

# Genetic Design: Amplifying Our Ability to Deal With Requirements Complexity

R.Geoff. Dromey

Software Quality Institute, Griffith University,  
Nathan, Brisbane, Qld., 4111, AUSTRALIA  
[g.dromey@griffith.edu.au](mailto:g.dromey@griffith.edu.au)

**Abstract.** Individual functional requirements represent fragments of behavior, while a design that satisfies a set of functional requirements represents integrated behavior. 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 translation from individual functional requirements stated in natural language, may be composed, one at a time, to create an integrated design behavior tree (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 behavior designs of the individual components that make up the system – both are *emergent properties* of a DBT. The Early Warning System case study is used to illustrate this genetic design method, and show its potential for defect detection and control of complexity compared with the Statechart design method.

## 1 Introduction

The Early Warning 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, which include Statecharts, usually make it difficult to see many types of defects, particularly those that involve interactions between requirements [1][6][8]. A more practical way forward, we suggest, is to use a single integrated view. The challenges we must confront in his endeavour are:

- ~ how to get on top of requirements complexity,
- ~ how to preserve, and where necessary, clarify the intention of stakeholders, and
- ~ how to systematically, repeatably, 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 to existing software development methods. Tradition-

ally 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:

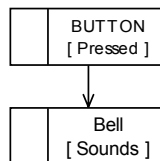
- ~ we have a representation that will formally represent the behavior in individual requirements
- ~ we have a way of combining/integrating individual requirements to create a system that will satisfy all requirements.

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. This notation is 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 behavior 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 the behavior tree below:



## 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 requirements behavior trees (RBTs), one-at-a-time, into an evolving 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 [12]. 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 behavior expressed in functional requirements does not “just happen”. There is always a precondition that must be satisfied in order for the behavior 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 behavior 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 behavior tree that forms the root of the integrated tree is excused from this requirement.*

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 originally stated functional requirements for the Early Warning System (see each sentence in Table 1). We will also provide brief commentary on the main steps. Elsewhere each of the steps in the process is described in more detail [3]

**Table 1.** Early Warning System Functional Requirements

<p><b>The EWS receives a signal from an external sensor. When the sensor is connected, the EWS processes the signal and checks if the resulting value is within a specified range. If the value of the processed signal is out of range, the system issues a warning message on the operator display and posts an alarm. If the operator does not respond to this warning within a given time interval, the system prints a fault message on a printing facility and stops monitoring the signal. The range limits are set by the operator. The system becomes ready to start monitoring the signal only after the range limits are set. The limits can be redefined after an out-of-range situation has been detected or after the operator has deliberately stopped the monitoring.</b></p> <p><b>D.Harel, M.Politi, "Modeling Reactive Systems with Statecharts", McGraw-Hill, N.Y (1998).</b></p>
---

### 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. Nouns in the text that have associated behavior are identified as components.

When requirements translation has been completed each individual functional requirement is translated to one or more corresponding RBTs. In Figure 1 we show the “raw” translations for sentences S1, S5 and S6 from the original statement of requirements in Table 1. As originally stated each of the three requirements is missing implied precondition information that would allow their direct integration. Implied preconditions (colour-coded yellow, and marked with a “+” where colour is not

available) have been added to allow the requirements to be directly integrated by finding where the root of one RBT occurs in another RBT. With S1 we have dropped the adjective “external” and acknowledged that something must be “sent” <... > in order to be “received” > ... <. From the context the behavior in S1 can only happen if the EWS is monitoring and the sensor is connected. In S5 the screen input “>> ... <<” must be sent somewhere (to the EWS). This all happens when the EWS is not monitoring. In S6 “system” is an alias for “EWS”.

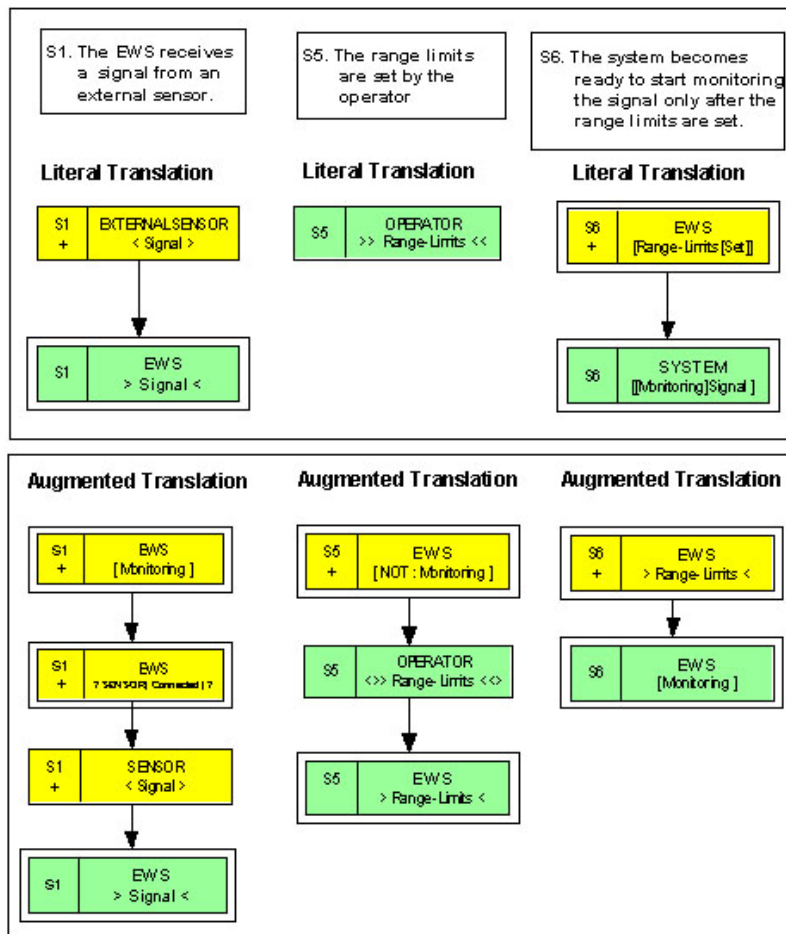


Fig. 1. RBTs for requirements S1, S5 and S6

### 3.2 Requirements Integration

Once we have carried out all the requirements translations we can systematically and incrementally construct a design behavior tree that will satisfy all its requirements by *integrating the individual requirements' behavior trees* 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 behavior 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/behavior missing from the set which prevents integration of a requirement. Attempts at integration uncover such problems with requirements at the earliest possible time. At the same time, such problems preclude automating requirements integration. Consider the case of integrating S1, S5, S6 and S7. It is not possible to integrate the initial literal translations for this set of requirements because they all have missing precondition information and a number of other problems. Each needs to be augmented as we have done above to make direct integration possible. The result of integrating the augmented requirements is shown in Figure 2.

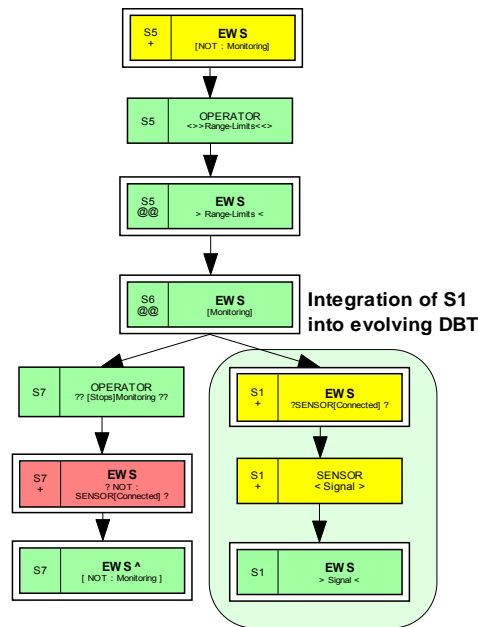


Fig. 2. Integration of the augmented RBT for S1 into the evolving DBT

Take the case of integrating S1 “the EWS receives a signal from an external sensor” with the other requirements. The literal translation of this requirement gives no direct clue how to integrate it. Examining the requirement the first thing we notice is that it implies that the sensor must “send” the signal in order for it to be subsequently “received”. For this to happen the sensor needs to be connected. Including this behavior still does not give us an RBT that can be directly integrated. To push the analysis further we must ask the question “what state must the EWS be in to receive a signal – the answer is, it must be in a “monitoring” state. When this precondition is added the augmented RBT for requirement S1 can be integrated with requirement S6 as shown in Figure 2. Similarly S7 integrates with S6. So what we see from the case of integrating S1 is that the process reveals problems with individual requirements that prevent direct integration. It also constructively provides clues about what preconditions are needed. Clearly, in this case the EWS is only in a position to receive a signal if it is monitoring and the sensor is connected. In some cases knowledge from a domain expert is necessary to resolve an integration problem. In other cases, temporal or causal information, and/or the states other components need to be in for integration to take place guide the decision.

In Figures 5 and 6 (following the paper) we show the DBT that results from integrating the RBTs that were produced by requirements translation of the sentences in Table 1, and then either direct integration or augmentation where needed to enable integration followed by 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. As well as finding integration problems, the translation and integration steps help us find and confront ambiguities and the use of aliases in the original statement of requirements.

### **3.3 Inspection of the Integrated Design Behavior Tree**

The design behavior tree turns out to be a very effective representation for revealing a range of incompleteness and inconsistency defects that are common in original statements of requirements. The Early Warning System case study has its share of incompleteness defects.

With the DBT there is the opportunity to do a manual visual formal inspection. Behavior Trees have been given a formal semantics [11] which has enabled us to build and use tools to do automated formal analyses as well. In combination, these tools provide a powerful armament for defect finding. With simple examples like the EWS it is very easy to do just a visual inspection and identify a number of defects. For larger systems, with large numbers of states and complex control structures the automated tools are essential for systematic, logically based, repeatable defect finding.

The tool [9] we have built allows us to graphically enter behavior trees and store them using XML. From the XML we generate a CSP (Communicating Sequential Processes) representation. There are several translation strategies that we can use to map behavior trees into CSP. Details of one strategy for translating behavior trees into CSP are given in [11]. One simple strategy involves defining sub-processes in

which state transitions for a component are treated as events. The CSP generated by the tool is then fed directly into the FDR model-checker. This allows us to check the DBT for deadlocks, live-locks and also to formulate and check some safety requirements [11]. We are currently extending the tool to do a number of consistency checks on a DBT. One important check that we are able to do is a reversion “^” check where control reverts back to an earlier established state. What this check allows us to do is see whether all components are in the *same* state at the reversion point as the original state realization point (e.g., with the EWS we can check EWS[Monitoring] and EWS^[Monitoring] for consistency of all the component-states that define these two system state realizations). Such a consistency check reveals that the alarm is in the “Posted” state when reversion “^” takes place, whereas it is in the “Off” state when the EWS has just realized the “Monitoring” state. This identifies an inconsistency which we have corrected in the DBT by including ALARM[Off] in requirement S3.

As we mentioned earlier for systems like the EWS it is relatively easy to do a visual inspection to identify incompleteness defects. Table 2 lists four incompleteness defects identified by inspection of the DBT for missing alternative cases.

**Table 2.** Missing Behavior found by Inspection of the DBT

1. The requirements do not say what to do if the sensor is not connected - presume must wait for connection event then start monitoring.
2. Does not say what to do if signal is "in range" - presume just goes back to monitoring.
3. Does not say what response operator needs to make if signal is not in range - presumed input new range limits.
4. Does not say what to do after printing fault message - here have presumed it goes back to a "not monitoring" state.

What is interesting when a comparison is made with the Statechart design (see figs. B2, B3, and B4, ref [6]) for this system is that none of these issues are seen as “defects” and yet the original requirements are silent in each of the cases. Perhaps some of these “defects” can be resolved as “commonsense”. However when all “gaps” are filled in this way it raises the chances of factoring in new requirements and new behavior that was not intended in the original requirements. What genetic design allows us to do is separate out what was actually stated and intended from other behavior that needs to be added in to make the behavior of a system complete. This latter behavior should be clearly delineated until it has been authorised by the stakeholders.

There is also another significant difference between the Statechart design and the design that results from requirements translation and integration. What we find with the Statechart design is that things get “added in” to the design that do not appear in the original statement of requirements. For example, in the original statement of requirements for the EWS there is no mention of the power going off and on and yet it

appears in the design (see figs. B2 and B4, ref. [6]) without any comment. We have no issue with need for the power to be off and on. However when these sort of design/requirements decisions are done “on the fly” then traceability to original requirements is lost. Also the chances of introducing something that was not intended are greatly increased. In applying genetic design to industry applications and comparing the design documents produced using UML and other representations, time and time again we have observed discrepancies between originally stated requirements and what ends up in the design. Things get left out and things get added in with no acknowledgement of what has happened or why it has happened. Genetic design with behavior trees provides a practical way of controlling and avoiding such problems.

The processes of translation, integration, and inspection of the DBT have revealed a number of defects and where they occurred. The incompleteness problems are identified in Table 2 and in Figures 5 and 6. The method has constructively guided us in the resolution of the missing requirements. To give some indication of the constructive “pull” of genetic design, only approximately two-thirds of the behavior in the DBT came from the original requirements. The other third of the behavior was either missing “-“ or 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.

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.*

### 3.4 Architecture Transformation

The component architecture, which is an emergent property of the DBT is described elsewhere [5]. We will use the Early Warning System DBT given in Figures 5 and 6 to illustrate how this transformation is done and how the architecture is derived. In the DBT, a given component may appear in different parts of the tree in different states (e.g., the EWS component may appear in the Monitoring-state in one part of the tree and in the NOT : Monitoring-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.

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 [3,5]. *Informally, the process starts at the root of the design behavior tree (here EWS[NOT : Monitoring]) and moves systematically down the tree towards*

the leaf nodes including each component (e.g. OPERATOR is included next after EWS) 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 EWS DBT we get the Component Interaction Network (CIN) representation (Figure 3), which is simply a component dependency network for all the components in the requirements. Notice in the DBT that DISPLAY receives input from the EWS and produces outputs to OPERATOR and the EWS. These relationships (arrows) are retained in CIN below. The “double-headed arrow in the EWS – OPERATOR case indicates that control flows first to the OPERATOR (singled-headed arrow) then subsequently from the OPERATOR back to the EWS (double-headed arrow). Clearly the CIN shows that the EWS (in this case acting as the system component in the design) is the principal controlling and integrating agent for the behavior of the system.

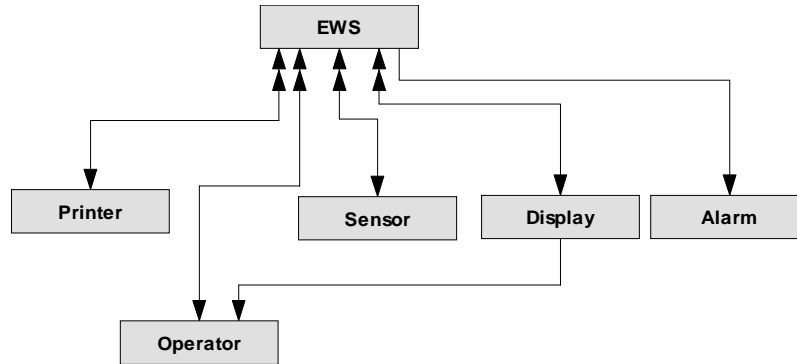


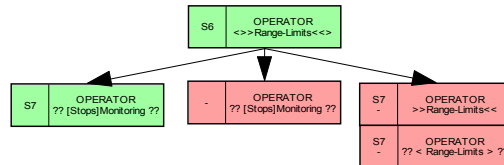
Fig. 3. CIN derived from EWS DBT in Figures 5 and 6

This CIN represents a “first-cut” at the architecture. We can often simplify the component interfaces. Space does not permit this process to be discussed here (see [3,5] for more details).

### 3.5 Component Behavior Projection

In the design behavior tree, the behavior of individual components tends to be dispersed throughout the tree (for example, see the OPERATOR component-states in the EWS 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 com-

ponent 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 OPERATOR component from the DBT for the EWS in Figures 5 and 6.



**Fig. 4.** Operator Component Behavior Projected from Figures 5 and 6 and augmented.

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 projected leaf nodes to see that they properly revert “^”. Here we need to add reversions “^” that requires that the operator will need to input new range limits after stopping monitoring. From this we see that projection helps to clearly identify and remove another class of defects from components.

#### 4 Comparison with Statecharts and Other Methods

As Jackson wisely observed, new notations and new design methods are generally not enthusiastically received [7]. Such proposals are seen as just muddying the waters and tinkering around the edges. What we have tried to show in this treatment and the accompanying case study is that there are some significant differences and potential advantages of Behavior Trees/genetic design over the leading, and most mature state-based design method - Statecharts[6]. Time, more widespread use, and independent validation of the method is needed to confirm these advantages.

We summarize some of the major differences and advantages we claim for genetic design:

- The most significant advantage of genetic design over Statecharts, UML and other methods is that it allows designers to focus on the complexity/detail of each individual requirement one at a time, 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, in turn, 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.
- Another important advantage of genetic design over Statecharts and other methods is that the component architecture and the component behavior designs of all individual components in a system are both *emergent properties* of the design behavior

tree that is constructed by integrating all the functional requirements of the system.

- We have also shown using the case study, and elsewhere [4] that integration of functional requirements is a powerful constructive force for finding *behavior gaps* and other incompleteness and inconsistency defects with a set of functional requirements. Because the use of Statecharts does not have the same focus on defect detection it is unlikely to consistently deliver a comparable detection rate.
- Statecharts and other development methods, also run a much greater risk of not preserving original intention because they do not employ a rigorous translation process to transition from an informal statement of requirements to a formal representation. Evidence of this is seen in figs. B2, B3 and B4 of ref. [6]. We find a number of new terms (e.g., “setting up”, “idle”, “halt”, and so on) appear in the design figures that are not in the original statement of requirements (see Table 1) while others like alarm “posted” disappear. This change in terminology contrasts with the focus in genetic design on direct translation of individual functional requirements which maximizes the chances of preserving and clarifying original intent and guaranteeing traceability to original statements of requirements. Because the focus is on translation the genetic design approaches repeatability in transitioning from an informal to a formal representation. Genetic design also provides a single integrated view of the requirements compared with the multiple views, (statecharts, activity charts and module charts) of the Statechart method. The integrated view, we claim, makes it easier to see and find defects either manually or using automated tools. It also makes it easier to see that original intent has been preserved in a design. For example, take the fourth sentence in Table 1. “If the operator does not respond to this warning in a given time interval, the system prints a fault message on a printing facility and stops monitoring the signal”. In the DBT (figs. 5 and 6) this requirement is directly traceable as the behavior fragment S4. In addition, the alternative case (not included in Table 1), when the operator responds is also accommodated. In contrast, with the statechart (fig. B4, ref. [6]) we claim it is much less obvious that this original intent has been completely and accurately captured.
- We have not emphasised it here but genetic design 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 [10]. 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.
- Genetic design also uses *structure trees*, *composition trees* and *user-interface behavior trees* to provide equally useful integrated views of all the data requirements, the formal structural requirements and the interaction requirements that we almost always encounter when designing large-scale systems.
- Behavior trees provide strong support for requirements elicitation and requirements analysis. They can be used equally well with broad user requirements, or a detailed SRS, or to formally model an individual scenario. The method does not however depend on requirements being nicely structured.

## 5 Conclusion

Amplification of our ability to deal with complexity is the single most important problem to overcome in order to advance the practice of software engineering. Genetic design has the potential to make an important contribution to solving this problem because it allows us to consider, translate, and integrate only one requirement at a time. Application of the method to the Early Warning System Case Study has demonstrated the constructive power of requirements integration as a means for complexity control and the early detection and resolution of significant problems with original high-level statements of requirements. The case study will allow others to benchmark genetic design against Statecharts, the leading state-based design method.

Genetic design 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 we are building to support the method.

## 6 References

1. Booch, G., Rumbaugh, J., Jacobson, I.: *The Unified Modelling Language User Guide*, Addison-Wesley, Reading, Mass. (1999)
2. Davis, A.: A Comparison of Techniques for the Specification of External System Behavior, *Comm. ACM*, vol. 31, No. 9, (1988), pp. 1098-1115
3. Dromey, R.G.: From Requirements to Design: Formalizing the Key Steps, SEFM 2003, IEEE International Conference on Software Engineering and Formal Methods, (Invited Keynote Address), Brisbane, September, (2003),pp.2-11.
4. Dromey, R.G.: Using Behavior Trees to Model the Autonomous Shuttle System, ICSE-2004, 3<sup>rd</sup> International Workshop on Scenarios and State Machines: Models, Algorithms and Tools, (SCESM'04), Edinburgh, May, (2004)
5. Dromey, R.G.: Architecture as an Emergent Property of Requirements Integration, ICSE-2003, Software Requirements to Architecture Workshop – STRAW' 03, Portland, USA, May (2003).
6. Harel, D., Politi, M., *Modeling Reactive Systems with Statecharts*, McGraw-Hill, N.Y. (1998).
7. Jackson, D.: Alloy: A Lightweight Object Modelling Notation, MIT Lab. for Comp. Sci. Report (1999)
8. Shlaer, S., Mellor, S.J.: *Object Lifecycles*, Yourdon Press, New Jersey (1992).
9. Smith, C., Winter, K., Hayes, I., Dromey, R.G., Lindsay, P., Carrington, D.: An Environment for Building a System Out of Its Requirements, Tools Track, 19<sup>th</sup> IEEE International Conference on Automated Software Engineering, Linz, Austria, Sept. (2004).
10. Wen, L., Dromey, R.G: From Requirements Change to Design Change: A Formal Path, SEFM 2004, IEEE International Conference on Software Engineering and Formal Methods, Eds., J.R. Cuellar, Z. Liu, Beijing, September, (2004), pp. 104-113.
11. Winter, K.: Formalising Behavior Trees with CSP, International Conference on Integrated Formal Methods, IFM'04, LNCS vol. 2999, (2004), pp. 148 – 167.
12. Woolfson, A: *Life Without Genes*, Flamingo, London (2000)

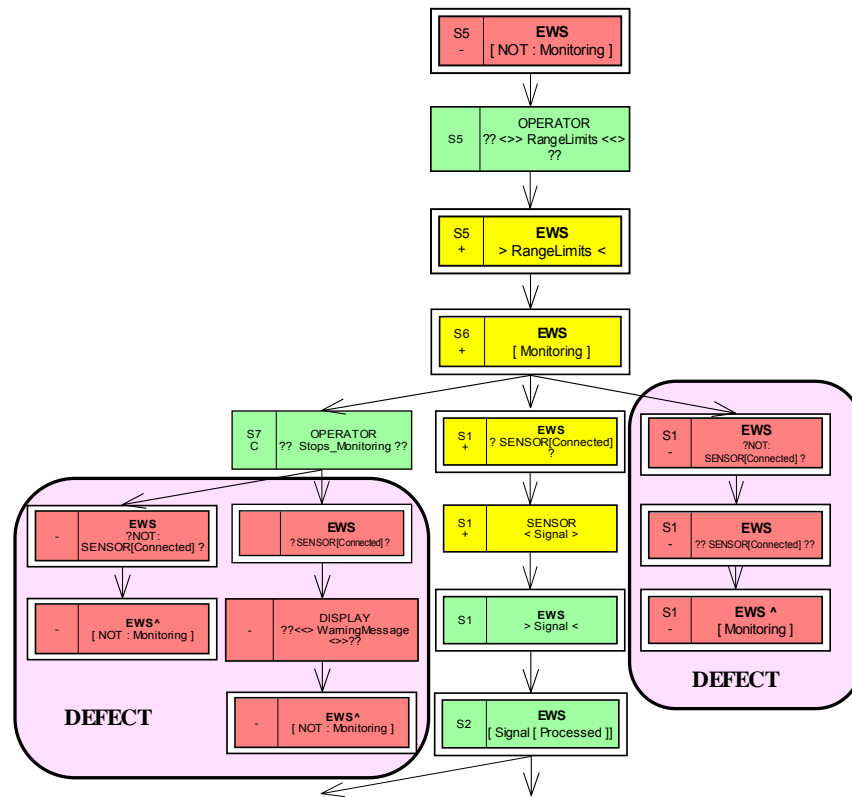


Fig. 5. Top Half of the Integrated DBT for the EWS

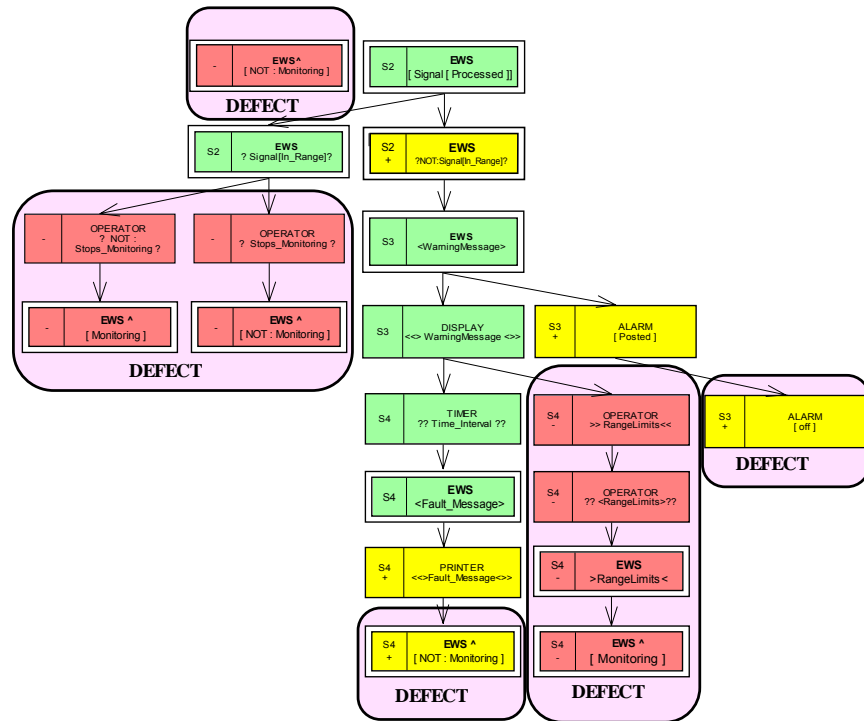


Fig. 6. Bottom half of the integrated DBT for the EWS