in algorithm design," *IEEE Transactions on Software Engineering*, Vol. 11, No. 11, Nov. 1985.

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

93