Scare Tactics: Evaluating Problem Decompositions Using Failure Scenarios

B. Robert Helm
Stephen Fickas

Oregon CIS-TR-92-06
February 28, 1992

Abstract

Our group is interested in the design of *composite systems*, ones that encompass multiple agents involved in ongoing, interactive activities. We have begun to formalize an approach to composite system design by decomposing problem statements. This involves (1) identifying and formalizing general tactics for decomposing problem statements, and (2) identifying knowledge which we could use to critique problem decompositions.

This paper focuses on the latter problem, that of critiquing problem decompositions. We describe a method for generating critiques using knowledge encoded as *failure scenarios*. We illustrate the method by showing how it rationalizes specific steps in a published development example. We then discuss the research issues raised by this example.

Submitted to the Workshop on Problem Reformulation and Representation Change,
Asilomar, CA, April 1992.

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

1.0 Introduction

Our group is interested in the design of *composite systems*, ones that encompass multiple agents involved in ongoing, interactive activities [Fickas and Helm, 1992]. 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 begun to formalize an approach to composite system design by decomposing problem statements. The designer begins with a statement of the global goals or constraints which the design should meet. The designer then incrementally decomposes the global constraints in the initial problem statement into the conjunction of more manageable subconstraints. Finally, the designer *assigns responsibility* for these constraints to particular agents. [Feather, 1987] illustrates the approach by an informal example.

For example, in designing a library circulation system, we start with constraints such as "Library patrons get the books they request" and "Every book is accounted for". We then decompose these constraints into a design of the system, including the responsibilities of systems agents. The decomposition gives pieces of the global constraints to human agents such as library patrons ("Look in the online catalog if the book title is known"), to mechanical systems such as antitheft devices ("Sound alarm when magnetized book passes through the gate"), and to software such as the catalog server ("Report catalog entry if book title found").

Elsewhere [Fickas and Helm, 1992] we have argued for the benefits of composite system design, and for an incremental approach to it. We have developed a design model, called Critter, to help a human designer create a composite system design. 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. We have begun to incorporate the problem decomposition method into Critter. This involves (1) identifying and formalizing general tactics for decomposing problem statements, and (2) identifying 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.

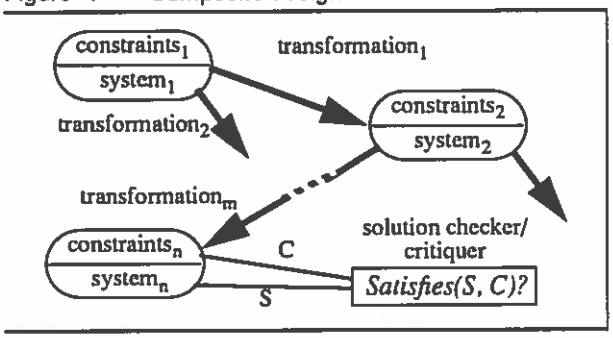
2.0 Searching for decompositions

We are interested in designing composite systems, systems of human, software, and hardware agents which interact to satisfy some global constraints. In this section, we outline the Critter composite system design model, and its support for generating problem decompositions.

Critter treats composite system design as search in a state space (Figure 1). Each state in the space represents a complete composite system design for the problem at hand. The "search operators" which move from state to state are design transformations. The solution states in the search are acceptable composite system designs.

Critter's search is interactive -- a human designer selects which state to expand next, and which operator to apply. The designer also decides when a solution state has been reached. Critter assists by applying operators, and by critiquing states proposed as solutions.

Figure 1 Composite design search.



2.1 Design states

Figure 2 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.

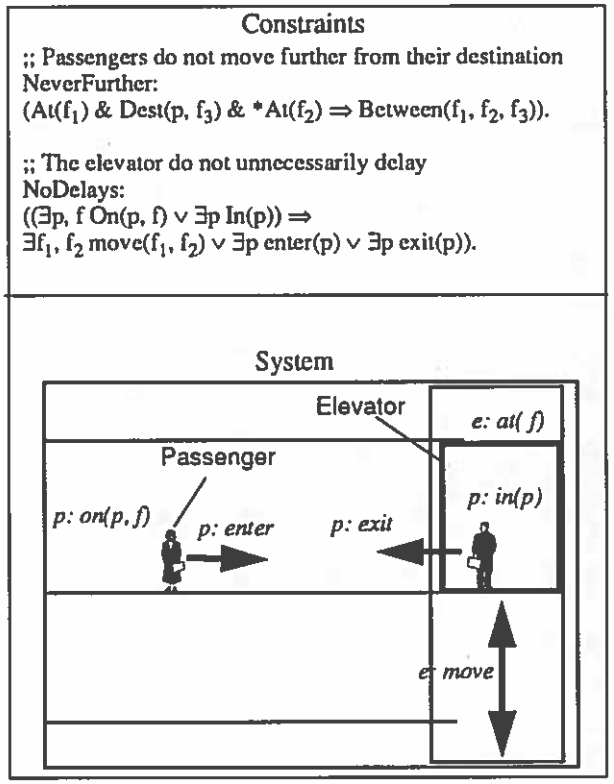
The system portion of a design state specifies a set of objects, a set of primitive relations, and a set of actions which can add or delete object tuples from the relations. The system is thus similar to a planning domain for a STRIPS-like planner.

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 2 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.

Figure 2 Initial state of the elevator problem.



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 2 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.

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 appearing in a constraint are universally quantified unless 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.

2.2 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 which 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 other 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 constraint into subconstraints based on the "zone" the object is in.

More formally, given a constraint C and a universally quantified variable v of C, we decompose C into subconstraints based on possible states of objects to which v can be bound. The application of Zone Defense to the NeverFurther constraint of the elevator 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 when the passenger enters an elevator, when an elevator moves, and when the passenger exits the elevator.

Having broken NeverFurther into more manageable subconstraints, the designer can next assign responsibility 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]),
agent(elevator)).

This transformation requires that the elevator control each move action so that NeverFurther holds, regardless of the actions of the passengers. Unfortunately, there is no way for both NeverFurther and NoDelays to be met if NeverFurther 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.

2.3 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 <u>not</u> 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 2 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.

3.0 Critiquing with attached 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.

3.1 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. 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 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 et al., 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 cri-

tiqued. As in SKATE, these "environment" components represent knowledge of well-known problems that crop up in multiagent 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].

3.2 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.

In section 2.2 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.

3.3 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) ⇒ NoDelays),
    agent(p), agent(elevator).
result: constraint(
    On(p, f) & At(e, f) ⇒ Responsible(p, enter(p, f))
    &

(In(p) & At(f_1) ⇒
Responsible(elevator, ∃ 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 the 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. ∃p, f On(p, f) & At(f);
2. In(p) & At(f) & ¬ElevatorResponsibleForMove

Note that the abstract scenario introduces a new 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 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.

4.0 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 can 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 NoDelays step in section 3.3. 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? Alternatively, how can we decide which of the scenarios is the most important to run?

5.0 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 we propose 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.

6.0 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.

[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.

[Fickas and Helm, 1992] S. Fickas, B. R. Helm, "Knowledge representation and reasoning for the design of composite systems," *IEEE Transactions on Software Engineering*, in press.

[Fickas et al., 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.

[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.