

# **A Transformational Approach to Composite System Specification**

**Stephen Fickas  
Rob Helm**

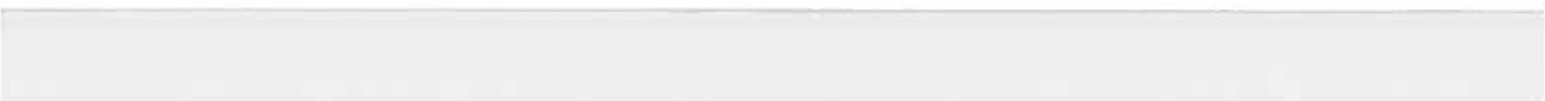
**CIS-TR-90-19  
November 16, 1990**

---

## **Abstract**

This paper presents a transformational framework for the design and rationalization of composite systems, and identifies some prerequisites to automated design within that framework.

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





## 1.0 Introduction

Our interest is in the construction of what we call Composite System Specifications, or CSS for short. A CSS differs from traditional formal specifications in the following ways:

1. A CSS typically involves multiple agents, where each agent may have its own abilities, goals, etc.
2. The construction of a CSS is centered on the notion of responsibility assignment (Feather, 1987). This in turn rests on an agent's abilities, motivation, and reliability.
3. The final CSS is often the result of inter-agent negotiation and compromise.

Roughly, the construction of a CSS starts with 1) an initial formal model of a system, and 2) one or more system-wide goals. The final product is a new model and/or new goals such that the goals are achieved by the model. Clearly, if the initial goals are met by the initial model, we are done. More typically, model and goals are modified during specification construction. It is this modification process we are interested in this paper. In particular, we propose an extended form of transformational development as an approach to formalizing and automating the process. We then identify some of the main problems which must be overcome to apply this approach to composite system specification.

### 1.1 The problem in terms of Glitter

Our system for construction of a CSS has similarities with an earlier transformational system called Glitter (Fickas, 1983), (Fickas, 1985). Briefly, Glitter's problem was to aid a human in efficiently implementing a formal specification written in the Gist language.

Figure 1 shows the transformational approach used by Glitter. Glitter explicitly represented the state of the specification design process in a *development state*. Given a focus on a development goal, methods were proposed from a library to achieve the goal. An evaluation of the alternative methods led to a weighting of competing methods and a selection. The selected method was applied to generate a new development state. The

development state itself represented both the state of the specification under development, and the state of the problem-solving process which led to that specification.

The specification half of the development state represented the state of the specification as it moved from abstract-and-inefficient to concrete-and-efficient. To accomplish these moves, Glitter method catalog included correctness-preserving transformations, which replaced inefficient constructs in the specification with equivalent, more efficient ones.

Glitter also represented the state of problem-solving through a tree of past and present development goals or tasks. It had at its disposal a catalog of abstract, goal-achievement methods, which did not directly transform the specification, but which gradually narrowed the range of transformations to apply, allowing Glitter to make plausible "development moves".

In summary, a development state in Glitter had two components: the specification state and the problem-solving state (the development tree). There were "operators" defined for applying to either half.

A major Glitter component of interest to us is its selection rules. Frequently, more than one operator was proposed in the current development state. The choice among competing operators was critical - a bad choice could lead to inefficient programs in the case of specification operators (i.e., transformations), and to much wasted problem-solving effort or even dead-end states for goal-achievement methods.

To address the selection problem, Glitter maintained a catalog of selection rules for choosing among alternative operators. These rules spanned a wide range of both automation and domain knowledge. Most typically, they would combine simple types of theorem proving with domain specific questions to the human user. Once this information was gathered, the rules would make recommendations on the alternatives available in the current state. The highest rated method would be chosen, and the cycle would repeat.

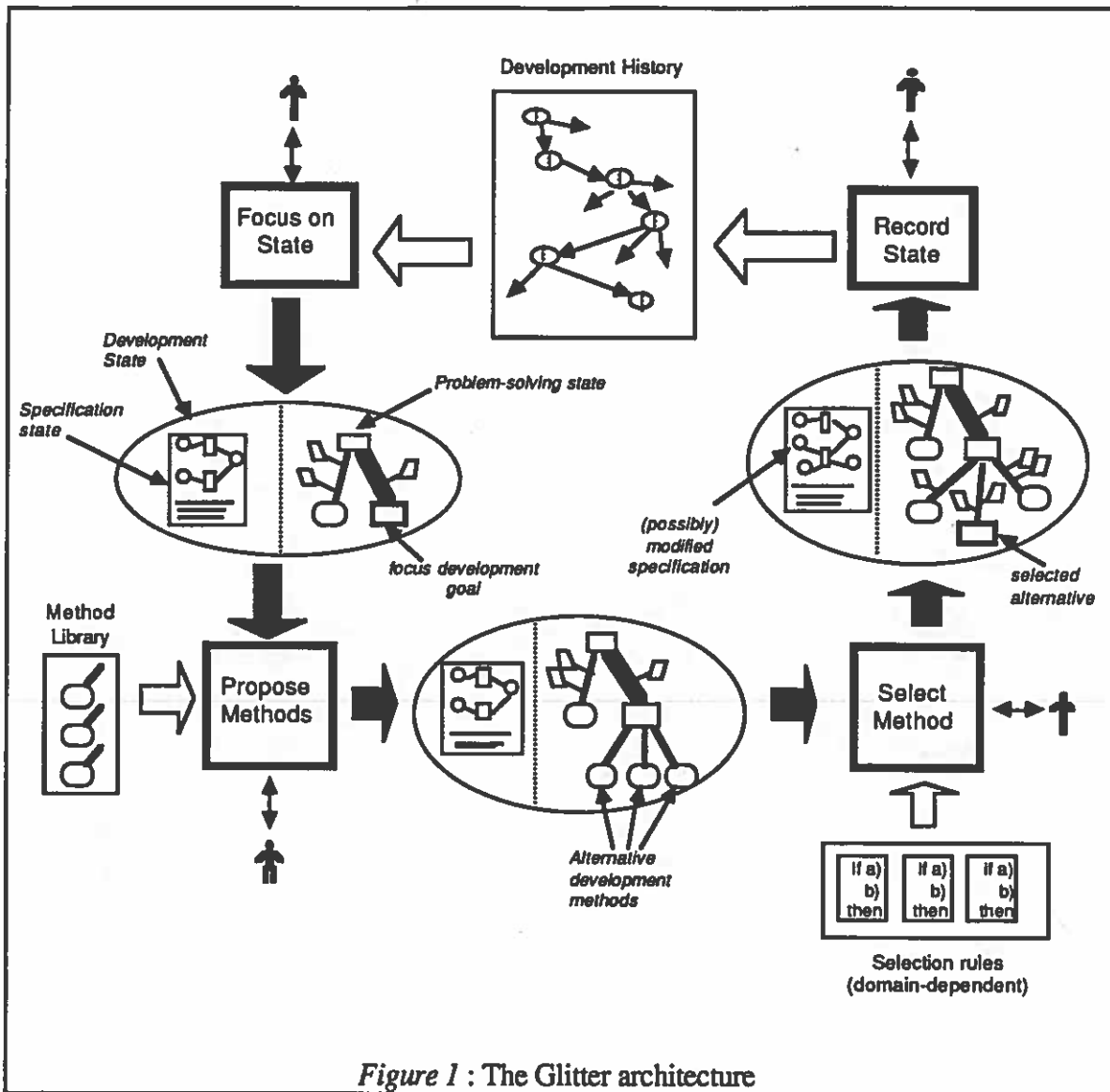


Figure 1 : The Glitter architecture

## 1.2 Critter -- Applying the Glitter model to CSS.

We have reapplied the Glitter model to the construction of CSS in a hypothetical system called Critter. In doing so, we made the following correspondences:

Development state:

The Critter development state, like Glitter's represents both the state of the specification, and of the problem-solving process. In Critter, we refer to the the problem-solving state as the "task tree".

**Specification operators:**

In Glitter, we maintained a catalog of correctness-preserving transformations. In Critter, we maintain a catalog of both correctness-preserving and non-correctness-preserving transformations. The latter are used to make stereotypical, but meaning-changing modifications to the specification.

**Development operators:**

Critter uses the same style of problem solving goals (tasks) and methods as Glitter. The difference is that Glitter's Gist-specific problem solving language is replaced with one centered on CSS construction, e.g., assignment of responsibility, conflict resolution.

**Evaluation rules:**

At present, Critter's evaluation component is equivalent to the selection rules of Glitter: a homogenous rule-based system that queries the user when it needs information. However, our research focus is squarely on extending this component of Critter to represent and integrate complex models of effectiveness, motivation, reliability, etc.

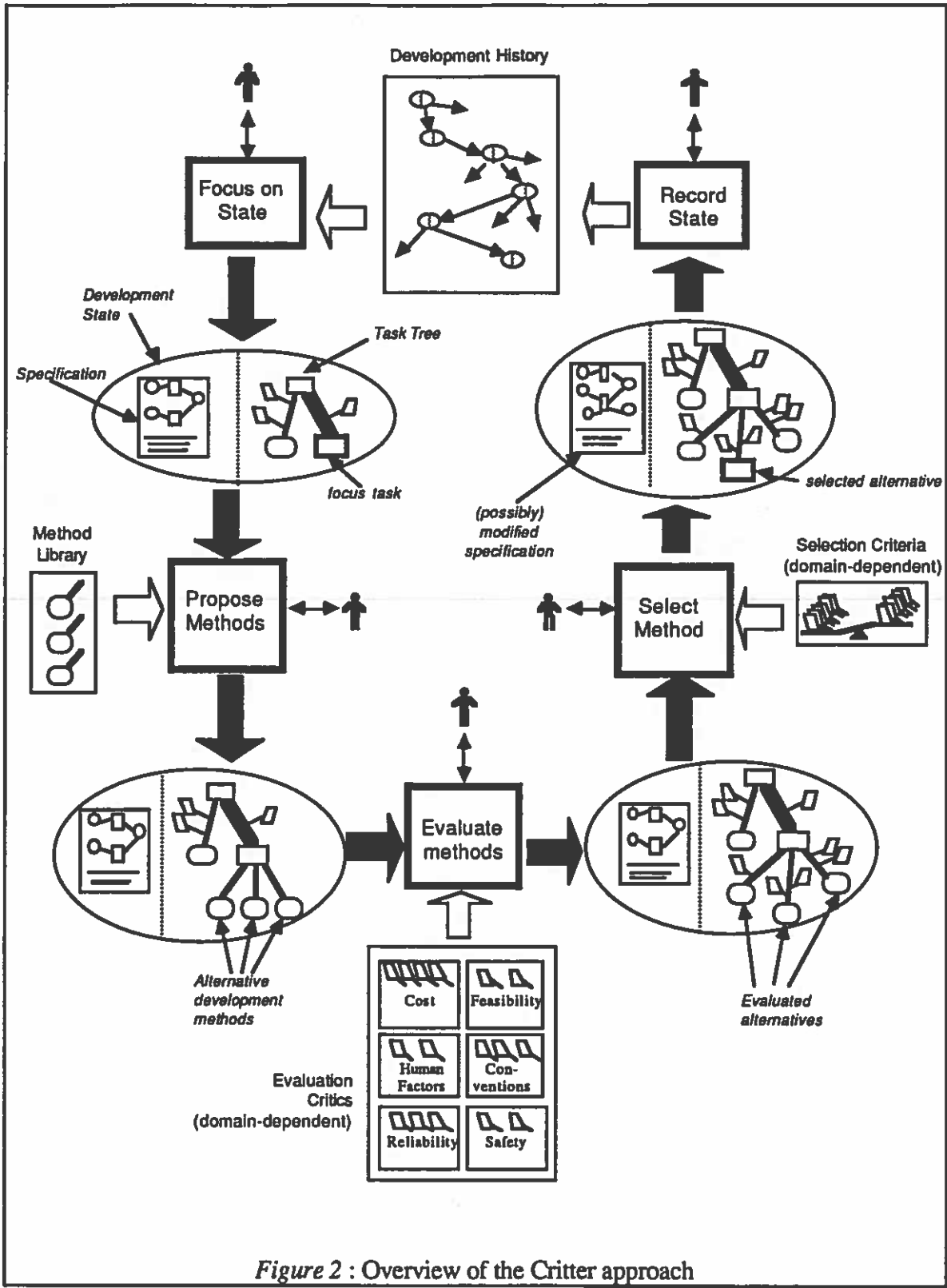
**Theorem prover:**

In Glitter, a theorem prover was used to prove some simple equivalence properties of expressions. In Critter, we rely on a planner (Anderson & Fickas, 1989) to prove certain specification properties, and to provide execution traces to assist more sophisticated analysis by the human analyst.

**The human user:**

Like Glitter, Critter uses an assistant-based approach to problem solving. We do not suspect that the user can be "automated out" of our system in the foreseeable future.

Figure 2 is the graphical view of Critter. We have begun to fill in pieces of the various catalog and components in the figure. In this paper, we raise the key topics of using a transformational development system like Critter to address the problems of CSS construction.



## 2.0 An example.

To illustrate our approach and some of the issues it raises, we next present a detailed example. In the example, we derive part of the specification of an elevator system. Figure 8 previews the example development. We assume that, as a result of prior development, the analyst begins in the development state S1.

As described above, the development state at each point records the current specification state and the current task tree. The initial specification state is supplied by the analyst. The specification state is encoded in a language we call Composite Petri Nets (CPN), and has three components:

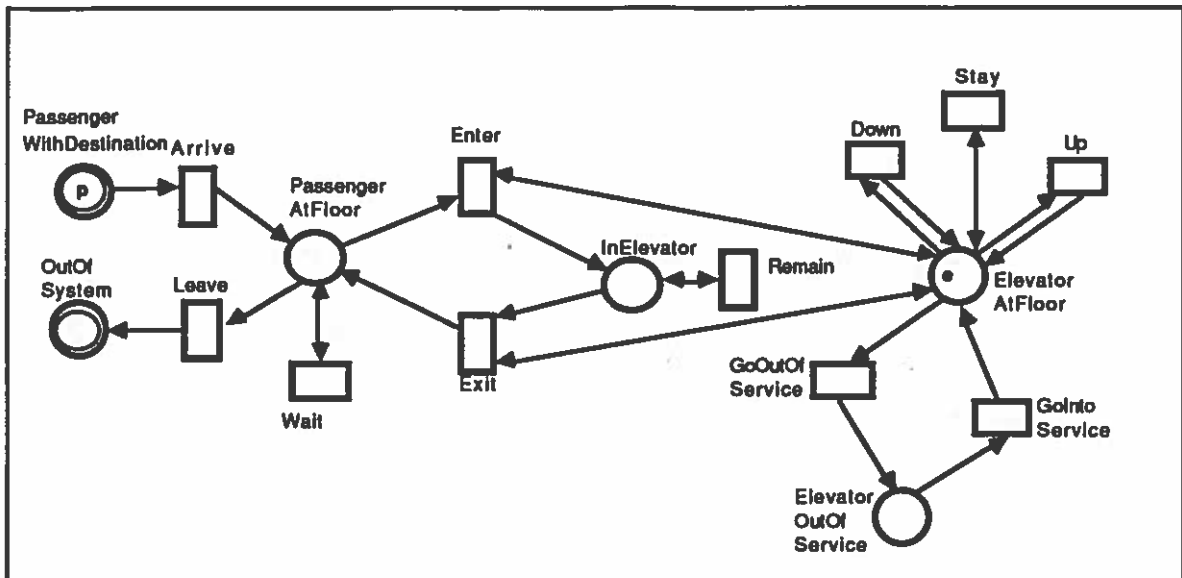
- 1) A behavior model, which describes all possible behavior of the system.
- 2) A set of system goals, which describe desired behavior of the system.
- 3) A set of agents, which control the behavior of the system

The language used for behavior models resembles that of Numerical Petri Nets (Wilbur-Ham, 1985). It extends ordinary Petri net notation in the following ways: The behavior model of the initial elevator specification discussed here appears in figure 3.

A CPN behavior model describes possible system behavior non-deterministically; it describes the actions which could occur in a particular state, without necessarily specifying how a particular action is selected. For example, figure 4 depicts the part of the behavior allowed by the net in figure 3 when the passenger on the same floor as the elevator. In this state, among other things, the passenger may enter the elevator, or wait at their current floor. We do not specify the circumstances in which the passenger will do one or the other; either event is possible. We refer to a possible sequence of events in a nondeterministic specification language as a *history* or *plan*. The set of all histories permitted by a specification gives its behavior.

The second part of the specification state is a set of system goals. Each goal defines a set of histories (sequences of states) in the behavior model which should be prohibited, or which should be achieved. The notation we use to describe histories is largely compatible with that in (Dubois, 1988). Figure 5 depicts the initial goals for the elevator example.





**Entity types**

Floor = {1..9}

**Token types**

Passenger p = (location: Floor U {"InElevator"}, destination: Floor)

Elevator e = (floor: Floor)

**Predicates**

Leave: p.location = p.destination

Enter: p.location = e.floor

Up: e.floor ≠ max Floor

Down: e.floor ≠ min Floor

**Actions**

Enter: p.location := "InElevator"

Exit: p.location := e.floor

Up: e.floor := e.floor + 1

Down: e.floor := e.floor - 1

**External constraint**

"Only one passenger at a time"

prohibit PassengerWithDestination(p1) and (PassengerAtFloor(p2) or InElevator(p2) or PassengerWithDestination(p2) and p2 ≠ p1

**Initial condition (marking)**

"Only one elevator, start at lowest floor"

ElevatorAtFloor(e) and e.floor = min Floor

**Timing Assertions**

none

Figure 3 : Possible behavior in initial development state

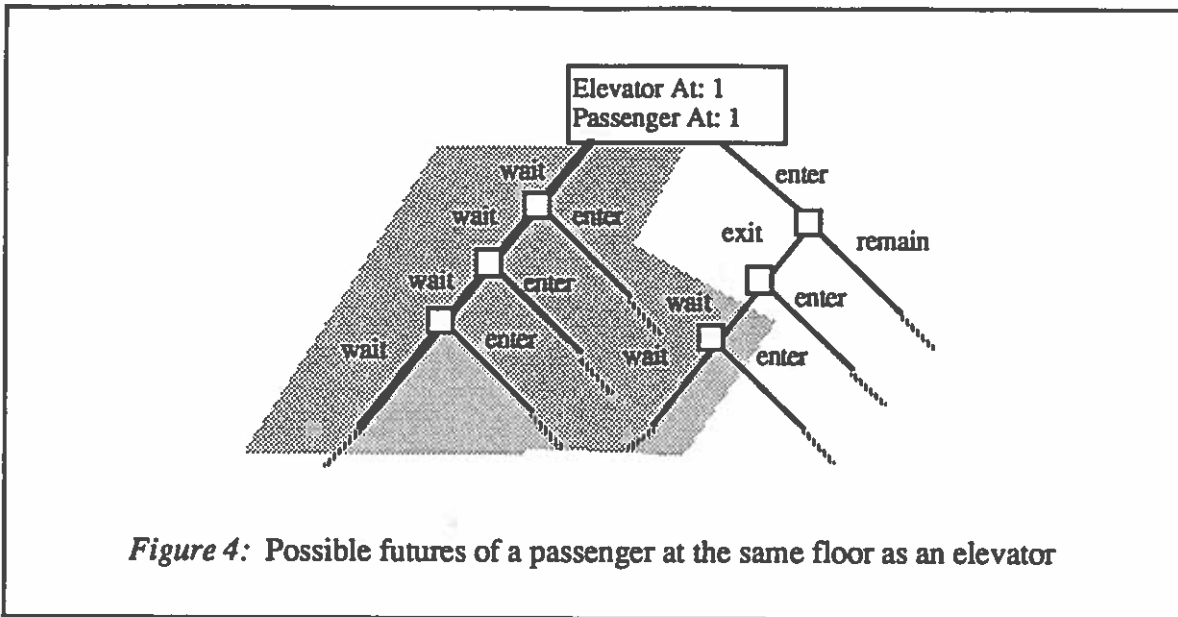


Figure 4: Possible futures of a passenger at the same floor as an elevator

A CPN behavior model is correct with respect to some system goal if its behavior always satisfies the goal. It is incorrect if the behavior includes even a single history which violates the goal. For example, the initial behavior model of figure 3 is incorrect with respect to the system goal dontLikeToWait. It allows histories such as that in figure 6, in which a passenger arrives at a floor other than the elevator's.

#### System goals

dontLikeToWait: "Don't force passenger to wait for elevator"

prohibit PassengerAtFloor(p) and not (Elevator AtFloor(e) and e.floor = p.location)

getPassengerOn: "Get waiting passenger on the elevator"

achieve InElevator(p)

getPassengerThere: "Get passenger who is in the elevator to her destination floor"

achieve InElevator(p) and ElevatorAtFloor(e) and e.floor = p.destination

getPassengerOff: "Get passenger off the elevator at destination floor"

achieve PassengerAtFloor(p) and p.location = p.destination

Figure 5 : System goals for an elevator system

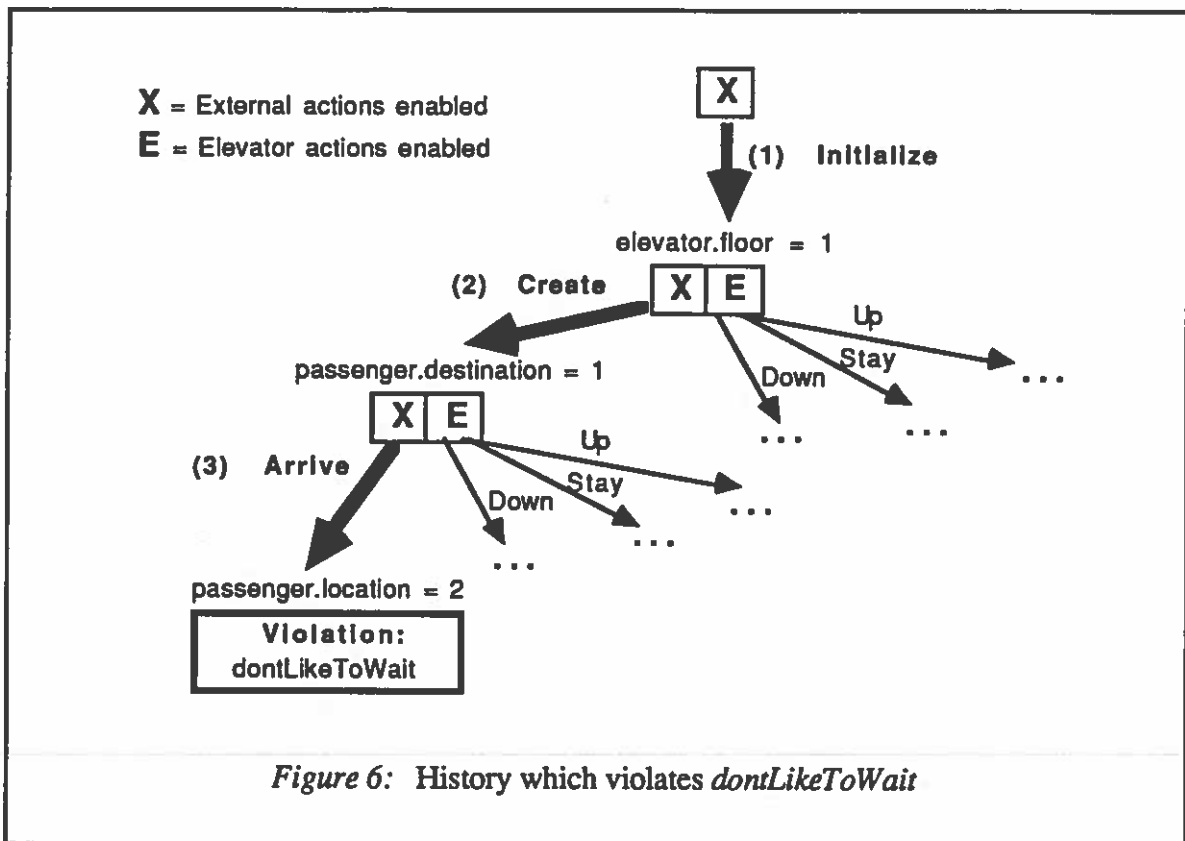


Figure 6: History which violates *dontLikeToWait*

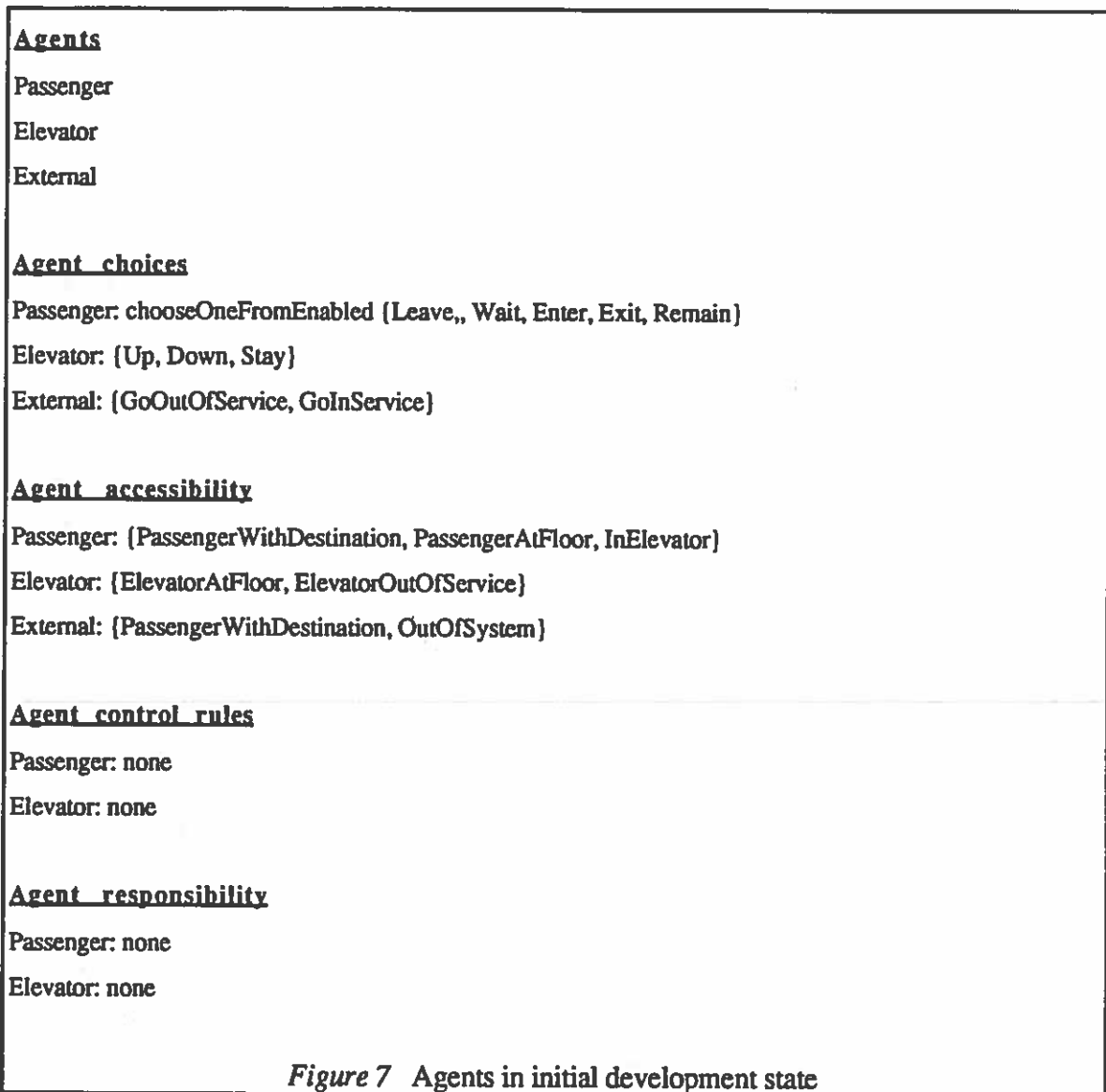
The third part of the specification state is a set of agents (figure 7). Agents are components which have the ability to sense the system's current state, and choose among some set of possible transition firings from that state. Agents can thus eliminate some or all of the possible behavior of the CPN specification. When an agent receives responsibility for a goal, it is expected to choose actions to exclude histories that violate the goal. Assume, for example, that the passenger is responsible for getting on the elevator when it is possible to do so. This requires the passenger to always enter the elevator in the state depicted, thus eliminating histories in which the passenger never enters the elevator (in the shaded area of figure 4).

Agents have the following four attributes:

1. An agent has a set of actions it controls. Actions are represented by transitions in the CPN behavior model. Each transition is controlled by at least one agent., and typically, an agent controls more than one transition. If more than one of an agent's transitions is enabled, the agent must choose one to execute, but is free to choose which, under

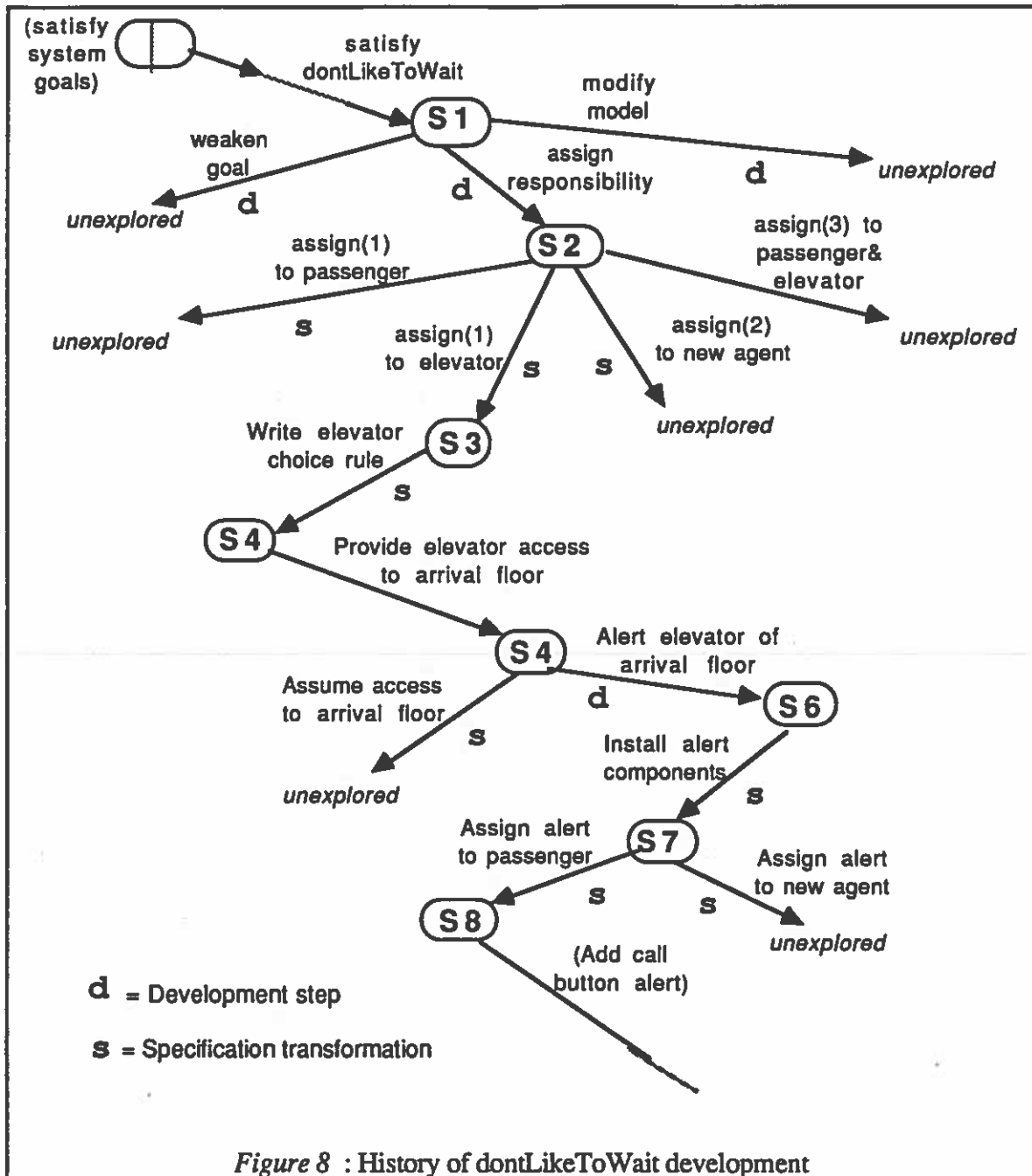
certain restrictions (see 4 below). The transitions controlled by each agent appear in figure 7 in the section labeled *Agent choices*.

2. An agent can access, or “see”, some portion of the system's state. Each place in the CPN is accessible to zero or more agents. If an agent has access to a place P, then that agent can detect what tokens are present at P at any time. This is shown in figure 7 by the section labeled *Agent accessibility*. Accessibility is used by an agent to choose “correctly” among its enabled transitions (see 4 below).
3. An agent is responsible for a set of assigned goals (found under *agent responsibility* in figure 7). The agent may share responsibility for a goal with some other agent. When we use the terms *correct action* and *choosing correctly*, we are referring to an agent choosing to carry out an action that achieves its assigned goals.
4. An agent has a set of control or choice rules. These rules describe what action an agent should choose in a given situation, in order to satisfy its assigned goals. These rules define a partial order on the set of enabled transitions of an agent in order to restrict an agent's choice to the correct one. In order to evaluate a control rule, an agent may need to read some part of the system's state which is not accessible to it. This generates a requirement for communication with another agent, or some other information-gathering activity. Thus, control rules derive many of the requirements that are unique to composite systems.



## 2.1 Deriving specification features from a system goal.

Figure 8 shows our example development based on (Doerry et. al., 1990). We present only the initial portion of this development in detail, to illustrate some key issues. Figure 9 shows the task tree resulting from the first phase of development, up to state S3.



### State S1

*Focus.* The initial task in the task tree is satisfy system goals. In the interest of space, we will assume that this initial task has been refined to satisfy dontLikeToWait, and that this task is our focus.

Critter first determines whether the task is satisfied, i.e., is the system goal "don't-like-to-wait" achieved in the current specification state. A planner is called to find a counterexample to show that it is not achieved. The planner produces the counterexample shown graphically in figure 6; in summary, the counterexample runs as follows:

- 1) The External agent initializes the system by placing the elevator at floor 1.
- 2) The External agent creates a passenger with a destination of floor 1.
- 3) The External agent then places this passenger at floor 2.

*Propose.* Three methods are proposed to implement the satisfy dontLikeToWait task:

- Method 1: weaken the goal (don't-like-to-wait) so that it is satisfied in the current state.
- Method 2: modify the model so that the goal is satisfied.
- Method 3: assign responsibility to one or more agents for the goal, thus forcing agents to cooperate in achieving the goal.

*Evaluate.* Some general rules of thumb here are that weakening a goal will lose functionality and convenience, but at the same time may be the most feasible. Modifications to the model or assignment to agents have highly domain specific costs that are hard to generalize at this high level in the development state.

*Select.* At this level, we rely on the human analyst to make the choice. For our example, she will choose method 3, assigning responsibility for don't-like-to-wait to one or more agents. If this turns out to be a bad choice, the analyst can return to this state and explore either of the other two alternative methods at a later time: Critter records the choices and evaluation in a way that allows any alternative to be explored.

The selection of the assign responsibility method produces a new task, **assign responsibility** for the goal don't-like-to-wait. This change to the task tree generates a new state S2.

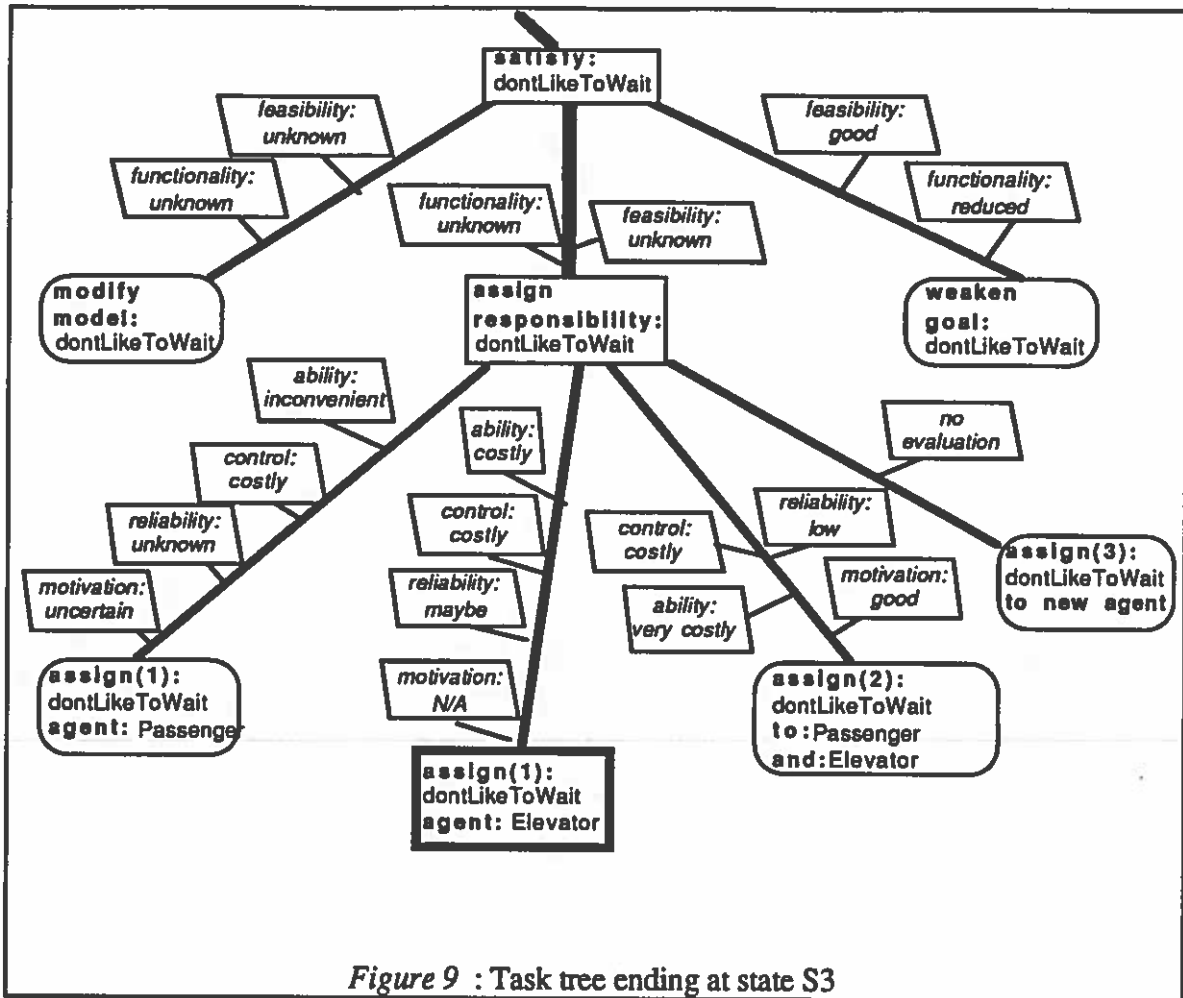


Figure 9 : Task tree ending at state S3

## State S2

*Focus.* For simplicity, we will assume that the focus will follow a depth-first traversal of the task tree, as shown in figure 9. Using this strategy, the focus is the newly generated task "assign responsibility for goal don't-like-to-wait".

*Propose.* Three methods are found that implement the assign responsibility task:

Method 1: Assign responsibility for goal don't-like-to-wait to a single agent A. From this method, two choices are generated: assign don't-like-to-wait to the passenger and assign don't-like-to-wait to the elevator.

Method 2: Assign responsibility for goal don't-like-to-wait to a newly created agent N. From this method, one choice is generated: assign don't-like-to-wait to a new but yet undetermined agent. That is, add a new agent to the system and then assign the goal to it.



**Method 3:** Assign responsibility for goal don't-like-to-wait to a combination of agents, i.e., force them to cooperate to achieve the goal. From this method, one choice is generated: assign don't-like-to-wait to the passenger and elevator, jointly.

*Evaluate.* There is a standard set of criteria that is used for evaluating responsibility assignment methods in Critter:

Ability: Is the proposed agent capable of achieving the goal? Does it have the right set of actions for carrying out a set of steps that will lead to goal achievement? If not, what is the cost of adding those actions to the agent?

Control: If the agent has (or is given) the right set of actions, what needs to be added to the agent to allow it to act "responsibly"? That is, what actions must the agent take to ensure that the goal is never violated? What information does the agent need in order to decide when to take those actions? What is the cost of providing that information to the agent?

Reliability: If the agent has both the right set of actions and the necessary information for selecting among them, is the agent reliable? Will it ever "forget" or "lose" the information, or sometimes be unable to carry out one of its actions? What is the cost of addressing the given reliability problems?

Motivation: If the agent has both the right set of actions and the necessary information for selecting among them, is the agent motivated? Does it have "local" goals that conflict with the system-wide goal that is to be assigned? If so, can we expect the agent to act "irresponsibly" in some situations? What is the cost of addressing motivation problems?

These questions are at the heart of our research effort. Our goal is to call upon a mixture of formal and semi-formal cost models in conjunction with human assistance to produce an evaluation of the three responsibility assignment methods. To give the reader a feel for what is involved in this effort, we will reason informally about the choices we are faced with in state S2.

Assignment to the passenger: The passenger agent, alone, does not have the ability to achieve the goal. It does not control the create and arrive actions, both of which are under control of the external agent. A potential modification is to give the passenger an action that

allows it to move between floors other than by the elevator (e.g., stairs). In this way the passenger can move to the floor that the elevator is on, and board immediately.

Following this a step further, the passenger will need access to information it currently does not have: the location of the elevator. Given this, it can restrict its actions to climb (or descend) to the elevator floor, and thus guarantee the don't-like-to-wait goal. A potential source of the needed information is the elevator, itself. Perhaps it can signal the passenger of its whereabouts in some way.

Even with these changes, can we rely on the passenger to carry out its actions? If movement is by stairs, are there circumstances where this movement is hindered, e.g., passengers in wheelchairs? Must we consider alternative or back up actions for these passengers?

Finally, are there any "selfish" goals a passenger might have that would interfere with its responsibility of meeting the don't-like-to-wait goal? It is difficult to imagine such a conflict in our simple system. In a more complex system, one with multiple passengers and elevators, it is likely that selfish or even mischevious behavior might occur that conflicts with the good of the whole. Can we either influence or supress such behavior?

Assignment to the elevator: The elevator has the ability to satisfy the goal don't-like-to-wait, by moving to the floor where the passenger will arrive before the passenger gets there. However, this implies that the elevator's Up/Down transitions should always be able to "outrance" the Arrive transition. This requires a fast elevator, and brings higher construction and maintenance costs.

In addition, the elevator needs access to the floor number where the passenger intends to arrive, in order to move to the proper floor. It might be costly to supply this information to the elevator; the passenger might require a personal console, or possibly a pocket pager, by which it could signal the elevator and dispatch it to a floor.

We must also consider the reliability of the elevator. Upon mechanical failure, the elevator may be unable to move to a floor quickly enough to satisfy the goal, or may fail to move at all. The elevator may also misread or misinterpret its instructions from the passenger, and move to the wrong floor. We must consider the cost of constructing and maintaining an elevator at the level of availability implied by the goal.

As a mechanical agent, the elevator lacks any motivation as such, but if a human operator controls its motion, we must evaluate the cost of preventing mischief or selfish behavior by the operator.

Assignment to the elevator and the passenger:

Critter also must evaluate possible divisions of responsibility between the two agents. For instance, we may require the passenger to move to the elevator if the elevator is on an adjacent floor, while requiring the elevator to close the distance in other cases.

As before, the cost of giving an agent the ability to perform its responsibilities can help choose among these alternative divisions. Critter would need to relate the distance moved by the passenger to the cost of architectural features (such as stairs or escalators) to support that movement. In addition, if the elevator only moves longer distances to meet the passenger, it will make fewer stops and starts, lowering maintenance costs and failure probabilities.

Assigning multiple agents to a goal also complicates the control requirements. For instance, if both elevator and passenger might move, each needs to know the location of the other in order to meet. In addition to increasing the cost of communication between the agents, this also requires more sophisticated decision-making circuits in the elevator, which might have higher costs in some implementations.

Assigning a goal to multiple agents typically requires more complex coordination between the agent's actions, which may be a source of unreliability. If only the elevator can move between floors prior to passenger arrival, its actions do not depend on those of the passenger, allowing a simpler control regime. In addition, if both agents have been assigned partial responsibility for a goal, both must operate reliably; assuring the reliability of both agents may cost more than assuring each of them separately.

Divisions of responsibility may also be more susceptible to sabotage by unmotivated agents. For instance a passenger refusing to move to the elevator might deadlock the system, as the elevator waited for the passenger and the passenger waited for the elevator.

Assignment to a new agent. Finally, we might consider introducing a new agent to guarantee the goal. Such an agent would have to move either the elevator, the passenger, or both of these other agents to the same floor.

It is difficult to evaluate this possibility at a high level. Initially, the new agent would have no ability to complete its task, but at a minimum, it would require the ability to transport one of the other agents between floors. Adding operators to do so might lead to conflicts between agents, in which, for example, the new agent tries to move the elevator up while the elevator attempts to move itself down. The new agent would also initially lack access to information (such as the passenger's location) required to satisfy the goal. Until more is specified for the new agent it is not possible to determine the cost of its introduction.

*Select.* Selection is deficiency-driven: The system must identify the least costly alternative, based on estimates of the cost of overcoming deficiencies in each agent's abilities, control, reliability and motivation. If we are required to add an action to an agent, which is the least costly alternative? If we need to supply an agent with more information than it currently has, what is the relative cost? If we must build back-up systems for unreliable agents, what is the cost? If we must build carrot-and-stick mechanisms to motivate agents, what is the cost?

There is no one model that has been proposed for answering such questions. However, design disciplines outside of software engineering use diverse cost models that may answer specific questions. Construction estimation models and catalogs of elevator components are employed to answer questions on construction cost. Quality assurance supplies analytic and simulation methods to evaluate designs of mechanical systems for reliability. Human factors research has produced models and methods for estimating human reliability within a system. Critter needs models like these in order to partially automate the evaluation process.

Continuing with our example, we will select the elevator as the agent to make responsible for the don't-like-to-wait goal. A new task is posted -- assign don't-like-to-wait to the elevator -- and a system goal is added to the elevator agent's list of responsibilities. This leads to a change in both the specification state and the development tree.

Figure 10 previews the task tree from the next phase of the development (states S3 through S8).

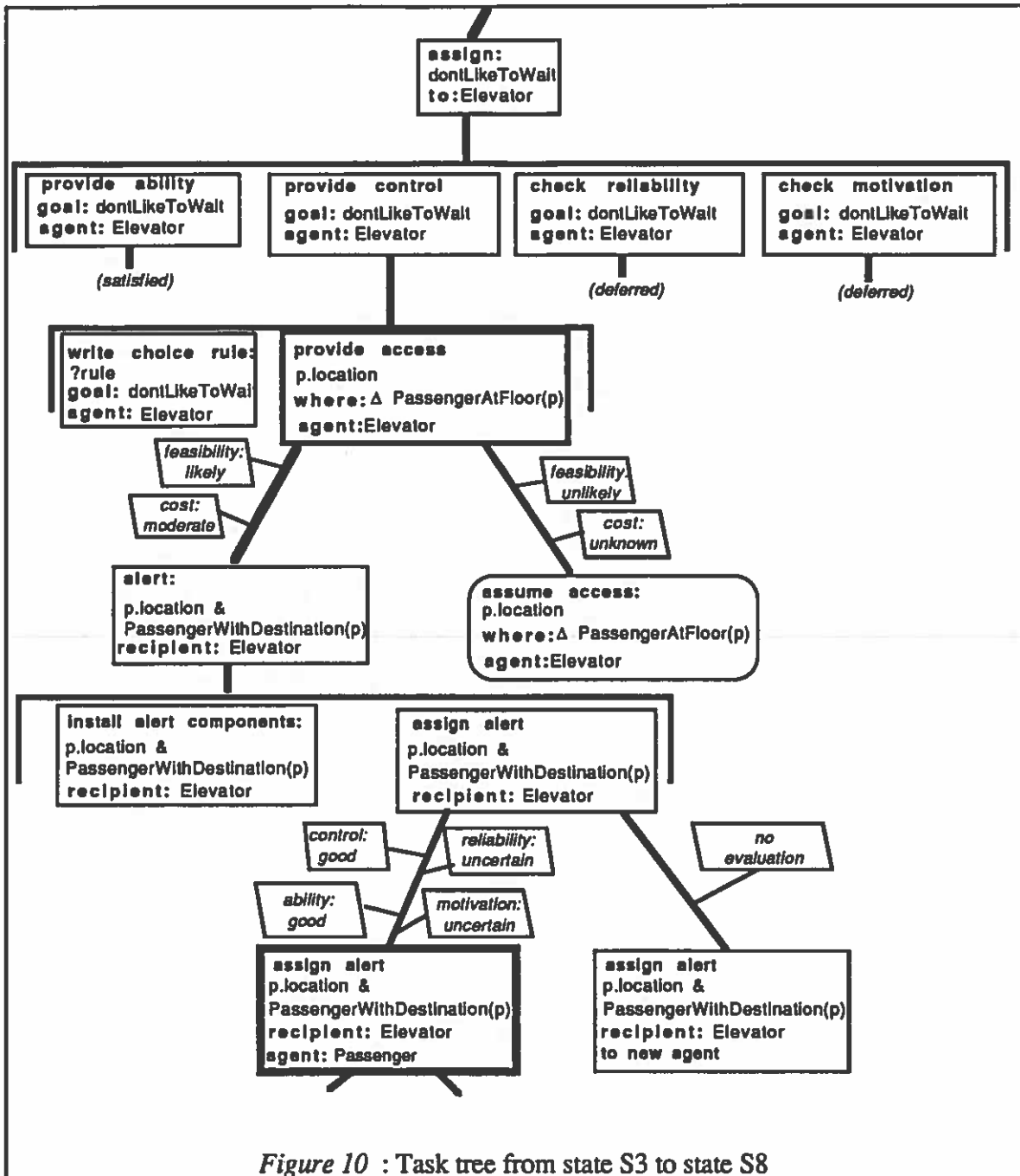


Figure 10 : Task tree from state S3 to state S8

### State S3.

*Focus.* The system focuses on the most recently posted task, assign don't-like-to-wait to the elevator .

*Propose.* A single method is found, one that refines the task into four sub-tasks: 1) give elevator the ability, 2) give elevator the necessary information to control its actions, 3) address reliability issues, and 4) address motivational issues.

*Evaluation.* This is the only problem reduction step available.

*Select.* The method is selected and applied, adding the four sub-tasks above to the task tree.

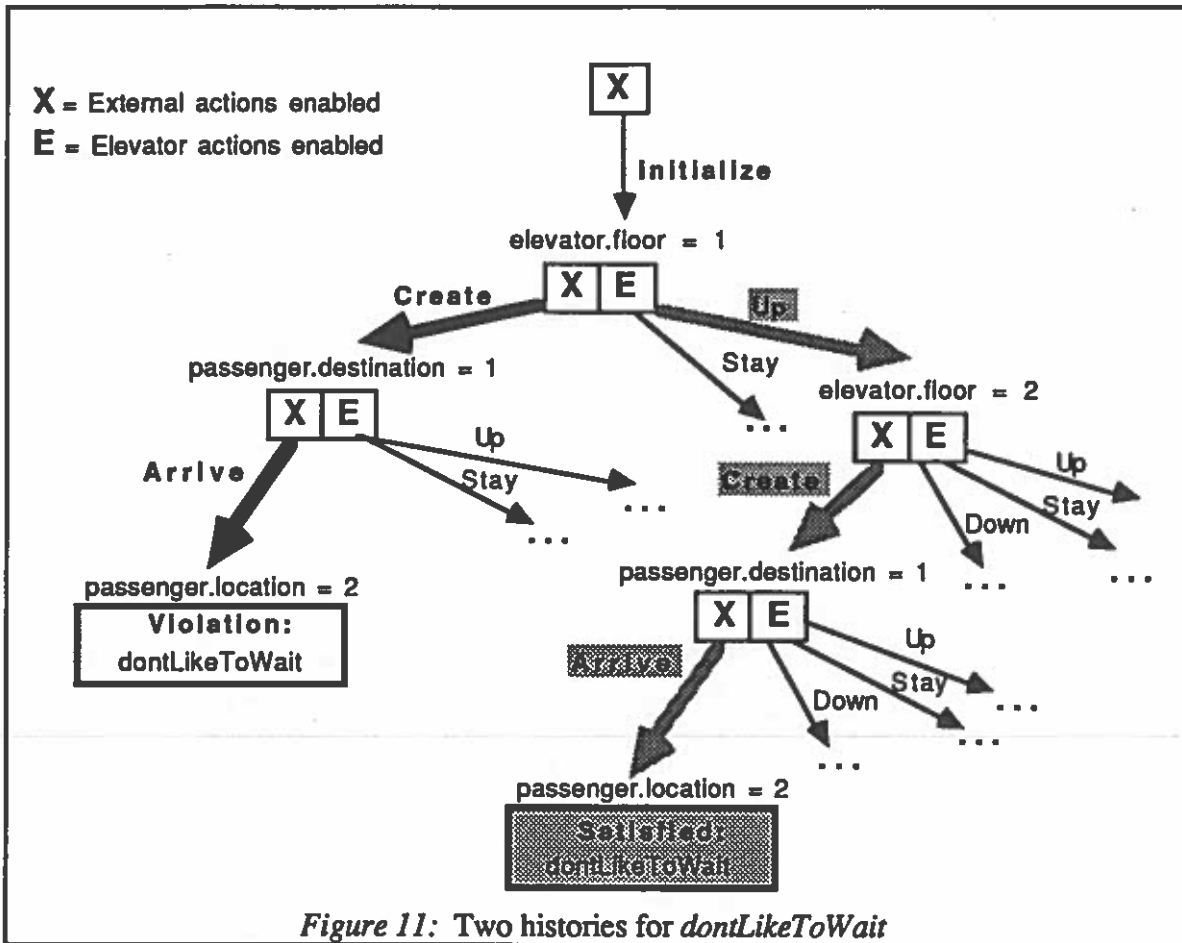
#### State S4.

*Focus.* Looking first at the task of giving the elevator the ability to achieve `don'tLikeToWait`, the analyst calls on the planner to find at least one plan in which the elevator acts to satisfy it. The planner returns the plan shown in figure 11 (gray line). In the plan, the elevator moves up one floor prior to the arrival of the passenger on the second floor. The analyst determines from this plan that the elevator contains the actions it needs to satisfy the goal without further modifications.

We next focus on the task of giving the elevator the necessary information to control its action to carry out the plan. The task is not completed (i. e., the elevator does not have the necessary control mechanism), so we must further refine the development state.

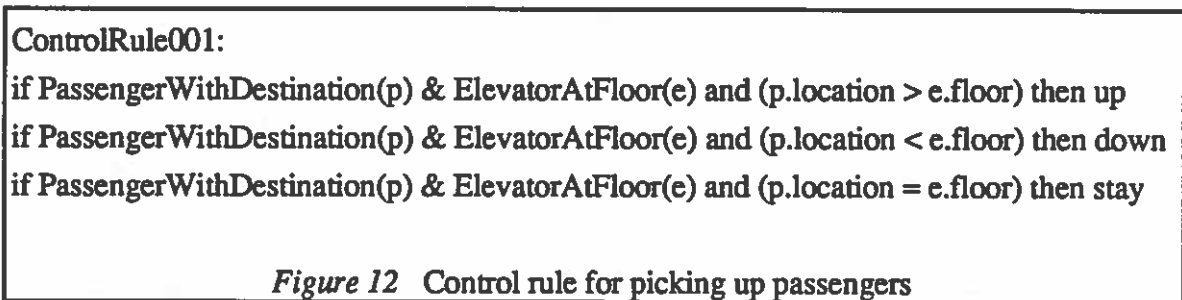
*Propose.* There is a single method available for this task: add one or more control rules that limit the agent's actions. Using the plan generated in the focus step as a guide, the analyst notes that the elevator could guarantee the goal if it had control rules such as that in figure 12.

The task becomes **provide access** to the passenger's arrival floor.



*Evaluation.* write choice rule is the only method available.

*Selection.* The method is selected, adding the provide access task to the task tree.



State S5.

*Focus.* The task we focus on is provide access to the passenger's arrival floor.

*Propose.* There are two methods available:

Method 1: find a place P where the information is available in the current model, and add P to the elevator's list of accessible places.

Method 2: use an alert template to make the information known in a new place Q, and give the elevator access to Q.

*Evaluation.* This is a clear example of the importance of strong evaluation techniques at high levels of specification design. It might appear that method 1 is the trivially correct method - it simply involves adding an existing place to those accessible to an agent. However, we must somehow map the cost of purely syntactic, box-and-pointer modifications in the CPN to real world costs. In this case, how would the implementors of the CSS actually build a mechanism that allows the elevator to learn a passenger's arrival floor before the passenger arrives and without any assistance from the passenger? A passenger arrival schedule? Mind probes? Passive sensors? There seems no easy way to implement this approach.

On the other hand, the second method embodies a standard means of dealing with "future reference", i.e., the problem of one agent needing to predict a future event so that it can act accordingly. The alert method is one that makes a future event known. It adds a new transition that allows an agent to communicate its intentions in advance to another agent. This has known implementation costs in most cases.

*Selection.* In the end, we are dealing with a method with no known low-cost implementation versus a method with known moderate-cost implementations. In Critter, we could explore the first, risky method a bit before returning to the safer method. However, we will choose the safe route to finish up our example.

### State S6.

We will terminate the detailed reenactment of the elevator design here and summarize the remaining steps solving `dontLikeToWait`.



Following the alert method<sup>1</sup> (figure 13), Critter must install its components (changing the specification to produce state S7), and select an agent to be responsible for carrying out the alert action. The passenger agent is chosen. Thus, following figure 13, a new transition T is added to the passenger agent's capabilities, and a new place P2 is made accessible to the elevator. The passenger's arrival floor becomes I, the information to be exchanged. This leaves the development at state S8.

**Method:** alert

**Description:** Give Agent A access to information I needed to evaluate control rule R

- Tasks:**
- 1) Install alert template components at P1, the source of I
  - 2) Assign responsibility: make information I available at P2 when R applies.

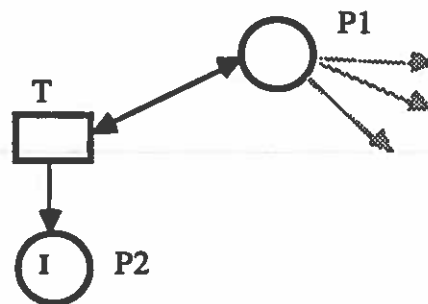


Figure 13: Alert method and components

We must guarantee the passenger will actually carry out T at the appropriate time. Further, we must consider the reliability of the passenger agent for alerting the elevator: will passengers forget to alert the elevator prior to arrival? The problem is addressed by adding a redundant component; specifically, an additional alert template is installed which the passenger can operate upon arrival at any floor. In effect, this specifies a "call button" for each floor, but whether this is an actual button or some other sensor is an implementation issue - the CSS will simply note that the passenger can alert the elevator of its arrival location by some means.

---

<sup>1</sup> The template shown is not complete: Agent x removes the alert when it is no longer needed (when the passenger gets on). We will omit these clean-up components here and in the future alerting examples for readability.

To close out this example, we suggest how the other three system goals (listed in figure 5) might be handled in this development:

- *getPassengerOn*. The goal is not guaranteed. Assigning this goal to any agent other than the passenger would require that agent to physically move the passenger into the elevator. This might be an interesting solution in some contexts (for a physically challenged passenger, for instance), but is costly in general. The passenger is judged capable of entering the elevator, so a control rule is added requiring the passenger to enter the elevator when it is possible to do so. It is decided that the passenger cannot always carry out this rule reliably -- the passenger might be distracted and fail to notice the arrival of the elevator, for instance -- so an additional alert is added which notifies the passenger when the elevator reaches the the passenger's floor. This alert is assigned to the elevator.
- *getPassengerThere*. The goal is not guaranteed. It is assigned to the elevator, as the elevator controls the necessary actions and other alternatives (such as allowing passengers to move between floors independently) are judged too costly. As with *dontLikeToWait*, the elevator must choose among going up, going down, or staying put at each point; it receives a control rule similar to that in figure 12. The passenger is assigned responsibility for an alert which communicates the passenger's destination floor to the elevator.
- *getPassengerOff*. The goal is not guaranteed. Following reasoning similar to that for *getPassengerOn*, the passenger is assigned responsibility to move out of the elevator and given a control rule to do so. This requires an additional alert which tells the passenger when the destination floor has been reached.

### 3.0 Conclusions

We have described an architecture for the transformational development of composite system specifications. The system uses an abstract library of transformations tailored for composite systems, only some of which are correctness-preserving. Transformations are selected to apply based on human input, and on domain-specific knowledge in the form of evaluation critics. We have attempted to identify the kinds of domain-specific knowledge the evaluation critics would need to do informed selection.

Many of these evaluation critics will use representations which are less formal and less general than that of the specification itself. A construction cost estimation model for

elevators, for instance, would ultimately have to rely on the costs of actual elevator components, not on an abstract theory. In some domains, such as nuclear reactor operation, empirical studies may supply tables of human response times and failure rates when performing very specific tasks, comparable to tables of component costs (Meister, 1985). In others, we may only have a description of actual failures brought on by specific misfeatures of a particular design. For instance, neither a detailed theory or human action, nor empirical data are available to evaluate elevator designs for their resistance to pranks such as elevator surfing<sup>2</sup>. Instead we may need to rely on descriptions of cases where a particular combination of elevator features has allowed a prank to occur. An earlier specification critic (Fickas and Nagarajan, 1987) illustrated this case-based approach.

A question of interest is how to combine the results of evaluation critics with diverse representations to perform method selection. (Weld, 1990) has considered the problem of combining multiple qualitative process and differential equation models of a system for behavior explanation and prediction. (Farley & Liu, 1990) have also demonstrated a system for qualitative explanation of electronic circuits, using two qualitative process models at different levels of detail. (Rissland, 1988) also describes an architecture combining rule-based and case-based knowledge bases in legal reasoning. Research in distributed problem-solving (Bond & Gasser, 1988) has addressed the problem of integrating knowledge from diverse, uncertain sources. Research in decision theory provides analytic methods for selection based on multiple evaluation criteria (Zeleny, 1982).

Our research now focuses on locating and integrating diverse evaluation models, both formal and informal, in order to realize the Critter approach in a useful computer-based tool for CSS applications.

## References:

Anderson, J. S. & Fickas, S. (1989). A Proposed Perspective Shift: Viewing Specification Design as a Planning Problem. *Proceedings: Fifth International Workshop on Software Specification and Design* (Pittsburgh, PA). ACM SIGSOFT Engineering Notes, Volume 14, Number 3 (May 1989).

---

<sup>2</sup> Riding atop a moving elevator, either by forcing open the outer elevator doors on an upper floor and jumping on, or by climbing through a ceiling service hatch.

- Bond, Alan H. and Gasser, Les (1988). *Readings in Distributed Artificial Intelligence*. San Mateo, CA: Morgan Kauffman, 1988.
- Doerry, E, Feather, M., Fickas, S., Helm, R.. (1990). Deriving User Interface Requirements through Composite System Design, University of Oregon, Department of Computer and Information Sciences Technical Report, to appear.
- Dubois, E.. (1988). A Logic of Action for Goal-Oriented Elaboration of Requirements,, *Proceedings of the Fifth International Workshop on Software Specification and Design*.,
- Farley, A. M., Liu, Z. Y. (1990). Shifting Ontological Perspectives in Reasoning about Physical Systems, *Proceedings of the 1990 AAAI Conference*, Boston.
- Feather, M. S. (1987). Language Support for the Specification and Development of Composite Systems. *ACM Transactions on Programming Languages and Systems*, 9(2), 198-234.
- Fickas, S. (1983). Automating the Transformational Development of Software (volume 1). University of Southern California/Information Sciences Institute, Research report RR-83-108.
- Fickas, S. (1985). Automating the Transformational Development of Software. *IEEE Transactions of Software Engineering*, 11(11), 1268-1277.
- Fickas, S., Nagarajan, P. (1988). Being suspicious: critiquing problem specifications, In *Proceedings of the 1988 AAAI Conference*, Minneapolis.
- Meister, David (1985). *Behavioral Analysis and Measurement Methods* . New York: John Wiley and Sons, 1985.
- Rissland, Edwina L. & Skalek, David B. (1988). Interpreting Statutory Predicates, In *The Second International Conference on Artificial Intelligence and Law: Proceedings of the Conference* (Vancouver, BC). New York: Association for Computing Machinery, 1988.
- Weld, Daniel S. (1989). Automated Model Switching: Discrepancy-Driven Selection of Approximation Reformulations. University of Washington (Seattle, WA), Department of Computer Science and Engineering, Technical Report 89-08-01.
- Wilbur-Ham, M. C. (1985). Numerical Petri Nets - A Guide. Telecom Australia Research Laboratories, Report 7791.
- Zeleny, Milan (1982) *Multiple Criteria Decision Making*. New York: McGraw Hill, 1982.