CHAPTER 1

FORMALIZING THE TRANSITION FROM REQUIREMENTS TO DESIGN

R.Geoff. Dromey

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

Despite the advances in software engineering since 1968, current methods for going from a set of functional requirements to a design are not as direct, formal, repeatable and constructive as we would like. Progress with this fundamental problem is possible once we recognize that 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 of individual functional requirements may be composed, one at a time, to create an integrated design behavior tree. From this problem domain representation it is then possible to transition directly, systematically, and repeatably 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 the integrated design behavior tree.

"I believe that failure is less frequently attributable to either insufficiency of means or impatience of labour than to a confused understanding of the thing actually to be done".

John Ruskin

1. Introduction

A great challenge that continues to confront software engineering is how to proceed in a systematic way from a set of functional requirements to a design that will satisfy those requirements. In practice, the task is further complicated by defects in the original requirements and, subsequent

changes to the requirements. A first step towards taking up this challenge is to ask what are functional requirements? Study of diverse sets of functional requirements suggests that it is safe to conclude individual requirements express constrained behavior. By comparison, a system that satisfies a set of functional requirements exhibits integrated constrained behavior. The latter behavior of systems is not inherently different. Therefore, we may ask, can the same formal representation of behavior be used for requirements and for a design? If it could, it may clarify the requirements-design relationship. Functional requirements contain, and systems exhibit, the behavior summarized below.

- Components realise states
- Components change states
- Components have sets of attributes/properties that are assigned and change values
- Components, by changing states, can cause other components to change their states
- Conditions/decisions, and events are associated with components and states.
- Interactions between components also play a key role in describing behavior. They involve sequential, concurrent and threaded control-flow and/or data-flow between components.

Notations like sequence diagrams, class and activity diagrams from UML[1], data-flow diagrams, Petri-nets[2], Statecharts[6], and Message Sequence Charts (MSCs)[7], accommodate some or all of the behavior we find expressed in functional requirements and designs. Individually however, none of these notations provide the level of constructive and integrated support we need in a single representation. This forces us to contemplate another representation for functional requirements and designs. As Jackson wisely remarked[8], such ventures are generally not enthusiastically received a consensus is that new proposals just muddy the waters. Our justification for ignoring this advice is that the Behavior Tree Notation solves a fundamental problem it provides a clear, simple, constructive and systematic path for going from a set of functional requirements to a design that will satisfy those requirements. In other words, it provides a systematic means to transition from the microscopic behavior of functional requirements to the integrated macroscopic behaviour of a system that satisfies those functional requirements. In addition the component architecture and individual

component behavior designs for the system are both emergent properties of the integrated macroscopic behavior of a system.

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 in terms of components realizing states, augmented by the logic and graphic forms of conventions found in programming languages to support composition, events, control-flow, data-flow, and threads. Behavior trees are equally suited to capture behavior expressed in the natural language representation of functional requirements as to provide an abstract graphical representation of behavior expressed in a program. To use David Harels metaphor, Behavior Trees represent a lifting up of behavior expressed in programming languages to a higher level of abstraction.

Definition: A Behavior Tree is a formal, tree-like graphical form that represents behavior of individual or networks of entities which realize or change states, make decisions, respond-to/cause events, and interact by exchanging information and/or passing control.

To support the implementation of software intensive systems we must capture, first in a formal specification of the requirements, then in the design, and finally in the software; the actions, events, decisions, and/or logic obligations, and constraints expressed in the original natural language requirements for a system. Behavior trees do this. They 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. Figure 1 shows a sample translation to a behavior tree. Components are in **bold** and *states*, *conditions* and *events* are in *italics*.

The principal conventions of the notation for component-states are the graphical forms for associating with a component [State],??Event??, ?Decision?, [Attribute := expression | State] or output data-flow <Data-Output> and input data-flow >Data-Input<. Exactly what can be an event, a decision, a state, etc are built on the formal foundations of expressions, Boolean expressions and quantifier-free formulae (qff). To assist with traceability to original requirements a simple convention is followed. Tags (e.g. R1 and R2, etc, see below) are used to refer to the original requirement in the document that is being translated. Record/data def-

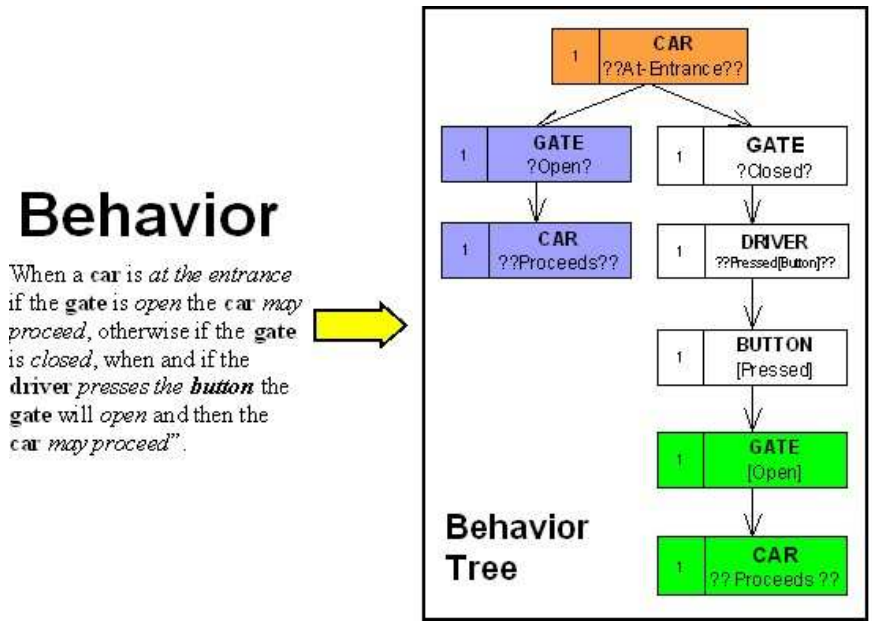

Fig. 1. Translation of natural language to a Behavior Tree

initions, other constraints, and relations are handled by state extension. System states are used to model high-level (abstract) behavior, some pre-conditions/postconditions and possibly other behavior that has not been associated with particular components. They are represented by rectangles with a double line (===) border. A brief summary of key elements of the notation is given in Figure 2, (see web-site http://www.sqi.gu.edu.au/gse/ for details).

In practice, when translating functional requirements into behavior trees we often find that there is a lot of behavior that is either missing or is only implied by a requirement. We mark implied behavior with a + in the tag (and/or the colour yellow if colour can be shown). Behavior that is missing is marked with a - in the tag (and/or the colour red). Explicit behavior in the original requirement that is translated and captured in the behavior tree has no +/- marking, and the colour green is used - see Figure 4. These conventions maximize traceability to original requirements. The Green-Yellow-Red traffic-light metaphor is intended to indicate the need for caution (yellow) and danger (red) and to draw attention to deficiencies

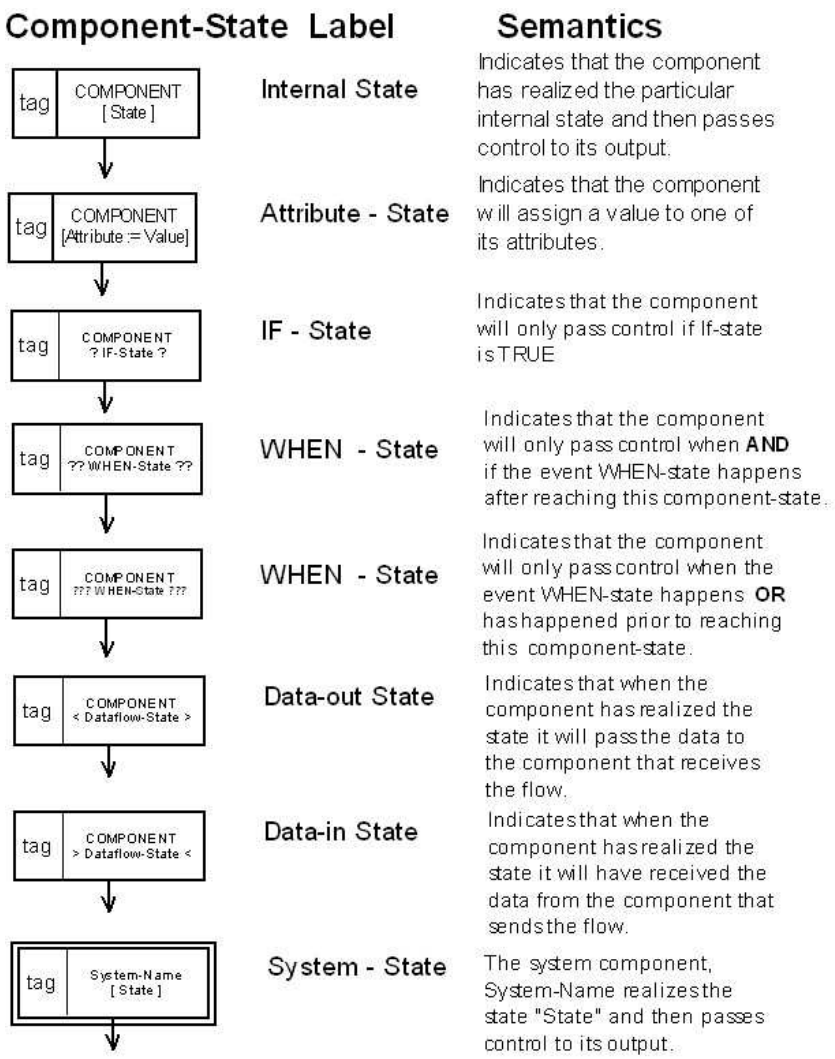

Fig. 2. Behavior Tree Notation, Key Elements

in the original requirements. Subsequent change to a working system requirements/design is marked by a ++ in the tag and/or the colour blue. These conventions are particularly useful when discussing requirements and designs with stakeholders. It provides a clear record of the evolution of, and

deficiencies in the original system. We can now explore the relationship between a set of functional requirements and their corresponding design. And from this follows a systematic method for constructing a design from its requirements.

3. Genetic Design

Conventional software engineering applies the underlying design strategy of constructing a design, D that will satisfy its set of functional requirements F. F may be represented by a set of natural language statements $\{R_1, R_2, \ldots, R_m\}$. Representing this symbolically we have:

$$D \textbf{ sat } F$$

In contrast to this, genetic design enables us to use the behavior tree notation to construct a design out of its set of functional requirements. We achieve this by first applying a translation relation T to the natural language description of each functional requirement. A translation relation T takes a natural language statement of a functional requirement as input and produces a set of requirements behavior trees as output (in most cases one natural language statement of a functional requirement translates to one requirements behavior tree). In general, the requirement R_i yields one or more (the set F_i) requirements behavior trees (RBTs). The matter is further complicated because there may be more than one equivalent behavior tree translation (the set E_i) for the original natural language requirement R_i. So in each case, we have:

$$R_i \text{ T } F_i \text{ for all } i \in [1..m] \text{ and } F_i = \left\{F_{i1}, F_{i2}, , F_{ik(i)}\right\} \text{ and } F_i \in E_i$$

A complete set of requirements behavior trees F is obtained from the union of all the sets F_i. A design behavior tree D is constructed by integrating the behavior trees for all individual functional requirements (RBTs), one-at-a-time, into an evolving design behavior tree (DBT). Applying the behavior tree integration relation I to all the RBTs yields a design behavior tree D (it may be possible to construct more than one DBT for a given set of RBTs when this happens the resulting set of DBTs yield equivalent architectures and component behavior projections because integration does not add any new direct component-state to the component-state relationship of a set of RBTs). We have:

$$F \text{ I } D \text{ where } F = \{F_1 \cup F_2 \cup \ldots \cup F_m\}$$

If we use $a \leq b$ to denote that the behavior tree a is a sub-structure of the behavior tree b then:

$$F_{i1} \leq D \wedge F_{i2} \leq D \wedge \ldots \wedge F_{ik(i)} \leq D \text{ for all } i \in [1..m] \text{ and all } k(i) \geq 1$$

This tells us that each RBT is a substructure of the design behavior tree.

This very significantly reduces the complexity of a key part of the design process and any subsequent change process. Any design, built out of its requirements will conform to the weaker criterion of satisfying its set of functional requirements.

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/integration. This property is the exact same property that a set of pieces for a jigsaw puzzle possess. And, interestingly, it is the same property possessed by a set of genes that create a living entity. Witness the remark by geneticist Adrian Woolfson: in his recent book[15](p.12), Life Without Genes, *"we may thus imagine a gene kit as a cardboard box filled with genes. On the front and sides of the box is a brightly coloured picture of the creature that might in principle be constructed if the information in the kit is used to instruct a biological manufacturing process"*

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. In practice, this precondition may be embodied in the behavior tree representation of a functional requirement (as a component-state or as a composed set of component states) or it may be missing - the latter situation represents a defect that needs rectification. The point to be made here is that this precondition is needed, in each case, in order to integrate the requirement with at least one other member of the set of functional requirements for a system. (In practice, the root node of a behavior tree *often* embodies the precondition we are seeking). We call this foundational requirement of the genetic software engineering method, 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 pivotal, 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 requirements 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 rule for composing/integrating functional requirements expressed as behavior trees is more formally expressed by the following axiom.

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 functional requirement that forms the root of the design behavior tree is excluded from this requirement. The external environment makes its precondition applicable*).

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 the first stage of the design process, in the problem domain, 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
- Component behavior projection

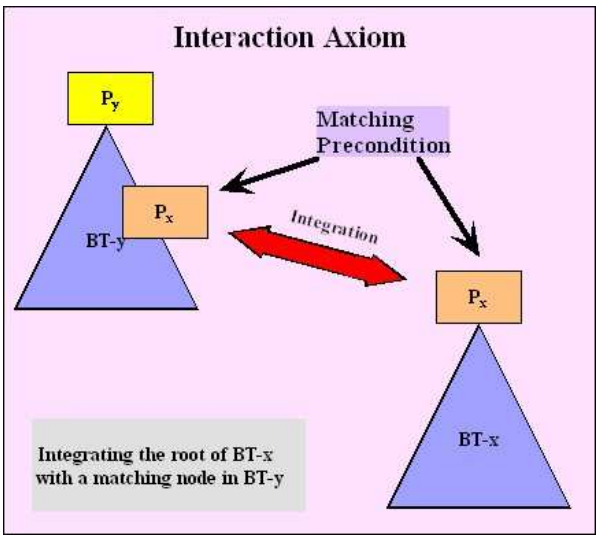

Fig. 3. Interaction Axiom-Graphic Form

Each overarching step needs to be augmented with a verification/defect detection and refinement step designed specifically to isolate and correct the class of defects that show up in the different work products generated by the process.

Our intent now is to introduce the main ideas of genetic design. Study of a simple example has proven to be a good way to provide an initial understanding of the overall process. For our purposes, and for the purposes of comparison, we will use a design example for a Microwave Oven that has already been published in the literature[11]. The seven stated functional requirements for the Microwave Oven problem[11](p.36) are given in Table 1. Shlaer, and Mellor have applied their state-based Object-Oriented Analysis method to this set of functional requirements.

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.

Table 1. Functional Requirements for Microwave Oven

R1.	There is a single control button available for the use of the oven. If the oven is idle with the door closed and you push the button, the oven will start cooking (that is, energize the power-tube for one minute).
R2.	If the button is pushed while the oven is cooking, it will cause the oven to cook for an extra minute.
R3.	Pushing the button when the door is open has no effect (because it is disabled).
R4.	Whenever the oven is cooking or the door is open, the light in the oven will be on.
R5.	Opening the door stops the cooking.
R6.	Closing the door turns off the light. This is the normal idle state, prior to cooking when the user has placed food in the oven.
R7.	If the oven times out, the light and the power-tube are turned off and then a beeper emits a sound to indicate that the cooking has finished.

3.1.1. *Translation*

Translation identifies the *components* (including actors and users), the *states* they realise (and *attribute/property* 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.

Example Translation

Translation of R7 from Table 1 will now be considered in slightly more detail. For this requirement we have put the *states/actions* in italics and made the **components** bold, that is If the **oven** *times out* the **light** and the **power-tube** are *turned off* and a **beeper** *emits a sound* to indicate that *cooking has finished*. Figure 4 gives a translation of this requirement R7, to a corresponding requirements behavior tree (RBT). In this translation we have followed the convention of trying wherever possible to associate higher level system states (here OVEN states) with each functional requirement, to act as preconditions/postconditions.

What we see from this translation process is that even for a very simple example, it can identify problems that, on the surface, may not otherwise be apparent (e.g. the original requirement, as stated, leaves out the precondition that the oven needs to be cooking in order to subsequently time-out). In the behavior tree representation tags (here R7) provide direct traceability back to the original statement of requirements. Our claim is that the translation process is highly repeatable if translators forego the temptation to interpret, design, introduce new things, and leave things out, as they do

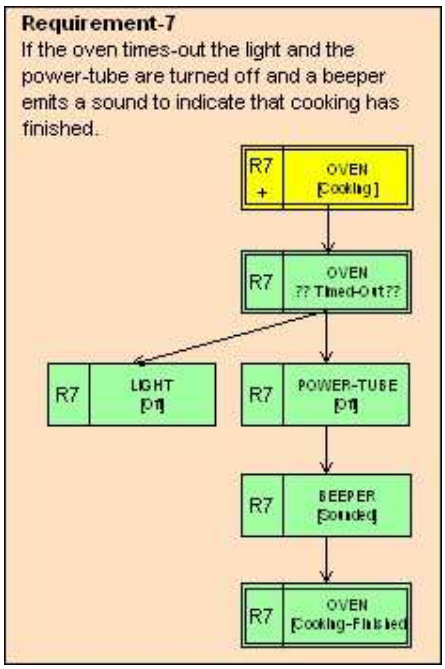

Fig. 4. Behavior Tree produced by translation of requirement R7 in Table 1

an initial translation. In other words translation needs to be done meticulously, sentence-by-sentence and word-by-word. In doing translations there is no guarantee that two people will get exactly the same result because there may be more than one way of representing the same behavior. The best we can hope for is that they would get an equivalent result. The translations of the other six functional requirements for the Microwave from Table 1 are shown in figure 5, together with implied behavior that includes oven system states we have chosen to add. While these additions are "new" they are clearly distinguished from the original behavior. Later, the relevance and importance of including system-states will be made clear. For now it suffices to say that they provide a representation (high-level description) of the behavior of a system independent of the behavior of any of the components in the system.

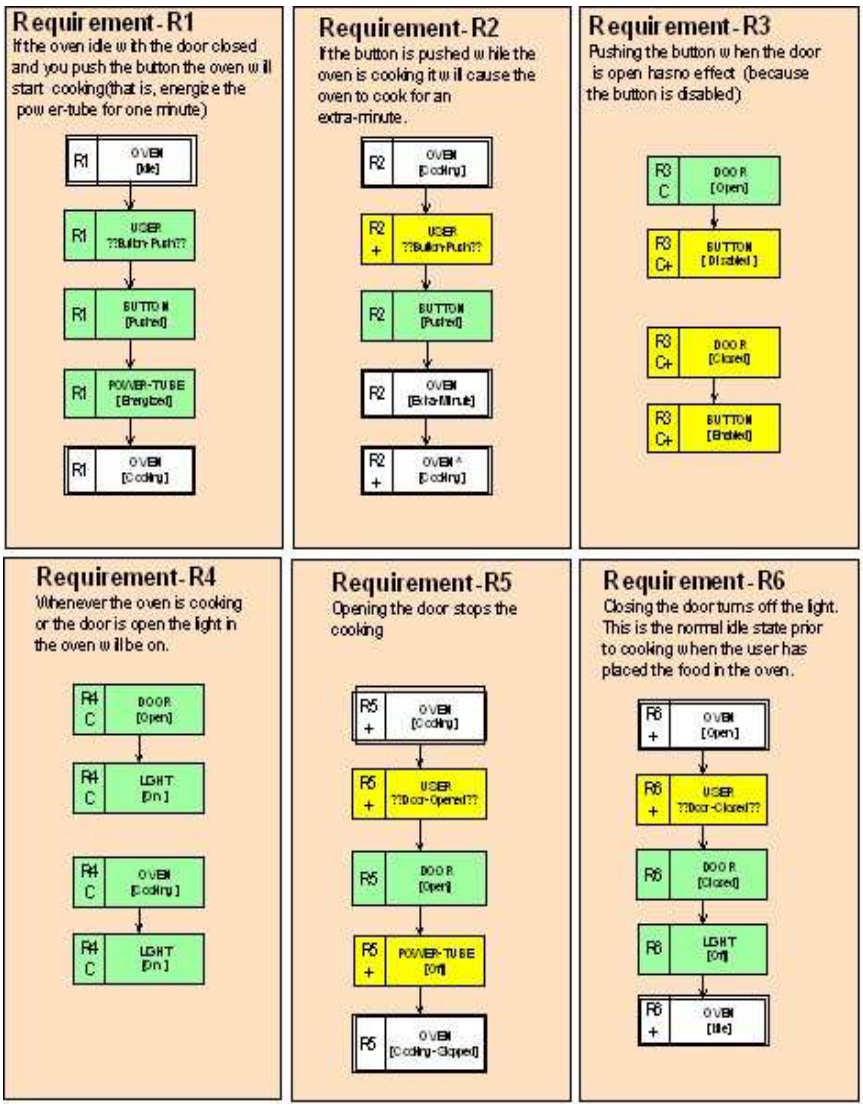

Fig. 5. Behavior Trees for Microwave Oven translated from Table 1

3.1.2. *Translation Defect Detection*

During initial translation of functional requirements to behavior trees there are four principal types of defects that we encounter:

- Aliases
- Ambiguities, where not enough context has been provided to allow us to distinguish among more than one possible interpretation of the behavior described. Unfortunately there is no guarantee that a translator will always recognize an ambiguity when doing a translation this obviously impacts our chances of achieving repeatability when doing translations.
- Incorrect causal, logical and temporal attributions. For example, in R4 of our Microwave Oven example, it is implied that the oven realizing the state "cooking" causes the light to go on. Here it is really the button being pushed which causes the light to go on and the system (oven) to realize the system-state "cooking". An example of the latter case would be "the man got in the car and drove off". Here "and" should be replaced by "then", because getting in the car happens first.
- Missing implied and/or alternative behavior. For example, in R5 for the oven, the actor who opens the door is left out, together with the fact that the power-tube needs to be off for the oven to stop cooking.

It is necessary to maintain a vocabulary of component names and a vocabulary of states associated with each component to maximize our chances of detecting aliases. In Table 2 we show the states collected for the OVEN component from requirement R7. As other requirements are translated more states for the Oven component may accumulate in this table.

Table 2. States for a Component from Requirement 7

COMPONENT	**STATE**	**FR**
OVEN	Cooking	R7
	Timed-Out	R7
	Cooking{*Finished*}	R7

In practice we have a tool[12] that automatically collects and generates this information as each requirement is entered graphically into the system.

A final point should be made about translation. It does not matter how good or how formal the representations are that we use for design/modeling, unless the first step that crosses the informal-formal barrier is as rigorous, intent-preserving, and as close to repeatable as possible, all subsequent steps will be undermined because they are not built on a firm foundation. Behavior trees give us a chance to create that foundation.

3.2. *Requirements Integration*

When requirements translation has been completed each individual functional requirement is translated to one or more corresponding requirements behavior tree(s) (RBT). We can then systematically and incrementally construct a design behavior tree (DBT) that will satisfy all its requirements *by integrating the requirements behavior trees* (RBT). 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 the other tree and grafting the two trees together at that point. This process generalises when we need to integrate N behavior trees. We attempt to integrate two behavior trees at a time either two RBTs, an RBT with a partial DBT or two partial DBTs. 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 an early time when the consequences and costs are likely to be minimized.

Example Integration

To illustrate the process of requirements integration we will integrate requirement R6, with part of the constraint requirement R3C to form a partial design behavior tree (DBT) (note in general constraint requirements need to be integrated into a DBT wherever their root node appears in the DBT. This is straightforward because the root node (and precondition) of R3C, DOOR[Closed] occurs in R6. We integrate R3C into R6 at this node. Because R3C is a constraint it should be integrated into every requirement that has a door closed state (in this case there is only one such node). The result of the integration is shown in Figure 6.

When R6 and R3C have been integrated we have a "partial design" (the evolving design behavior tree) whose behavior will satisfy R6, and the R3C constraint. In this partial DBT it is clear and traceable where and how each of the original functional requirements contribute to the design. Using this same behavior tree grafting process, a complete design is constructed (it evolves) incrementally by integrating RBTs and/or DBTs pairwise until we

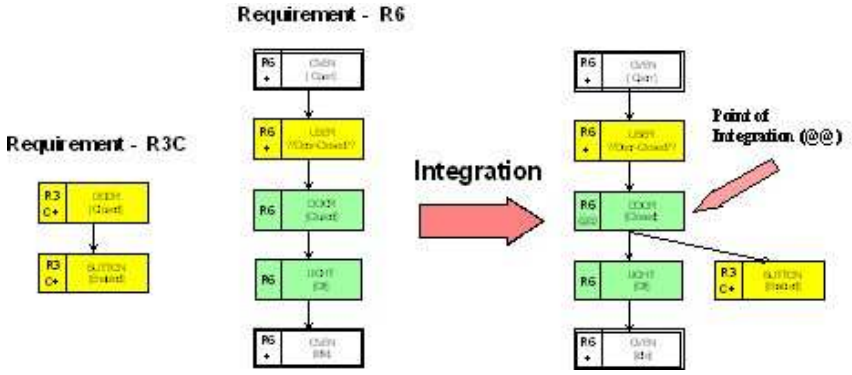

Fig. 6. Result of Integrating R6 and R3C

are left with a single final DBT (see Figure 7).

This is the ideal for design construction that is realizable when all requirements are consistent, complete, composable and do not contain redundancies. When it is not possible to integrate an RBT or partial DBT with any other it points to an integration problem with the specified requirements that needs to be resolved. Being able to construct a design incrementally significantly reduces the complexity of this critical phase of the design process. And importantly, it provides direct traceability to the original natural language statement of the functional requirements.

3.2.1. *Integration Defect Detection*

During integration of functional requirements represented as behavior trees (RBTs) there are four principal types of defects that we encounter:

- The RBT that we are trying to integrate has a missing or inappropriate (it may be too weak or too strong or domain-incorrect) precondition that prevents integration by matching the root of the RBT with a node in some other RBT or in the partially constructed DBT. For example, take the case of R5 for the Microwave Oven: it can only be integrated directly with R1 by including OVEN[Cooking] as a precondition.
- The behavior in a partial DBT or RBT, where the current RBT needs to be integrated, is missing or incorrect.
- Both of the first two types of defects may occur at the same time.

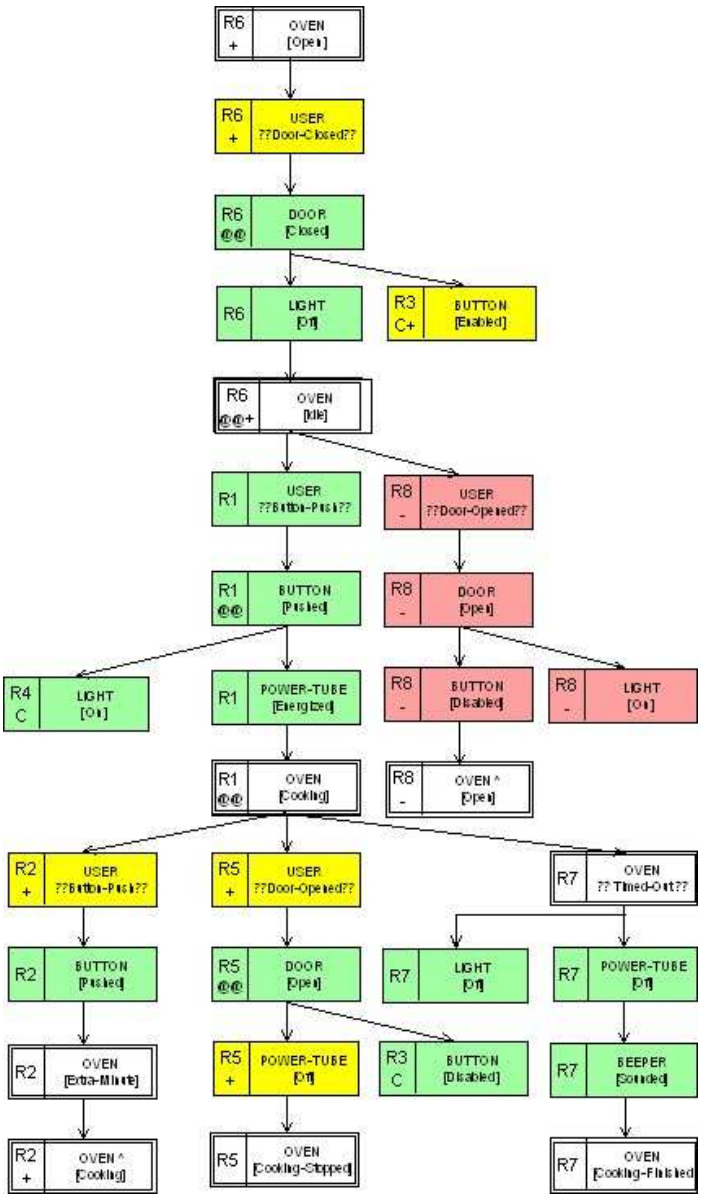

Fig. 7. Integration of all functional requirements for Microwave Oven

Resolving this type of problem may sometimes require domain knowledge. Examples of this and other integration problems can be found in reference [4].

- In some cases, when we attempt to integrate an RBT we find that more than the leaf node overlaps with the other RBT or partial DBT. In such cases this redundancy can be removed at the time of integration.

While in principal, it is possible to construct an algorithm to "automate" the integration step, because of the integration problems that we frequently encounter in real systems, it is better to have manual control over the integration process. Tool support can however be used to identify the nodes that satisfy the matching criterion for integration. Our experience with using integration in large industry systems is that the method uncovers early on problems that have been completely overlooked using conventional formal inspections. The lesson we have learned is that requirements integration is a key integrity check that it is always wise to apply to any set of requirements that are to be used as a basis for constructing a design.

3.2.2. *Inspection and Automated Defect Detection*

Once we have a set of functional requirements represented as an integrated design behavior tree we are in a strong position to carry out a range of defect detection steps. 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 Microwave Oven System case study has its share of incompleteness and other defects.

The DBT can be subject to a manual visual formal inspection and because Behavior Trees have a formal semantics[14] we can also use tools [12] to do automated formal analyses. In combination, these tools provide a powerful armament for defect finding. With simple examples like the Microwave Oven 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. We will now consider a number of systematic manual and automated defect checks that can be performed on a DBT.

Missing Conditions and Events

A common problem is with original statements of requirements that describe a set of conditions that may apply at some point in the behavior of the system. They often leave out the case that would make the behavior complete. The simplest such case is where a requirement says what should happen if some condition applies but the requirements are silent on what should happen if the condition does not apply. There can also be missing events at some point in the behavior of the system. For example, with the Microwave case study a very glaring missing event is in requirement R5. It says "opening the door stops the cooking" but neglects to mention that is possible to open the Microwave door when it is idle/closed. To systematically "discover" this event-incompleteness defect we can use the following process. We make a list of all events that can happen in the system (this includes the user opening the door). We then examine those parts of the DBT where events occur and ask the question "could any of the other events that we have listed occur at this point?" In the case where the OVEN[Idle] occurs the only event in the original requirements is that the user-event of pushing the button to start the cooking can occur (see Figure 8).

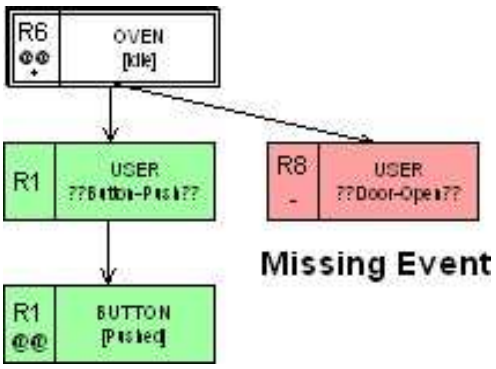

Fig. 8. Missing event detected by the event completeness check rule

In this context, when we ask what other event, out of our list of events could happen when the Oven is Idle, we discover the user could open the door. We have added this missing event in as requirement R8.

Missing Reversion Defects

Original statements of requirements frequently miss out on including details of reversions of behaviour that are needed to make the behaviour of a system complete. Systems that "behave" as opposed to programs that execute once and terminate must never get into a state from which no other behaviour is possible if such a situation arises the integrated requirements have a reversion defect. Take the case of the Microwave Oven DBT in figure 7. We find that if the Oven reaches either an OVEN[Cooking_Stopped] or an OVEN[Cooking_Finished] state then no further behaviour takes place. In contrast, when the system realizes an OVEN^[Cooking] leaf-node it "reverts" to a node higher up in the tree and continues behaving. To correct these two defects we need to append respectively to the R5 and R7 leaf nodes the two reversion nodes "^" shown in figure 9.

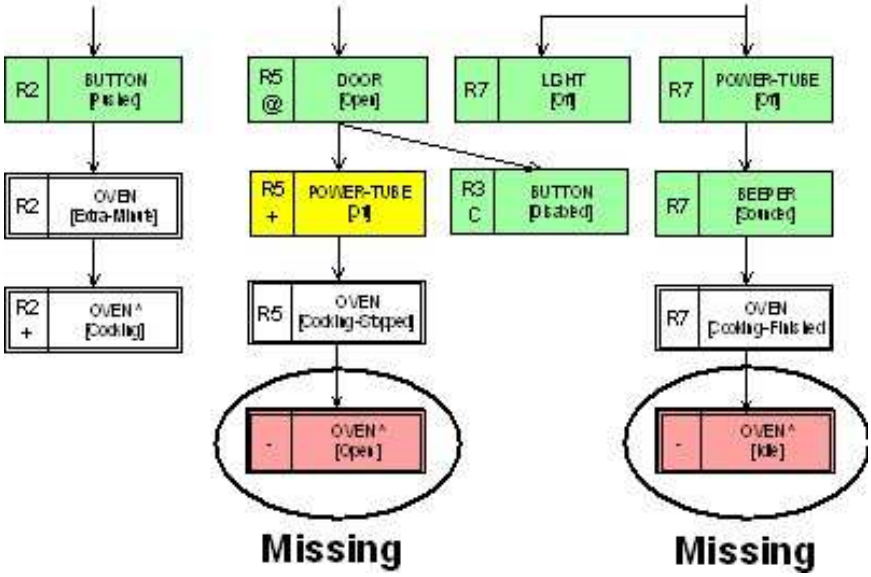

Fig. 9. Reversion "^" nodes added to make DBT behaviour reversion-complete

Deadlock, Live-lock and Safety Checking

The tool we have built allows us to graphically enter behavior trees and store them using XML[12]. From the XML we generate a CSP (Communi-

cating 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[14]. A similar strategy involves defining sub-processes in which state transitions for a component are treated as events. For example, to model the DOOR [Open] to DOOR [Closed] transition the following CSP was generated by the translation system:

$$\text{DoorOpen} = \text{userDoorClosed} \rightarrow \text{doorClosed} \rightarrow \text{DoorClosed}$$

The CSP generated by the tool is 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[14].

Reversion Inconsistency Defect Detection

The current tool does a number of consistency checks on a DBT. One important check to do is a reversion "$^\wedge$"check where control reverts back to an earlier established state. For example, for the Microwave Oven example in Figure 7, one reversion check that needs to be done is to compare the states of all components at OVEN[Idle] with those at OVEN$^\wedge$[Idle]. 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. Figure 10 shows the bottom part of the Oven DBT from Figure 7.

We see that requirement R7 (and the DBT in Figure 7) is silent on any change to the state of the BUTTON component. This means we have from R1 that BUTTON[Pushed] still holds when OVEN$^\wedge$[Idle] is realised. However this is inconsistent with OVEN[Idle] established by R6 and constraint R3 which has the state for BUTTON as BUTTON[Enabled]. That is, the system-state definitions which show up the inconsistency are as follows:

$$\text{OVEN[Idle]} \equiv \text{DOOR[Closed]} \wedge \text{LIGHT[Off]} \wedge \text{BUTTON[Enabled]} \wedge \ldots$$
$$\text{OVEN}^\wedge\text{[Idle]} \equiv \text{DOOR[Closed]} \wedge \text{LIGHT[Off]} \wedge \text{BUTTON[Pushed]} \wedge \ldots$$

These sort of subtle defects are otherwise difficult to find without systematic and automated consistency checking.

There are a number of other systematic checks that can be performed on a DBT, including the checking of safety conditions (e.g., in the Microwave Oven requirement R5, it indicates that the door needs to realize the state open to cause the power-tube to be turned off this clearly could be a safety concern). We will not pursue these checks here as our goal has only been

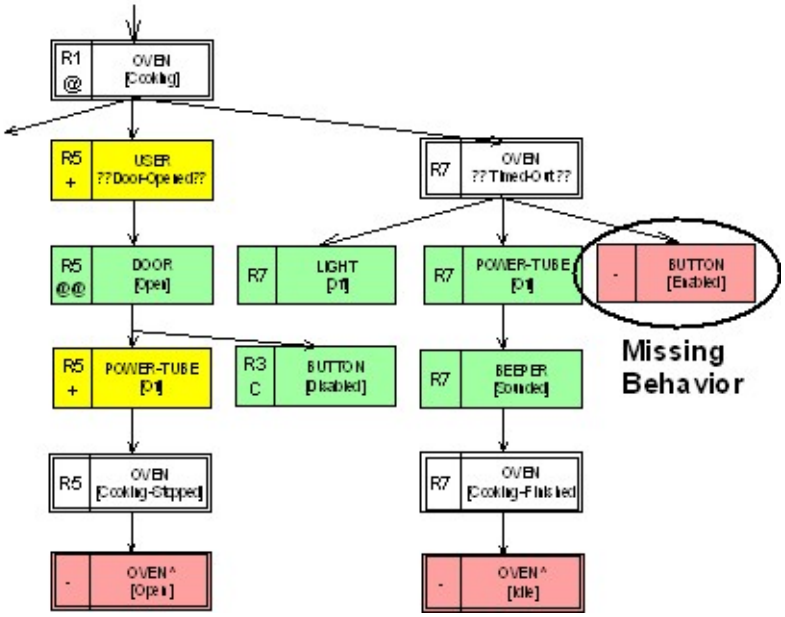

Fig. 10. Missing behaviour detected by checking OVEN[Idle]/OVEN^[Idle] component state consistency

to give a flavour of the sort of systematic defect finding that is possible with this integrated requirements representation. We claim, because of its integrated view, that a DBT probably makes it easier to "see" and detect a diverse set of subtle types of defects, like the ones we have shown here, compared with other methods for representing requirements and designs. We have found many textbook examples, where this is the case.

Once the design behavior tree (DBT) has been constructed, inspected and augmented/corrected where necessary, the next jobs are to transform it into its corresponding software or component architecture (or *component interaction network* - CIN) and project from the design behavior tree the component behavior trees (CBTs) for each of the components mentioned in the original functional requirements.

3.3. *Component Architecture Transformation*

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 component-state 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.*

In the DBT representation, a given component may appear in different parts of the tree in different states (e.g., in figure 7, the OVEN component may appear in the Open-state in one part of the tree and in the Cooking-state in another part of the tree). Interpreting what we said earlier in a different way, 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 (which may be thought of as a *specification* for the system) 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. *Informally, the process starts at the root of the design behavior tree and moves systematically down the tree towards the leaf nodes including each component 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 which each distinct component is represented only once and each component-component interaction is represented by a single line. We call this a Component Interaction Network (CIN) representation. Below, we show the eighth step of this transformation, involving the components on the eighth level (from the root) of the DBT. Here the POWER-TUBE gets included into the CIN and the link between the BUTTON and the LIGHT is added to the network.

Applying the tree-to-network conversion algorithm to level 8 of the DBT as shown in Figure 11 we see that the components, DOOR, BUTTON and LIGHT have been encountered earlier as have the DOOR→LIGHT and DOOR→BUTTON interactions. However the POWER-TUBE component has yet to be included in the evolving CIN. Also it is necessary to include the two interactions, BUTTON→LIGHT and BUTTON→POWER-TUBE as they have yet to be included. These level 8 inclusions are shown in the evolving CIN shown in Figure 12.

The complete Component Interaction Network derived from the Microwave Oven design behavior tree is shown below in Figure 13. It defines the component-component interactions for each component. It also captures the "business model" or "conceptual design" for the system and represents

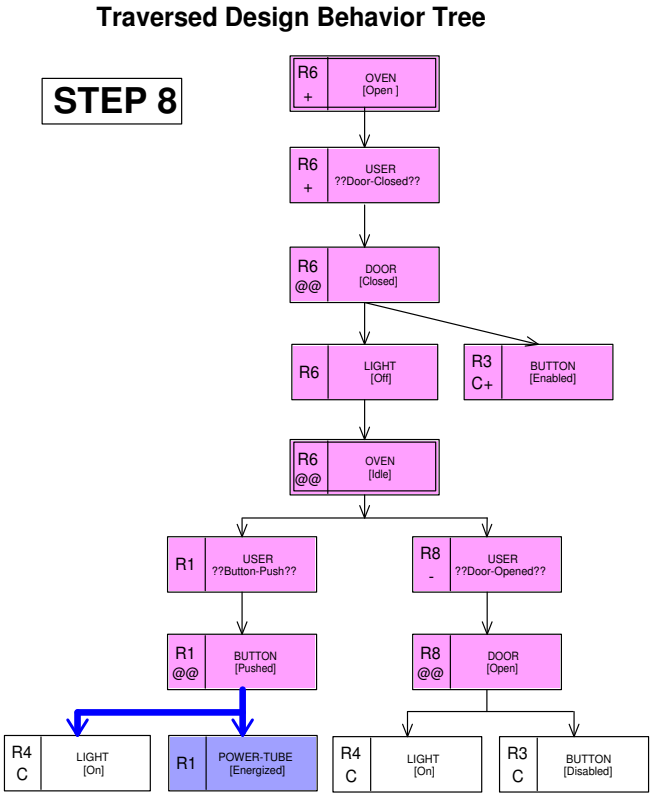

Fig. 11. Eighth step in converting Oven DBT into a component architecture

the "first cut" at the software architecture for a system. Why we say it is a first cut at the architecture is because it is sometimes possible to simplify the component interfaces and the number of interactions. For example, light has three "inputs". It only needs a single input to control its on/off status. In other situations, where there are a number of different interactions between two components it may be necessary to have more than one connection between two components (e.g., the interface between OVEN and USER requires two "arrows" to distinguish four distinct control inputs in the final component implementation architecture). Studying the network in figure 13, we note that the USER component interacts with only the DOOR and the BUTTON, as we would expect. This outcome was not something we consciously planned, but it is something that followed naturally from ac-

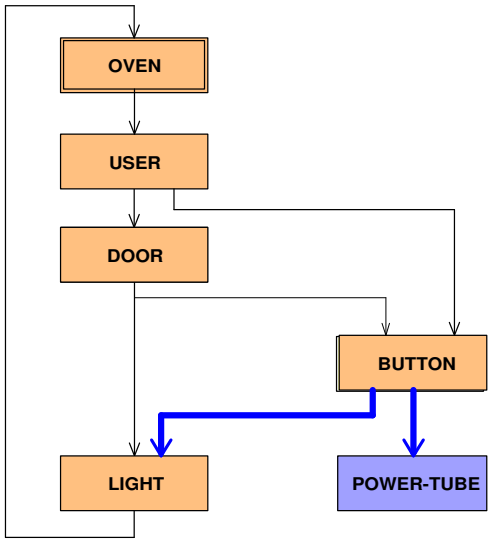

Fig. 12. Eighth step in converting Oven DBT into a component architecture

commodating the original requirements this shows the constructive power of the method for producing a semantically based system architecture.

The CIN provides the starting point for constructing a component-based design and implementation. It identifies the component interactions, subject to simplification and rationalisation. The job that remains is to identify the integrated behaviour of each of the components in the network, which conceptually we "embed" in each of the components in the network. We then have an integrated component design that can be easily refined into a component-based implementation. We will now describe how to isolate the behaviors of the individual components present in the architecture from the DBT using projection.

What we have focussed on presenting thus far is largely a mechanism for building a system out of components. It yields an architecture built out of a set of connected, visible (at that level), interacting components each of which encapsulates and executes behavior. What is important about this architecture is that it includes a *system-behavior-component* (the Oven in our example) which encapsulates the external behavior that the system exhibits. A system with this architecture has the important property that it can easily be used either:

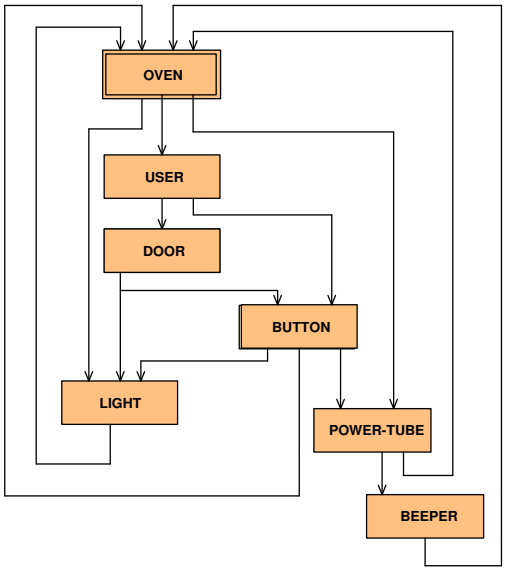

Fig. 13. Component Interaction Network (CIN) derived from the OVEN DBT

- as a standalone system, or
- as a component in a more complex system.

The inclusion of a system-behavior-component allows us to access the external behavior of the system without having any knowledge of its internal components and internal workings. All that is needed to use it is acquaintance with its external behavior. This facilitates reuse. Note that a *system-behavior-component* in this context is very different and separate from the "glue code", or *integration-component*, or other means needed to integrate the components that make up the system.

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 DBT in figure 7).

To implement components that can be embedded in, and operate within, the derived component interaction network, it is necessary to "concentrate" each components behavior. We can achieve this by systematically *projecting* each components behavior tree (CBT) from the design behavior tree.

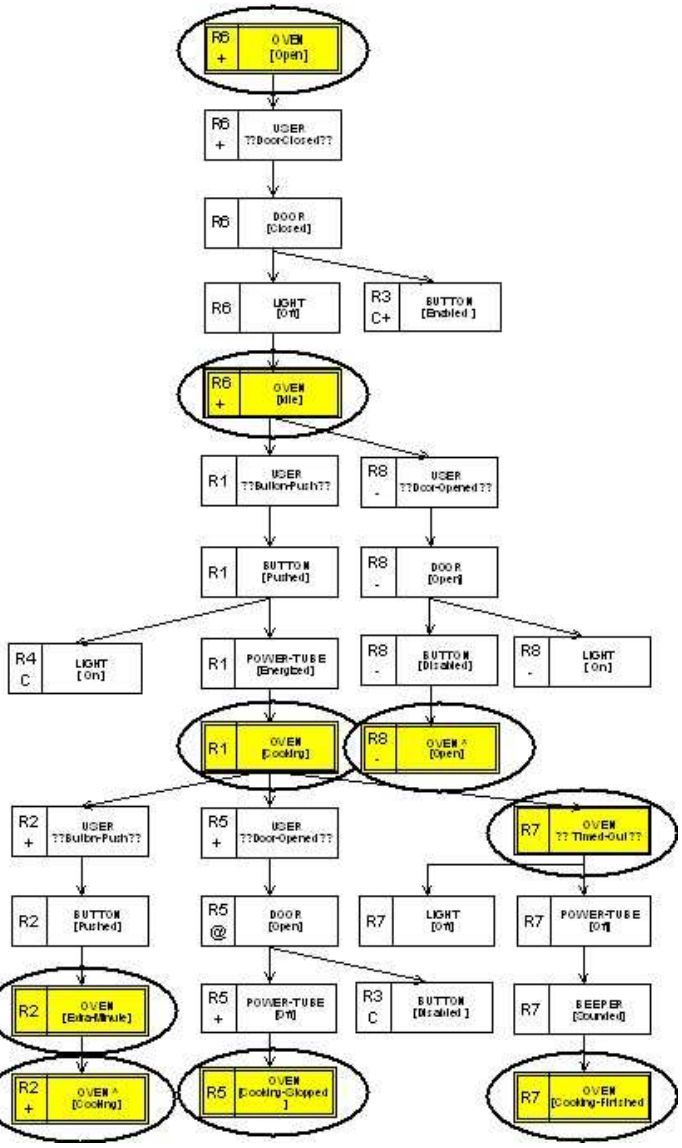

Fig. 14. Microwave Oven DBT with oven component behaviour highlighted

We do this essentially by ignoring the component-states of all components other than the one we are currently projecting. 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. When conducting each projection we need to preserve information that allows us to identify alternative behaviors that result from sets of either events and/or conditions.

Example Component Behavior Projection

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 7.

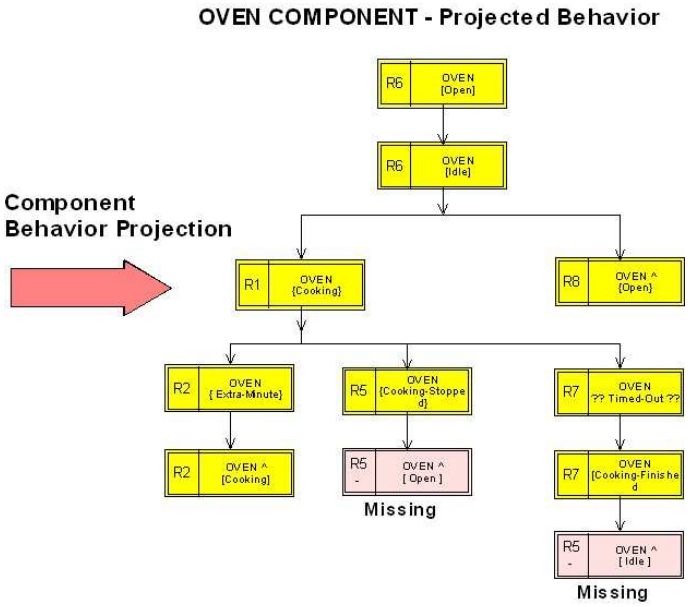

Fig. 15. Projected Behavior for the OVEN component derived from the DBT

In Figure 14 the OVEN component is highlighted in the DBT and the result of projection is shown in Figure 15. 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. When this process has been carried out for ALL the components in the DBT, that is, USER, BUTTON, etc, all the behavior in the DBT has been projected into the components

that are intended to implement the system. ***That is, the complete set of component behavior projections conserve the behavior that was originally present in the DBT.*** What this set of component projections allows us to achieve is a metamorphosis from an integrated set of functional requirements to an integrated component based design. It is worth commenting on what happens when we project out the behavior for the light component.

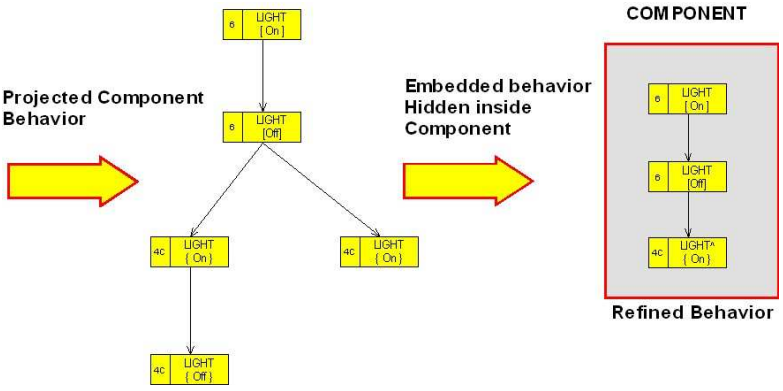

Fig. 16. Light component behaviour projection and simplification

What we see from this projection is that there is considerable "repeated" behavior that can be removed before embedding the light behaviour inside the light component. The implications of what has happened here are significant. What it means in general is that this component-based representation of the behaviour for a design factors out redundant and partially overlapping behavior, just as we have observed with the light component. This contrasts with object-oriented implementations where different scenarios that partially overlap each need to be separately implemented.

Component behavior projections frequently show up incompleteness and other defects, as is the case with the OVEN component projection figure 15. Missing is the behavior that should happen next when the cooking is stopped by opening the door - what should happen after cooking has finished? We see from this that projection provides another systematic way of finding and removing subtle requirements defects that are difficult to identify by other means. Leaf nodes for the OVEN component need to revert (^) back to earlier behavior in order for the components behavior to be

complete and consistent. For example, we need to add after OVEN[Cooking-Stopped] a reverting leaf node OVEN^[Open] that transfers control back to OVEN[Open] at the top of the CBT. And, after OVEN[Cooking_Finished] we need to add a reverting leaf node OVEN^[Idle] to make the oven behavior complete. Such defects may be caught at the reversion-check stage of the DBT, discussed earlier, or later at the component behavior projection stage as we have indicated here.

Component Behavior Design

Once we have projected the component behavior tree (CBT) for each component from the DBT and corrected any defects it is relatively straightforward to design the internal workings for a component together with its input/output interface. Below we show the corrected CBT for the button component.

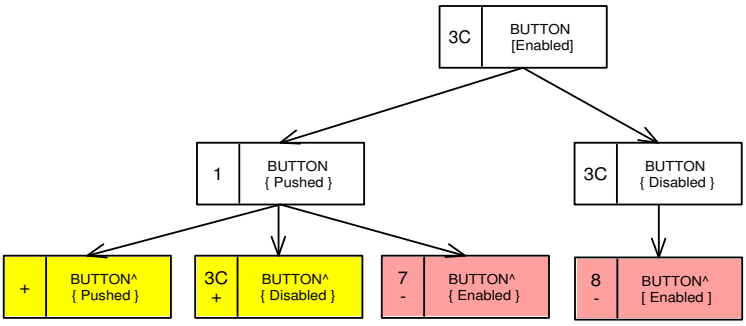

Fig. 17. Projected and reversion-corrected Component Behavior Tree for Button

We proceed with the internal design of the button component by identifying each of the possible "output-states" for the button and then asking, for each output-state from which internal button states can the component transition to the particular output state. The Button CBT directly provides this information. Take for example, the output state BUTTON[Disabled]. The CBT tells us that the Button component can transition to this state when it is either in a BUTTON[Pushed] or a BUTTON[Enabled] state. Below in the button component design we show how this information is recorded together with the transition information for all the other output states.

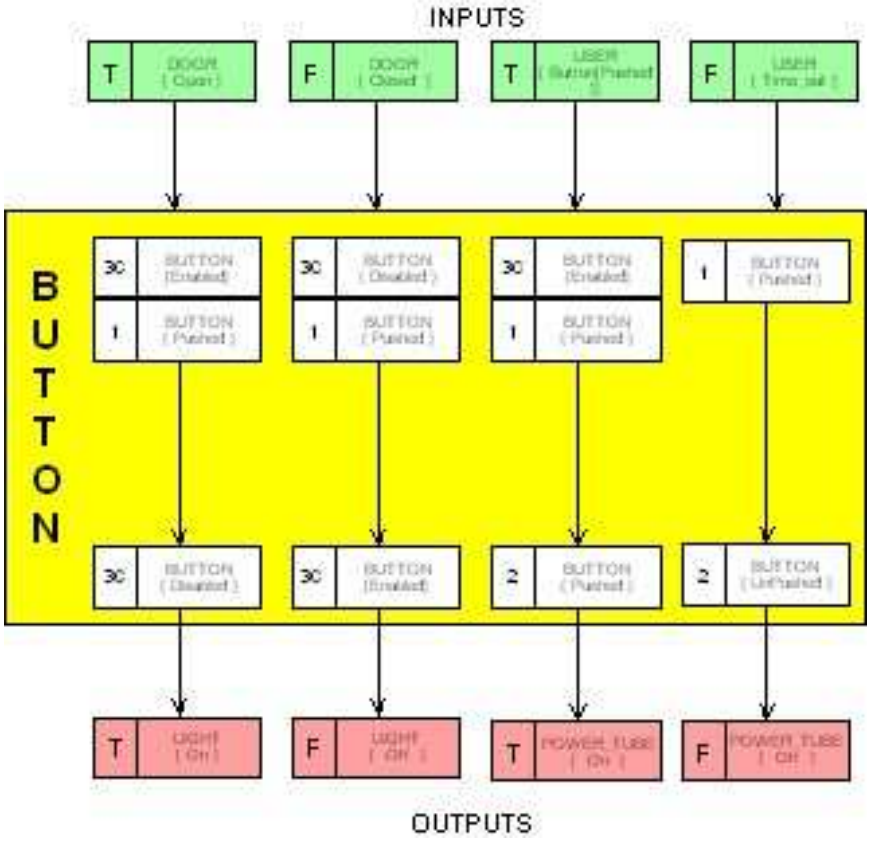

Fig. 18. Button Component internal state transitions derived from the Button CBT

In undertaking the design of the Button component we have chosen to simplify and identify the set of components that provide input to button. We have also done a similar thing with buttons outputs. Because in this example Button only receives control from other components and only passes control to other components we have used Booleans (**T** and **F**) as inputs and outputs. This gives the button component the same level of independence as a component in a hardware system would enjoy. It also allows us to clearly separate component implementation from the integration of the components in the system. In figure 19 the simplified input and output interactions needed for the button component are shown.

To complete the component-based design, we embed the behaviors of

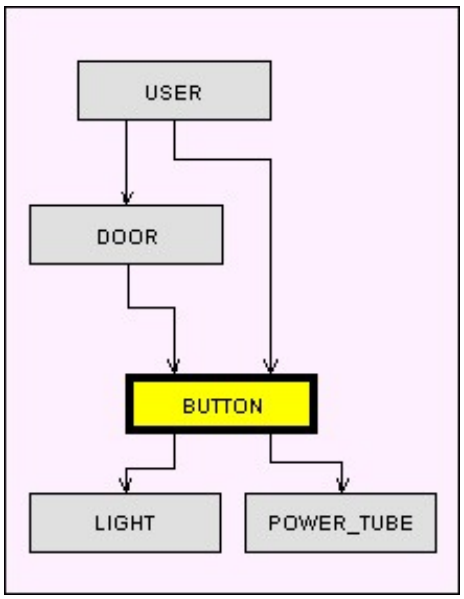

Fig. 19. Input and Output Context for the Button component

each component into the architectural design refined from the component interaction network (CIN). This involves simplifying or augmenting the component interfaces where needed and implementing the component interactions that deliver the integrated system behaviors. And finally, we must provide implementations to support the behaviors exhibited by each of the components. This is relatively straightforward to do from the component design. Component integration can be done using either the facilities of a component framework[1] or by mapping the graphic integrated network into a component-based code implementation.

The Microwave Oven problem has been previously studied in detail by Shlaer and Mellor[11].They employ a state transition diagram and a state transition table to model the behaviors. The state transition diagram (STD) bears some similarity to the projected behavior for the OVEN system component. However the STD is an explicit network form rather than a tree-like form with reversions. Events involving other components cause transitions between STD states. In contrast, using genetic design, the behavior of all other components in the system is incorporated directly in the DBT. Using STDs traceability to the original requirements is not direct and transpar-

ent. In going from original requirements to an STD additional behavior not specified in the original requirements has been added without comment (e.g. the behavior to allow the oven to be opened when it is idle). In genetic design, direct translation, defect detection, and augmentation of requirements are clearly separated steps. The use of STDs makes no provision for the determination of a problem-dependent architecture from the requirements or for the identification of behavior for other components. Instead the Shlaer and Mellor method proposes generic architectural classes for the finite state model, transition, timers and active instances (see [7], Chapter 9). In contrast, genetic design leads to an architecture and component behavior designs that are problem-dependent rather than generic.

3.5. *Systems Implemented on More than One Level*

The system component architecture that we have proposed (and illustrated using the Oven system) thus far only deals with behavior on a *single* level. Complex systems usually need to be able to deal with behavior on more than one level. For example, we might have a two-level Kitchen system that exhibits behavior at the Kitchen level but includes a Microwave Oven which exhibits the behavior given in our treatment of the problem above.

Once we open up the possibility of behavior on more than one level the two obvious questions to ask are: "(a) what is the role of sub-systems in such an architecture, and (b) how are sub-systems related to components?" Understanding of exactly what a sub-system is, often is not very clear. Consider, for example, the following statement from the literature about sub-systems. "A sub-system is not an object, nor a function, but a package of classes, associations, operations, events, and constraints that are interrelated and have a reasonably well-defined and (hopefully) small interface with other sub-systems" [10].

In the Unified System Model (USM), proposed here, things are clearer. The concept of a sub-system is unnecessary. What valid reason could there possibly be for discarding the notion of sub-system, a concept with such wide currency and acceptance? The answer lies in three things: (1) how we use standalone systems in more complex systems (e.g., how do we use the Microwave Oven system in a Kitchen system), (2) how we describe behavior at all levels, and that (3) at whatever level we describe a system, it is built out of a set of connected, visible (at that level), interacting components each of which encapsulates and executes behavior. (Note that in the Microwave system, the DOOR component is *visible*, but when we are describing the

behavior of the Kitchen System (see below), using the Microwave System component, the DOOR component becomes invisible because it is at a lower behavior level).

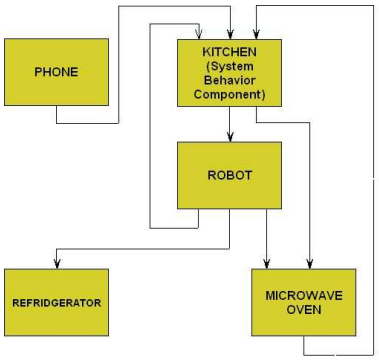

Fig. 20. Kitchen system that includes Microwave Oven system as a component

In response to (1), we may use what is a standalone system at one level of description, as a component (or more accurately a system-component) at the next higher level of behavioural description that is, the Microwave Oven system may be used as a component in the Kitchen system. Therefore system-components obviate the need for sub-systems. On the second point, fundamental to genetic design and the Unified System Model, is the idea that we describe behavior (and for that matter, requirements) in exactly the same way, at what ever level we are considering. For example, whether we are talking about the behavior of a Microwave Oven or the behavior of a Kitchen system that contains a Microwave Oven the treatment is exactly the same. It follows that if we loosen our definition of a system to say it is built out of connected, visible, interacting components and/or *system-components* then we have what we need. At whatever level we are describing a system it will be built out of connected components some of which may be systems in their own right. A consequence of this system description regime is that the need architecturally, or otherwise for sub-systems disappears. This model for a system seems preferable to some vague notion about what deserves to be a sub-system. The strategy also harmonises with the need for coarse-grained reuse of components. The word sub-system itself also gives us a semantic clue that reinforces this view there does not appear to be anything architecturally to distinguish a sub-system from a system

other than that it is part of a larger system. In a similar way, from an external view, in this model, there is nothing to distinguish a system from a component, both exhibit behavior and both have an external behavior interface.

One other very important point needs to be made about the description of behavior, and therefore the requirements of a system. On whatever level we are describing the system, the scope of the language we can use to describe requirements/behavior for this level is restricted to the components that are visible at that level, and the states that each of those components can realize no other information is relevant. We call this the *Behavioral Description Invariant*. Being mindful of, and employing this restriction simplifies the task of expressing requirements and behavior.

4. Comparison with UML and Other Methods

As Jackson observed, new notations and new design methods are generally not enthusiastically received [8]. 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 [13]. 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 following by component design. 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. In scaling up genetic design to larger systems we need to introduce *composition trees* that provide an integrated view of data requirements (c.f. function requirements and behavior trees) and *structure trees* that provide a formal integrated view of structures that behaviour takes place on (e.g., a rail network). We also focus on deriving an initial, high-level, integrated system behavior tree (SBT) from the original requirements to gain cognitive control of the systems behaviour before considering the behaviour of requirements in detail. Because of space limitations presentation of these aspects of the method will not be pursued here.

A comparison of Behavior Trees with Statecharts has been published elsewhere[5]. A separate comparison with Cleanroom Software Engineering[9] is available from the author.

Considerable thought has gone into whether it is appropriate to use the term "genetic design" given the established use of the term "genetic algorithms" in a different context. The parallels of the proposed method with key genetic principles spelled out in Woolfsons recent book[15] gives considerable justification to the claim that "genetic" is being accurately used here. The way behavior tree integration can result in the evolutionary growth of a design adds weight to the genetic characterization of the method. Genetic design exploits three fundamental genetic properties of a set of functional requirements that are revealed and become easily accessible when they are expressed and then integrated as behavior trees. It is these *emergent properties* that give the method its constructive power. Things may be summed up with the words of eighteenth century thinker Giambattista Vico, who said, "*To understand something, and not merely be able to describe it, or analyse it into its component parts, is to understand how it came into being its genesis, its growth ... true understanding is always genetic*".

5. Conclusion

To advance the discipline of software engineering four major problems need to be addressed. 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. This very significantly reduces the short-term memory overload problem that has plagued software development for so long.

A clarification of the steps to go from a set of requirements to a design is also central to advancing the practice of software engineering. Presently there would appear to be too much choice at every stage in terms of which process to follow, which notation(s) to use and which tools to employ. The root cause of this uncertainty seems to be a lack of a clear understanding of the relationship between a set of requirements and a design that will satisfy those requirements. The suggestion to build a design out of its requirements, directly leads to a clarification and a simplification of the design process, and a reduction in the need for different notations. It also guarantees direct traceability of original statement of requirements. That the component ar-

chitecture and individual component behavior designs are both emergent properties of the integrated requirements (the design behavior tree) represents a further simplification, systematization and a clarification of the design process.

Early detection of requirements defects is another very significant problem that thwarts software engineering practice. Requirements translation, requirements integration and component behavior projection coupled with both manual and automated analysis/inspection of design behavior trees offer a powerful set of techniques for early requirements defect detection. In particular, integration of requirements behavior trees turns out to a very effective way of uncovering otherwise obscure defects because it forces us to consider each requirement directly in the context where it is used behaviorally.

Yet another thorny challenge for software engineering is how to transition from a loose informal natural language statement of functional requirements to a formal representation. Unless this transition approaches repeatability all subsequent development within a formal framework is undermined because we may not be preserving the original intention. With many development approaches, when this barrier is crossed, we frequently find some things get left out, new things get added in and in other cases things are modified. In contrast, with behavior trees, because the focus is on translation, it follows that the emphasis is on meaning and on the preservation and clarification of intention. Although ambiguity is always a threat to repeatability, rigorous translation approaches repeatability when carried out by different translators.

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 do more sophisticated graphics, multi-user editing, vocabulary control, and consistency checking.

In summary, what we have presented is an intuitive, stepwise process for going from a set of functional requirements to a design. The method is attractive for its simplicity, its traceability, its ability to detect defects, its control of complexity, and its accommodation of change.

Acknowledgements

This work was supported by the Australian Research Council through a number of research grants. I would like to thank Xuelin Zheng, Lian Wen, Cameron Smith, David Billington, Zoran Milosevic and Dan Powell for many useful discussions on this work. I would also like to thank my colleagues in the SQI, in particular Bruce Hodgen, Don Abel, Angela Tuffley and Terry Rout and more broadly in Griffith University, Rodney Topor and Chengzheng Sun for their encouragement and I would like to thank my students Kate McClung, Liam Casey, Chris English, David Tannock, Elenkayer Sithirasenan, Casey Ackworth, David Whyte, Chris Mclean, Saad Zafar Michael Ransom-Smith, Brian Pack, Ashley Forsyth, John Seagrott, Henrik Hansen, Thomas Jansen and my software engineering students for their efforts in trialling the genetic design method and giving me plenty of useful feedback. Peter Lindsay, Ian Hayes, Kirsten Winter, David Carrington and Nisansala Yatapanage my colleagues from the ARC Centre for Complex System at University of Queensland are also thanked for their support and useful discussions as is David Abramson from Monash University. I would also like to thank David Harel and Rudy Marelly from the Weizmann Institute for their comments on an early version of this work and for sharing their pre-published work in this area. Brian Henderson-Sellers and Cesar Perez from the University of Technology, Sydney are thanked for their valuable work on meta-modelling and Bill Waite from the University of Colorado is thanked for his input on the Behavior Tree grammar. Adrian Pitman and Shireane McKinnie from the Australian Defence Materiel Organisation (DMO) are thanked for their support for supplying several defence projects. Terry Stevenson and Nev Delap from Boeing are especially thanked for their support and encouragement as are Kim Olsen from Queensland Rail, Adrian Mortimer from Emilex, James Ross from Telelogic, Peter Thornton from Department of Foreign Affairs, Duncan Cross from Suncorp, Brendan Lovelock, Linda Mathews and Steve Plant from Signature Software, and Mark Rheinlander, Shawn Parr and Rob Whitney from Calytrix Technologies.

References

1. G.Booch, J.Rumbaugh, I.Jacobson, The Unified Modelling Language User Guide, Addison-Wesley, Reading, Mass. (1999).
2. A.M.Davis, A Comparison of Techniques for the Specification of External System Behavior, Comm. ACM, vol. 31 (9), 1098-1115, (1988).
3. R.G.Dromey, From Requirements to Design: Formalizing the Key Steps, In-

ternational Conference on Software Engineering and Formal Methods, (Invited Keynote Address), Brisbane, September, (2003).
4. R.G.Dromey, Using Behavior Trees to Model the Autonomous Shuttle System, 3rd International Workshop on Scenarios and State Machines: Models, Algorithms and Tools, (SCESM04), Edinburgh, May, (2004).
5. R.G.Dromey, Behavior Trees: Amplifying Our Ability to Deal With Requirements Complexity, Proceedings Dagstuhl Seminar, September 2003, Scenarios: Models, Transformations and Tools, Lecture Notes in Computer Science, Edited by, Francis Bordeleau, Stefan Leue, and Tarja Systa (to appear).
6. D.Harel, Statecharts: Visual Formalism for Complex Systems, Sci. Comp. Prog., 8, 231-274 (1987).
7. D. Harel., W. Damm, LSCs: Breathing Life into Message Sequence Charts, 3rd IFIP Conf. On Formal Methods for Open Objected-based Distributed Systems, New York, Kluwer (1999).
8. D. Jackson, Alloy: A Lightweight Object Modelling Notation, MIT Lab. for Comp. Sci. Report (1999).
9. S.J. Prowell, C.J.Trammell, and R.C. Linger, J.H. Poore, Cleanroom Software Engineering: Technology and Process, Addison-Wesley, Reading Mass., (1999).
10. J.Rumbaugh, M.Blaha, W.Premerlani, F.Eddy, W.Lorensen, Object-Oriented Modeling and design, Prentice-Hall, Englewood Cliffs, NJ, (1991).
11. S. Shlaer, S.J. Mellor, Object Lifecycles, Yourdon Press, New Jersey, (1992).
12. C. Smith, K.Winter, I.Hayes, R.G. Dromey, P.Lindsay, D.Carrington, An Environment for Building a System Out of Its Requirements, 19th IEEE International Conference on Automated Software Engineering, Linz, Austria, Sept. (2004)
13. L.Wen, R.G.Dromey, From Requirements Change to Design Change: A Formal Path, SEFM04, IEEE International Conference on Software Engineering and Formal Methods, Beijing, September, (2004).
14. K.Winter, Formalising Behavior Trees with CSP, International Conference on Integrated Formal Methods, IFM04, LNCS vol. 2999, 148 167 (2004).
15. A.Woolfson, Life Without Genes, Flamingo, (2000).