

Using Behavior Trees to Model the Autonomous Shuttle System

R.Geoff. Dromey,

Software Quality Institute, Griffith University,
Nathan, Brisbane, Qld., 4111, AUSTRALIA
g.dromey@griffith.edu.au

Abstract

The requirements for problems like the Autonomous Shuttle Transport System are relatively easy to state informally and loosely in natural language. The question is how to use this information effectively to guide the development of a system that will satisfy the original intent, clarify the original intent where necessary, identify defects in stated requirements and at the same time manage the complexity of the problem. Progress is possible once we recognize that individual functional requirements represent fragments of behaviour, while a design that *satisfies* a set of functional requirements represents integrated behaviour. This perspective admits the prospect of constructing a design *out of* its requirements. A formal representation for individual functional requirements, called *behavior trees* makes this possible. Behavior trees, derived by rigorous word-by-word translation from individual functional requirements stated in natural language, may be composed, one at a time, to create an integrated design behavior tree (DBT). When we apply translation and integration for the original set of high-level requirements for the Shuttle System we discover they are behaviourally incomplete. Integration constructively forces us to confront and resolve the missing requirements problems at the earliest possible time in the development phase. Once this is done we should have a complete, fully integrated DBT. We can then transition from this *problem domain* representation directly and systematically to a *solution domain* representation of the component architecture of the system and the behaviour designs of the individual components that make up the system – both are emergent properties of a DBT.

1. Introduction

The Autonomous Shuttle Transport System is typical of many problems that are relatively easy to state informally and loosely in natural language. Such problem statements often have two significant characteristics: they imply a lot more than they state and they contain defects that can significantly impact subsequent design efforts. Confronted with these challenges, existing methods for requirements analysis [2], representation and then design usually opt for producing multiple partial views of a system. Our position is that the multiple partial views approaches usually make it difficult to see many types of defects, particularly those that involve interactions between requirements [1,4,6]. Given this, is there a more practical way forward – we suggest there is using a single integrated view. The challenges we must confront are:

- how to get on top of requirements complexity,
- how to preserve the intention of the stakeholders, and
- how to detect requirements defects early as possible.

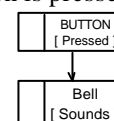
We suggest there is a way to deliver these benefits and consistently make real progress with the requirements problem. It demands that we use the requirements of a system in a very different way. Traditionally the goal of systems development is to build a system that will *satisfy* the agreed requirements. We suggest this task is too hard, particularly if there is a large and complex set of requirements for a system. *A much simpler and easier task is to seek to build a system out of its requirements.* If we opt to do this it implies two things:

- we have a representation that will formally represent the behaviour in individual requirements
- we have a way of combining/integrating individual requirements to create a system that will satisfy all requirements.

Behavior Trees handle both of these needs.

2. Behavior Trees

The Behavior Tree Notation captures in a simple tree-like form of composed component-states what usually needs to be expressed in a mix of other notations. Behavior is expressed primarily in terms of components realizing [State], ??Event??. ?Decision?, <Data_Out>, >Data_In< component attribute assignment “:=”, and reversion “^” to an equivalent component-state mentioned higher up in the tree, augmented by the logic and graphic forms of conventions found in programming languages to support composition. The vital question that needs to be settled, if we are to build a system out of its requirements, is can the same formal representation of behaviour be used for requirements and for a design? Behavior trees make this possible, and as a consequence, clarify the requirements-design relationship. Behavior trees provide a direct and clearly traceable relationship between what is expressed in the natural language representation and its formal specification. Translation is carried out on a sentence-by-sentence, word-by-word basis, e.g., the sentence “the bell sounds when the button is pressed” is translated to:



3. Genetic Design

Conventional software engineering applies the underlying design strategy of constructing a design that will *satisfy* its set of functional requirements. In contrast to this, a clear advantage of the behavior tree notation is that it allows us to construct a design *out of* its set of functional requirements, by integrating the behavior trees for individual functional requirements (RBTs), one-at-a-time, into an evolving design behavior tree (DBT) [3]. This very significantly reduces the complexity of the design process and any subsequent change process. What we are suggesting is that a set of functional requirements, represented as behavior trees, in principal at least (when they form a complete and consistent set), contains enough information to allow their composition. This property is the exact same property that a set of pieces for a jigsaw puzzle and a set of genes possess [5]. The obvious question that follows is: “what information is possessed by a set of functional requirements that might allow their composition or integration?” The answer follows from the observation that the behaviour expressed in functional requirements does not “just happen”. There is always a *precondition* that must be satisfied in order for the behaviour encapsulated in a functional requirement to be accessible or applicable or executable. We call this requirement of genetic design, the *precondition axiom*.

Precondition Axiom

Every constructive, implementable individual functional requirement of a system, expressed as a behavior tree, has associated with it a precondition that needs to be satisfied in order for the behavior encapsulated in the functional requirement to be applicable.

A second building block is needed to facilitate the composition of functional requirements expressed as behavior trees. Jigsaw puzzles, together with the precondition axiom, give us the clues as to what additional information is needed to achieve integration. With a jigsaw puzzle, what is key, is not the order in which we put the pieces together, but rather the *position* where we put each piece. If we are to integrate behavior trees in any order, one at a time, an analogous requirement is needed. We have already said that a functional requirement’s precondition needs to be satisfied in order for its behaviour to be applicable. It follows that some *other* requirement, as part of its behavior tree, must establish the precondition. This requirement for integrating functional requirements expressed as behavior trees is expressed as follows.

Interaction Axiom

For each individual functional requirement of a system, expressed as a behavior tree, the precondition it needs to have satisfied in order to exhibit its encapsulated behavior, must be established by the behavior tree of at

least one other functional requirement that belongs to the set of functional requirements of the system.

The precondition axiom and the interaction axiom play a central role in defining the relationship between a set of functional requirements for a system and the corresponding design. What they tell us is that in the first stage of the design process, in the problem domain, we can proceed by first translating each individual natural language representation of a functional requirement into one or more behavior trees. We may then proceed to integrate those behavior trees just as we would with a set of jigsaw puzzle pieces. What we find when we pursue this whole approach to software design is that the process can be reduced to the following four overarching steps:

- Requirements translation – (problem domain)
- Requirements integration – (problem domain)
- Component architecture transformation (solution domain)
- Component behavior projection (solution domain)

Each overarching step, needs to be augmented with a verification and refinement step designed specifically to isolate and correct each class of defects that show up in the different work products generated by the process. Because of space limitations here we only have room to show the results of translating then integrating the seven originally stated functional requirements for the Shuttle System (see below). We will also provide some commentary on the integration step. Elsewhere each of the steps in the process is described in more detail [3]

Table 1. Autonomous Shuttle Functional Requirements

S1. Shuttles are being assigned orders to transport goods between certain stations.
S2. Successful completion of an order results in monetary reward for the shuttle involved.
S3. In the case where an order has not been completed in a given amount of time, a penalty is incurred.
S4. New orders are made known to all shuttles, thus all shuttles can make an offer.
S5. The shuttle with the best offer, i.e., lowest offer will receive the assignment.
S6. Using the tracks will receive a toll depending on the distance covered.
S7. Maintenance of shuttles is possible at any station and will cost both time and money.
Ref. Graduate School of Dynamic Intelligent Systems, University of Paderborn

3.1 Requirements Translation

Requirements translation is the first formal step in the Genetic Design process. Its purpose is to translate each natural language functional requirement, one at a time, into one or more behavior trees. Translation identifies the *components* (including actors and users), the *states* they realise (including attribute assignments), the *events* and

decisions/constraints that they are associated with, the data components exchange, and the causal, logical and temporal dependencies associated with component interactions. When requirements translation has been completed each individual functional requirement is translated to one or more corresponding requirements behavior tree(s) (RBTs). Below in figure 1 we show the “raw” annotated translation for sentence S4 from the original statement of requirements in Table 1.

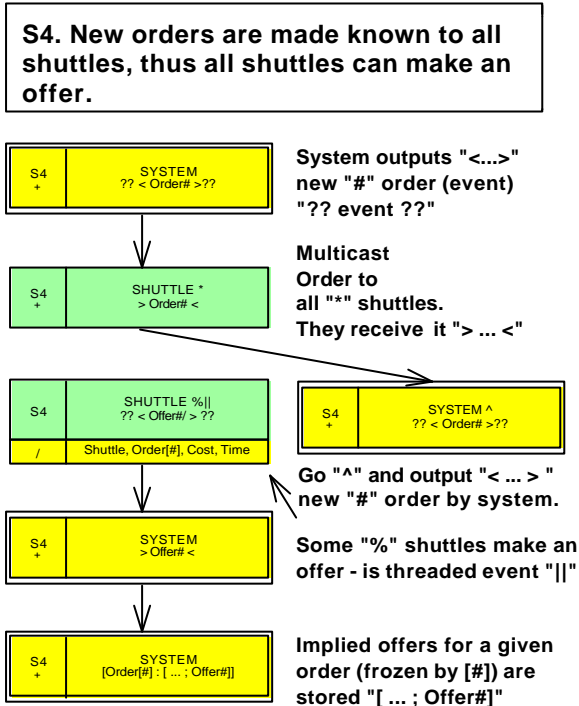


Figure 1. RBT for requirement S4

3.2 Requirements Integration

We can next systematically and incrementally construct a design behaviour tree (DBT) that will satisfy all its requirements *by integrating the individual requirements' behavior trees* (RBT) one at a time. Integrating two behavior trees turns out to be a relatively simple process that is guided by the precondition and interaction axioms referred to above. In practice, it most often involves locating where, (if at all) the component/state root node of one behavior tree occurs in another tree and grafting the two trees together at that point. This process generalises when we need to integrate N behavior trees. We only ever attempt to integrate two behavior trees at a time. In some cases, because the precondition for executing the behavior in an RBT has not been included, or important behaviour has been left out of a requirement, it is not clear where a requirement integrates into the design. This immediately points to a problem with the requirements. In other cases, there may be requirements/behaviour missing from the set which prevents integration of a requirement. Attempts at

integration uncover such problems with requirements at the earliest possible time. Consider the case of integrating S4 with S5 given below - there is an integration problem.

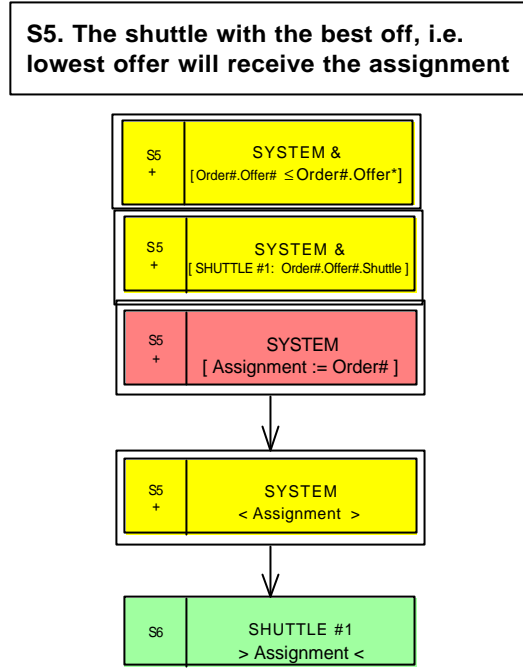


Figure 2. RBT for requirement S5

S4 tells us that “all shuttles can make an offer” while S5 tells us that the shuttle with “the lowest offer will receive the assignment”. Clearly the lowest offer can only be worked out after a set of offers is received and compared. There is however nothing in S4 or any of the other six requirements about when the shuttles finish making an offer for a particular order. Therefore, to achieve integration of S4 and S5 we need to include behaviour that effectively kills off the set of thread events that correspond to shuttles making an offer at random relative times for a particular order. This then becomes the precondition for integrating S5 and thereby selecting the lowest offer for a particular order. The result of including this additional behaviour, that makes integration of S4 and S5 possible and carrying out the integration of these two requirements, along with all the other integrations is shown in figures 6 and 7 at the end of the paper.

There is other important missing behaviour that involves S4 and S5. Because “all shuttles can make an offer”, it follows that a shuttle can make an offer and possibly be assigned an order when it is in the middle of delivering another order. This implies three things (1) that when a shuttle is assigned an order, if it is not in the process of delivering another order it can proceed straight away to pick up and deliver the order. (2) However, if the shuttle is currently delivering an order it will need to store details of the order for later processing. (3) If an order needs to

stored then we also need to include behaviour that checks for any stored orders once a delivery is completed. If such an order (or orders) exists it needs to be retrieved. The Shuttle can then “reuse”, by reversion “^”, the prescribed behavior for delivering an order. The necessary behavior to carry out these additional tasks has been factored into the integrated DBT shown in figures 6 and 7. None of these three key pieces of behaviour are stated in the original requirements in Table 1. So what we see from this is that the process of requirements integration is a key constructive force in genetic design. At the same time we still have direct traceability, in this case through the sentence tags, S1, S2, etc, to the original stated functional requirements. We use “+” for implied behaviour, and “-“ for missing behaviour in the requirement tags.

Below (on the last two pages of the paper in figures 6 and 7) we show the Design Behavior Tree (DBT) that results from integrating the seven RBTs that were produced by requirements translation, integration and augmentation where needed to enable integration. It is easy to see because of the tags, S1,S2, etc, where each functional requirement occurs in the integrated DBT. “@@” mark integration points. The processes of translation, integration and inspection of the DBT revealed the following missing requirements (see Table 2 below) and where they occurred. That is, the method constructively guides the resolution of the missing requirements.

Table 2. Missing Behavior found by integration

- | |
|--|
| <ol style="list-style-type: none"> 1. The need to "close-out" the offer process for an order before assigning it. 2. The need for a shuttle to store an "assigned" order if it is currently delivering an order. 3. The need to first collect an order before delivering it. 4. The need to check for and process any stored orders after completing an order. |
|--|

To give some indication of the constructive “pull” of genetic design only approximately a third of the behaviour in the DBT came directly from translation of the original requirements. The other two-thirds of the behaviour was either missing “-“ or only implied “+” in the original set of statements we have used to guide the design.

We have been able to systematically transition from a loose natural language, high-level statement of requirements to a complete and consistent integrated formal set of requirements that preserve the intent of the original requirements. This integrated design provides a framework for making a number of other refinements to the detailed or “intelligent” behaviour that shuttles in the system might exhibit. For example, in estimating a bid price

a shuttle would need to, if it were currently delivering an order, factor in the travel cost from its current delivery destination to the pick up point for the new order, etc.

Once the missing behaviors and other problems with the DBT have been rectified it is then possible to transition to the *solution domain*. A design behavior-tree is the *problem domain* view of the “shell of a design” that shows all the states and all the flows of control (and data), modelled as interactions without any of the functionality needed to realize the various states that individual components may assume. *It has the genetic property of embodying within its form two key emergent properties of a design: (1) the component-architecture of a system and, (2) the behaviors of each of the components in the system.*

3.3 Architecture Transformation

The component architecture transformation for the Shuttle DBT (see last two pages) is not very interesting because it only involves two components. We will therefore use part of Microwave Oven DBT [3] to better illustrate the mechanics of the transformation. In the partial DBT (figure 3), a given component may appear in different parts of the tree in different states (e.g., OVEN component may appear in the Idle-state in one part of the tree and in the Cooking-state in another part of the tree). To implement the architecture transformation we need to convert a design behavior-tree to a component-based design in which each distinct component is represented only once.

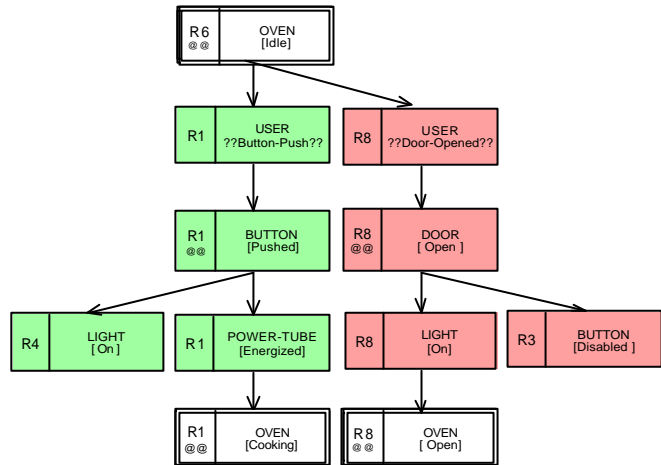


Figure 3. Partial DBT for Microwave Oven

This amounts to shifting from a representation where functional requirements are integrated to a representation, which is part of the *solution domain*, where the components mentioned in the functional requirements are themselves integrated. A simple algorithmic process may be employed to accomplish this transformation from a tree into a network [3]. *Informally, the process starts at the root of the design behavior tree (here OVEN[Idle]) and moves*

systematically down the tree towards the leaf nodes including each component (e.g. USER is included next after OVEN) and each component interaction (e.g. arrow) that is not already present. When this is done systematically the tree is transformed into a component-based design (in general a network) in which each distinct component is represented only once. When this algorithm is applied to the DBT above we get the Component Interaction Network (CIN) representation (figure 4), which is simply a component dependency network for all the components in the requirements. Notice in the DBT that BUTTON receives inputs from USER and DOOR and produces outputs to POWER-TUBE and LIGHT. These relationships (arrows) are retained in CIN below.

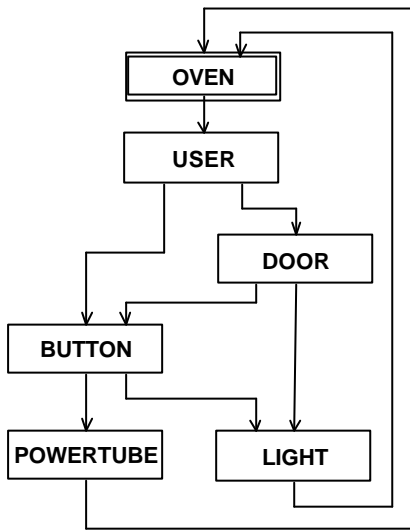


Figure 4. CIN derived from Oven DBT in Fig. 3.

This CIN represents a “first-cut” at the architecture because we can often simplify the component interfaces. Take the case of the LIGHT component. It has two inputs in the CIN. These two inputs can be reduced to a single input, because the light only has two internal states ON and Off which can be controlled by a single Boolean input.

3.4 Component Behavior Projection

In the design behavior tree, the behavior of individual components tends to be dispersed throughout the tree (for example, see the OVEN component-states in the Microwave Oven System DBT). To implement components that can be embedded in, and operate within, the derived component interaction network, it is necessary to “concentrate” each component’s behavior. We can achieve this by systematically *projecting* each component’s behavior tree (CBT) from the design behavior tree. We do this by essentially ignoring the component-states of all components other than the one we are currently projecting. In addition we must preserve

branching information at the time of projection. The resulting connected “skeleton” behavior tree for a particular component defines the behavior of the component that we will need to implement and encapsulate in the final component-based implementation. To illustrate the effect and significance of component behavior projection we show the projection of the OVEN SYSTEM component from the DBT for the Microwave Oven in figure 5.

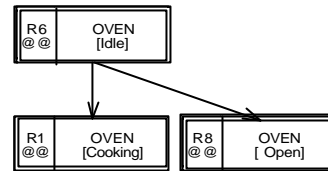


Figure 5. Oven Component Projected from Fig. 3

Component behavior projection is a key design step in the solution domain that needs to be done for each component in the design behavior tree. As part of the process it is necessary to check the leaf nodes of the projected leaf nodes to see that they properly revert “^”. We will not pursue this defect detection step here due to space limitations. However, it is a step that must always be done to ensure each component’s behavior is complete.

4. Comparison with UML and Other Methods

As Jackson wisely observed, new notations and new design methods are generally not enthusiastically received [5]. Such proposals are seen as just muddying the waters and tinkering around the edges. Our justification for ignoring this advice is that the *Behavior Tree Notation* and the accompanying *genetic design* method solve a fundamental problem – they provide a clear, simple, constructive and systematic path for going from a set of functional requirements to a design that will satisfy those requirements. Some of the major differences and advantages of the present approach are summarised below.

- The most significant advantage of genetic design over UML [1] and other methods is that it allows designers to focus on the complexity/detail of individual requirements while not having to worry about the detail in other requirements. That requirements can be dealt with one at a time (both for translation and integration) significantly reduces the complexity of creating a design. This very significantly reduces the short-term memory overload problem that has plagued software development for so long. In fact this approach to design actually *amplifies* our ability to deal with complexity. UML and other methods do not do this.
- Another important advantage of genetic design over UML is that the component architecture and the component behaviour designs of all individual components in a system are both *emergent properties* of the design behavior tree (DBT) that is constructed

by integrating all the functional requirements of the system.

- We have shown with the case study, that integration of functional requirements is a powerful way to find *behaviour gaps* and other incompleteness and inconsistency defects with a set of functional requirements. Use-cases and scenario representations that involve abstraction and loose partial views of requirements information do not have the same focus on defects and therefore are unlikely to consistently deliver the same level of constructive defect detection.
- The focus on direct translation of individual functional requirements maximizes the chances of preserving and clarifying intent and guarantees traceability to original statements of requirements. Because the focus is on translation the method approaches repeatability in design construction. The method also provides a single integrated view of the requirements which we claim makes it easier to see and find defects either manually or using automated tools.
- We have not emphasised it here but the genetic design method provides a formal, automatable method for mapping changes of requirements to changes in the architecture, the component interfaces, and the behaviors of the individual components affected by the change [8]. This follows because the architecture and individual component designs are emergent properties of the DBT that is modified by the change in functional requirements of the system.
- The main steps to get to a design are very clear: translation of requirements to behavior trees, integration of behavior trees, architecture transformation, component behaviour projection for all components. In contrast with UML there is a choice of notations to use and an accompanying set of process choices. Where to start and how to proceed is less obvious.

A detailed comparison of genetic design with the Shlaer-Mellor method [6] has previously been given in [3]. Comparisons with State-charts [4] and Cleanroom Software Engineering are available from the author.

Behavior trees now have tool support for automatic translation from the graphic design representation, via XML to CSP. The CSP can be then fed into the FDR model checker to check the system for deadlocks and livelocks [9]. The tool also does a number of consistency checks on a DBT. CSP provides a formal specification of the behavioural aspects of the semantics of Behavior Trees. We have also used extended weakest preconditions to formally characterise the behaviour semantics of the language.

5. Conclusion

Control of complexity is key in detecting and removing defects from large sets of functional requirements for systems. Genetic design facilitates the control of complexity because it allows us to consider, translate, and integrate only one requirement at a time. Application of the method to the Shuttle Case Study has demonstrated the constructive power of requirements integration as a means for the early detection and resolution of significant problems with an original high-level statement of requirements. While we have not emphasized it here, the single integrated graphical behavioral view (the DBT), is also well-suited for conducting manual formal inspections for defects. Equally, because the notation has a formal semantics, a DBT can be transformed into an input for a model-checker (e.g., FDR). This allows systematic checking for deadlocks, live-locks and a range of consistency problems that are easily detected in a DBT.

This whole approach has been successfully applied to a diverse range of real (often large) industrial applications. In all cases the method has proved very effective at defect detection and in the control of complexity (in larger systems there can be layers of behavior – the method easily accommodates this). We expect the utility of the method will increase as we enhance the tool to do sophisticated graphics, vocabulary control, and consistency checking.

6. References

- [1] Booch, G., Rumbaugh, J., Jacobson, I., 1999. *The Unified Modelling Language User Guide*, Addison-Wesley, Reading, Mass.
- [2] Davis, A., 1988, A Comparison of Techniques for the Specification of External System Behavior, *Comm. ACM*, vol. 31, No. 9, pp. 1098-1115.
- [3] Dromey, R.G., 2003. From Requirements to Design: Formalizing the Key Steps, SEFM'03, IEEE International Conference on Software Engineering and Formal Methods, (Invited Keynote Address), Brisbane, September, 2003.
- [4] Harel, D., 1987, Statecharts: Visual Formalism for Complex Systems, *Sci. Comp. Prog.*, vol. 8, pp. 231-274.
- [5] Jackson, D., Alloy: A Lightweight Object Modelling Notation, MIT Lab. for Comp. Sci. Report (1999)
- [6] Shlaer, S., Mellor, S.J., 1992. *Object Lifecycles*, Yourdon Press, New Jersey.
- [7] Woolfson, A., 2000. *Life Without Genes*, Flamingo, London.
- [8] Wen, L., Dromey, R.G., 2004. From Requirements Change to Design Change: A Formal Path, SEFM'04, IEEE International Conference on Software Engineering and Formal Methods, Beijing, September, 2004 (submitted)
- [9] Winter, K., 2004, Formalising Behavior Trees with CSP, International Conference on Integrated Formal Methods, IFM'04, LNCS vol. 2999, 148 – 167.

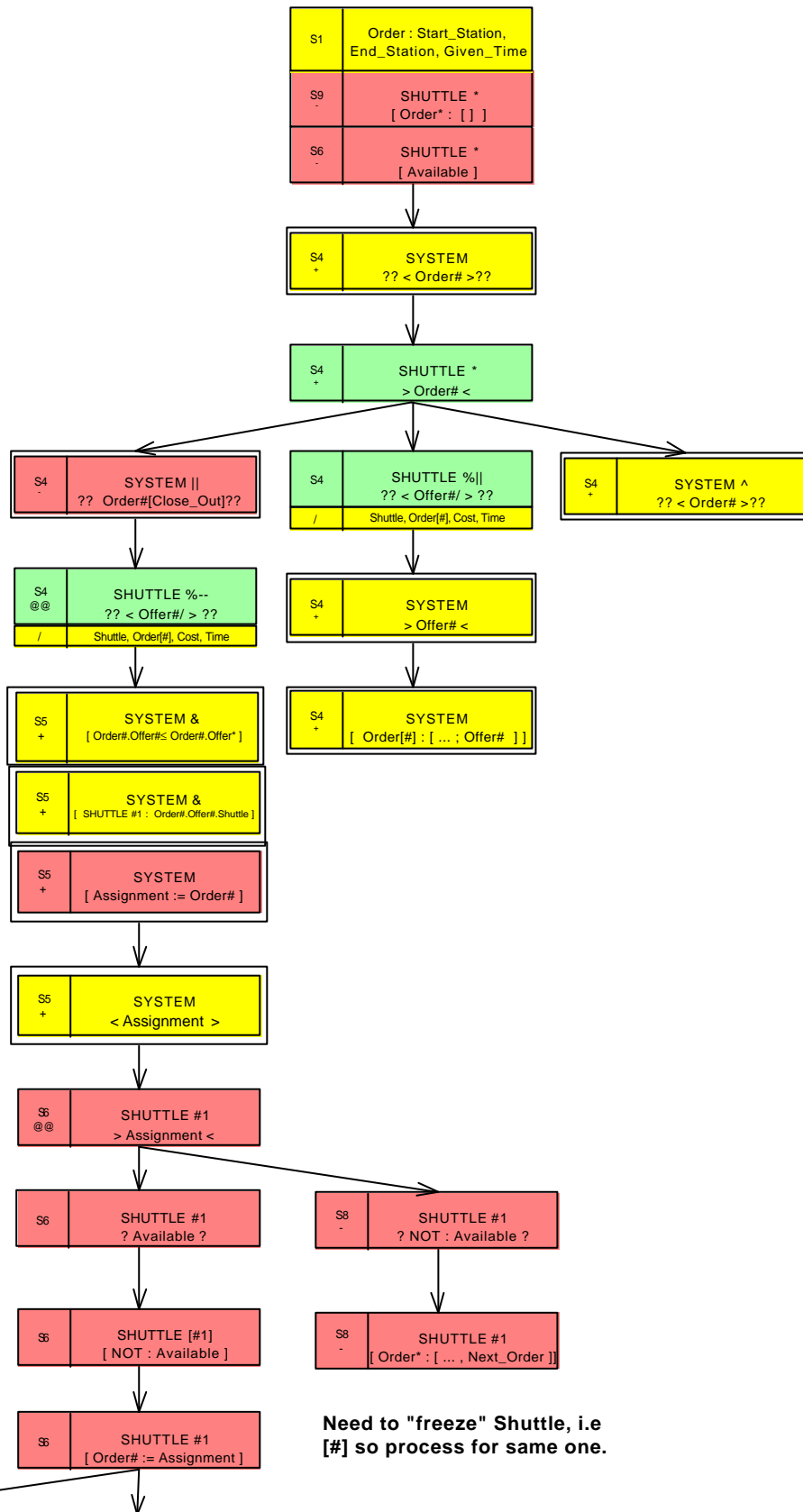


Figure 6. Top half of the integrated and augmented DBT for the Shuttle System

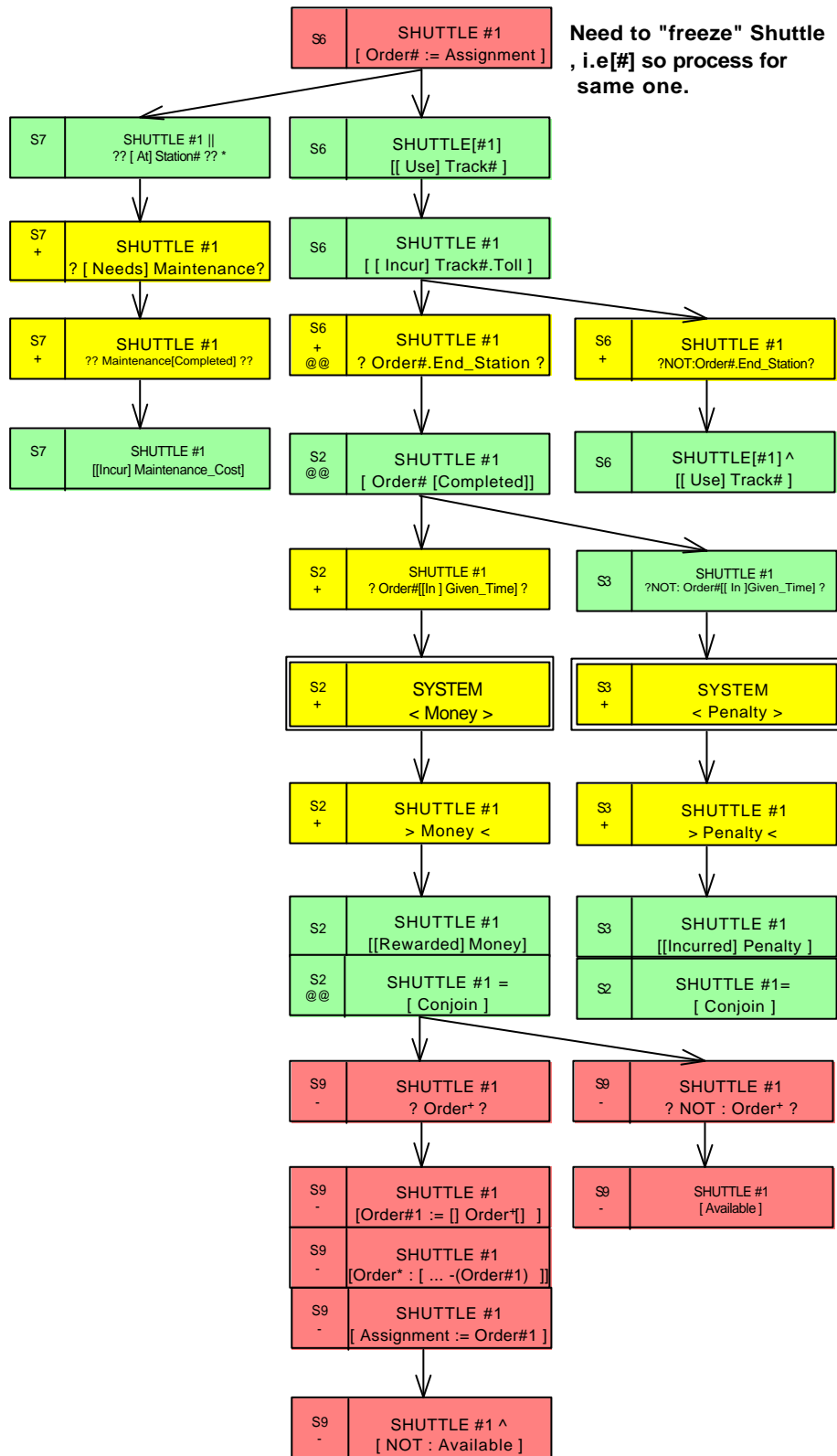


Figure 7. Bottom half of integrated and augmented DBT for Shuttle System