From Requirements to Design: Formalizing the Key Steps

R.G. Dromey,
Software Quality Institute, Griffith University,
Nathan, Brisbane, Qld., 4111, AUSTRALIA
rgd@cit.gu.edu.au

Abstract

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, repeatable and constructive as we would like. Progress with this fundamental problem is possible once we recognize that individual functional requirements represent fragments of behaviour, while a design that *satisfies* a set of functional requirements represents integrated behaviour. This perspective admits the prospect of constructing a design *out of* its requirements. A formal representation for individual functional requirements, called *behavior trees* makes this possible. Behaviour trees of individual functional requirements may be composed, one at a time, to create an integrated design behaviour tree. From this *problem domain* representation it is then possible to transition directly and systematically to a *solution domain* representation of the component architecture of the system and the behaviour designs of the individual components that make up the system – both are emergent properties.

"Finding deep simplicities in a complex logical task leads to work reduction"- Harlan Mills.

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 suggest that it is safe to conclude individual requirements <u>express</u> *constrained behaviour.* By comparison, a system that satisfies a set of functional requirements <u>exhibits</u> *integrated constrained behaviour.* The latter behaviour of systems is not inherently different. Therefore, can the same formal representation of behaviour 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 that are assigned values
> - Components, by changing states, can cause other components to change their states
> - Supplementing these component-state primitives are conditions/decisions, and events involving component-states.
> - Interactions between components also play a key role in describing behaviour. They involve 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], state-charts [3], and Message Sequence Charts (MSCs) [4], accommodate some or all of the behaviour we find expressed in functional requirements and designs. Individually however, none of these notations provide the level of constructive support we need. This forces us to contemplate another representation for functional requirements and designs. As Jackson wisely remarked [5], 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.
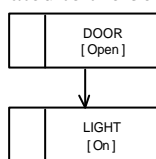
2. Behavior Trees

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.

Definition: *A Behavior Tree is a formal, tree-like graphical form that represents behaviour 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 basis, e.g., the sentence "when the door is opened the light should go on" is translated to the behaviour tree below:

The principal conventions of the notation for component-states are the graphical forms for associating with a component a [State], ??Event??, ?Decision?, [Sub-cpt[State]], or [Attribute := expression | State]. Exactly

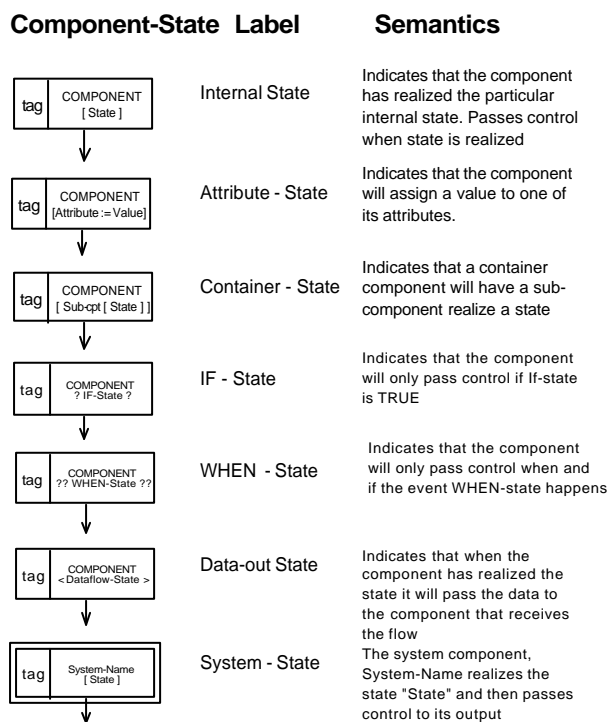| Component-State	Label	Semantics
	Internal State	Indicates that the component has realized the particular internal state. Passes control when state is realized
	Attribute - State	Indicates that the component will assign a value to one of its attributes.
	Container - State	Indicates that a container component will have a sub-component realize a state
	IF - State	Indicates that the component will only pass control if If-state is TRUE
	WHEN - State	Indicates that the component will only pass control when and if the event WHEN-state happens
	Data-out State	Indicates that when the component has realized the state it will pass the data to the component that receives the flow
	System - State	The system component, System-Name realizes the state "State" and then passes control to its output

Figure 1. Behavior Tree Notation, key elements

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 definitions and other constraints are signalled by a "/". System states, are used to model high-level (abstract) behavior, some preconditions/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 1, above (see EBNF, semantics, web-site http://www.sqi.gu.edu.au/gse/papers).

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 Fig. 4 below. 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 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 users or clients and developers/maintainers. 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 requirements and its corresponding design. And from this follows a systematic method for constructing a design from its requirements.

3. Genetic Software Engineering Method

Conventional software engineering applies the underlying design strategy of constructing a design that will <u>*satisfy*</u> 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 <u>*out of*</u> its set of functional requirements, by integrating the behavior trees for individual functional requirements (RBTs), one-at-a-time, into an evolving design behavior tree (DBT). This very significantly reduces the complexity 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. This property is the exact same property that a set of

pieces for a jigsaw puzzle possess. And, interestingly, it is the same property which a set of genes that create a living entity possess. Witness the remark by geneticist Adrian Woolfson: in his recent book ([6], 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 behaviour expressed in functional requirements does not "just happen". There is always *a precondition* that must be satisfied in order for the behaviour encapsulated in a functional requirement to be accessible or applicable or executable. In practice, this precondition may be embodied in the behaviour 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 behaviour 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 behaviour 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 behaviour trees in any order, one at a time, an analogous requirement is needed. We have already said that a functional requirement's precondition needs to be satisfied in order for its behaviour to be applicable. It follows that some *other* requirement, as part of its behaviour tree, must establish the precondition. This requirement for composing/integrating functional requirements expressed as behaviour 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
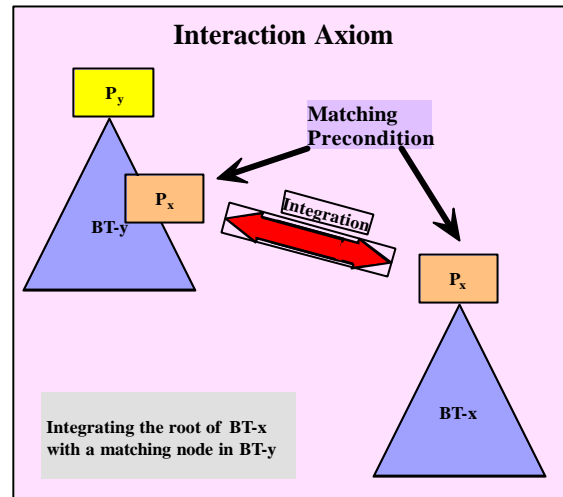

Figure 2. Interaction Axiom - graphic form

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 behaviour trees. We may then proceed to integrate those behaviour 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 behaviour projection

Each overarching step, needs to be augmented with a verification 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.
.

To maximize communication our intent here is therefore to only introduce the main ideas of the method, and do so in a relatively informal way. The whole design process is best understood in the first instance by observing its application to a simple example. 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 [7]. The seven stated functional requirements for the Microwave Oven problem [7, p.36] are given in the table below. Shlaer, and Mellor have applied their state-based Object-Oriented Analysis method to this set of functional requirements.

Table 1. Functional Requirements for Microwave Oven

R1. There is a single control button available for the user 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 is finished.

3.1 Requirements Translation

Requirements translation is the first formal step in the Genetic Software Engineering (GSE) design process. Its purpose is to translate each natural language functional requirement, one at a time, into one or more behaviour 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.

Example Translation

The translations for the first six functional requirements for the Microwave Oven given in Table 1 are shown in figure 4. Translation of R7 from Table 1 will now be considered in slightly more detail. For this requirement we have underlined the states/actions and made the **components** bold, i.e., "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 3. (see below) 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.
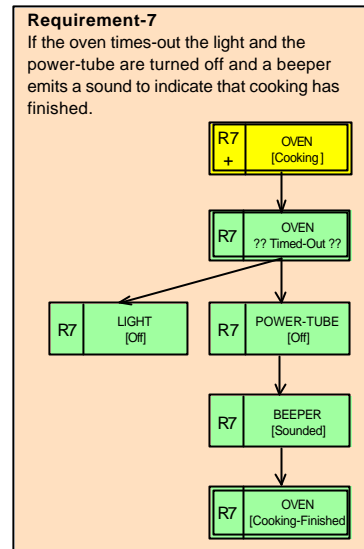

Figure 3. Behavior Tree for Requirement R7

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 addition, the behavior tree representation tags (here R7) are able to provide very 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, and introduce new things as they do an initial translation. Once the initial translation has been done it is then necessary to carry out systematic inspections of each individual requirement behavior tree (RBT) to identify missing preconditions, missing alternate cases, redundancy and/or other completeness or inconsistency problems. We should also determine whether each RBT has a precondition that enables it to be integrated to make a complete system. We do this by checking whether the root node of each RBT occurs in another RBT. If it does not it is either the root node of the whole system, or it requires a precondition to be added (as was the case with R7 above, where we added the precondition OVEN[Cooking]) or the set of requirements is incomplete or there is behavior missing from the requirement it needs to integrate with. These integration checks allow us to find and rectify many otherwise subtle defects in the behavior of a set of requirements. This is particularly important with systems that have a large number of functional requirements. *Why is translation of individual functional requirements to one or more behavior trees feasible?* We suggest this is so because individual functional requirements of interest express constrained but implementable behavior. This implies that such behavior can ultimately be implemented in a programming language. The behavior tree notation
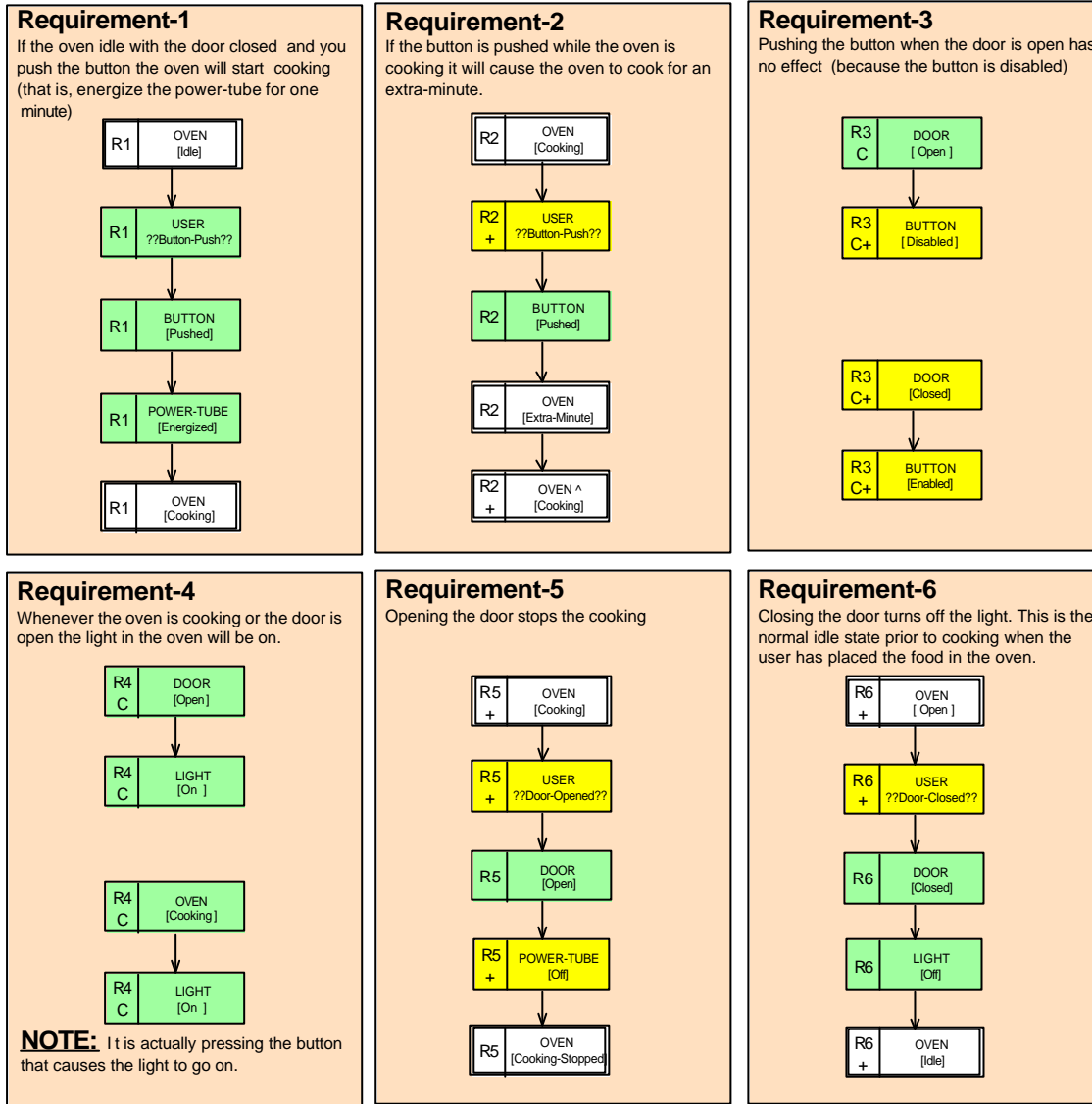
Requirement-1
If the oven idle with the door closed and you push the button the oven will start cooking (that is, energize the power-tube for one minute)

R1	OVEN [Idle]
R1	USER ??Button-Push??
R1	BUTTON [Pushed]
R1	POWER-TUBE [Energized]
R1	OVEN [Cooking]

Requirement-2
If the button is pushed while the oven is cooking it will cause the oven to cook for an extra-minute.

R2	OVEN [Cooking]
R2 +	USER ??Button-Push??
R2	BUTTON [Pushed]
R2	OVEN [Extra-Minute]
R2 +	OVEN ^ [Cooking]

Requirement-3
Pushing the button when the door is open has no effect (because the button is disabled)

R3 C	DOOR [Open]
R3 C+	BUTTON [Disabled]
R3 C+	DOOR [Closed]
R3 C+	BUTTON [Enabled]

Requirement-4
Whenever the oven is cooking or the door is open the light in the oven will be on.

R4 C	DOOR [Open]
R4 C	LIGHT [On]
R4 C	OVEN [Cooking]
R4 C	LIGHT [On]

NOTE: I t is actually pressing the button that causes the light to go on.

Requirement-5
Opening the door stops the cooking

R5 +	OVEN [Cooking]
R5 +	USER ??Door-Opened??
R5	DOOR [Open]
R5 +	POWER-TUBE [Off]
R5	OVEN [Cooking-Stopped]

Requirement-6
Closing the door turns off the light. This is the normal idle state prior to cooking when the user has placed the food in the oven.

R6 +	OVEN [Open]
R6 +	USER ??Door-Closed??
R6	DOOR [Closed]
R6	LIGHT [Off]
R6 +	OVEN [Idle]

Figure 4. Behavior trees for Microwave Oven

contains, as a subset of its expressive capability, a graphic representation for basic logical forms and for sequence, selection, iteration, data-flow and assignment - the core building blocks of programming languages. It follows behavior trees have enough expressive power to capture implementable behavior described in individual functional requirements.

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 behaviour 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 behaviour trees. We only ever attempt to integrate two behaviour trees at a time – either two RBTs, an RBT with a DBT or two partial DBTs. In some cases, because the precondition for executing the behavior in an RBT has not been included, or important behaviour has been left out of a requirement, it is not clear where a requirement integrates into the design. This immediately points to a problem with the requirements. In other cases, there may be requirements/behaviour missing from the set which prevents integration of a requirement. Attempts at integration uncover such problems with requirements at the earliest possible time.

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 behaviour tree (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 below.
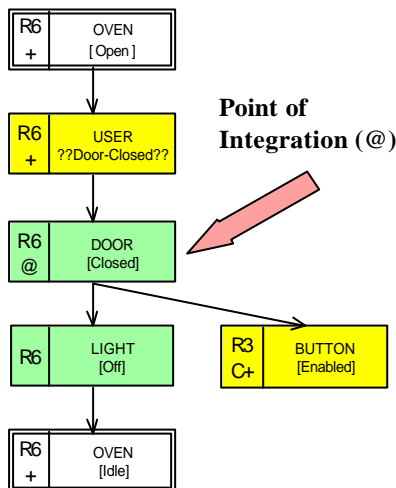

Figure 5. Result of Integrating R6 and R3C

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 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 are left with a single final DBT (see Figure 6 below). 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 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. From a careful inspection of the integrated DBT (Fig. 6) we see that there is a missing requirement associated with opening the oven when it is idle. This has been added as requirement R8. Note with constraint R4 we have used the causal relationship for the light turning on rather than the literal translation of the requirement.
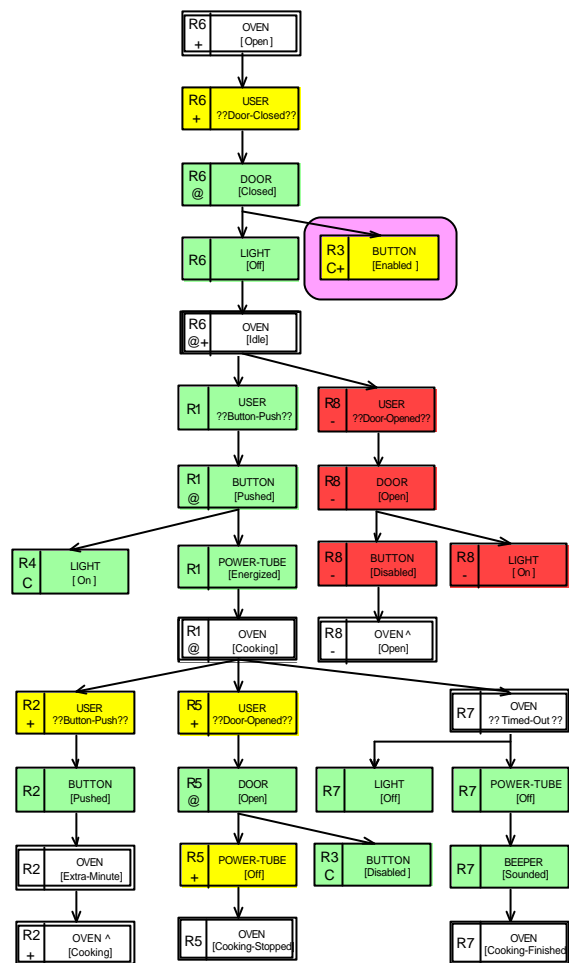

Figure 6. Integration of all functional requirements

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

3. 3 Software 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., the OVEN component may appear in the Open-state in one part of the
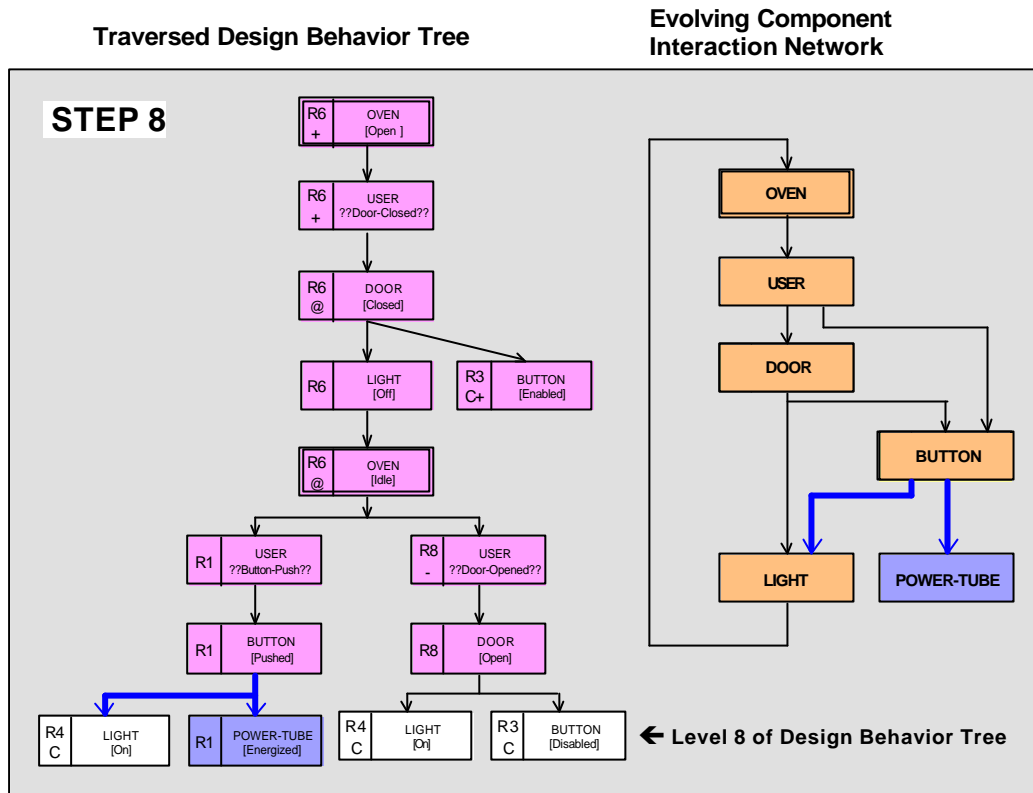
Figure 7. A step in the Tree-to-Network Transformation

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 <u>once</u>. 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. *Informally, the process starts at the root of the design behavior tree and moves systematically down the tree towards the leaf nodes including <u>each component</u> and <u>each component interaction</u> (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 <u>once</u>. We call this a Component Interaction Network (CIN) representation. Above, we show the eighth step of this transformation, involving the components on the eighth level 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.

The complete Component Interaction Network derived from the Microwave Oven design behavior tree is shown below in Figure 8. It defines the component-component

interactions and therefore the interfaces for each component. It also captures the "business model" or "conceptual design" for the system and represents the first cut at the software architecture for a system. Studying the network in figure 8, 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 accommodating the original requirements – this shows the constructive power of the method for producing a semantically based system architecture. The next important task is to isolate the behaviours of the individual components present in the architecture from the DBT using projection.

3.4 Component Behavior Projection

In the design behavior tree, the behavior of individual components tends to be dispersed throughout the tree (for example, see the OVEN component-states in the Microwave Oven System DBT). To implement components that can be embedded in, and operate within, the derived component interaction network, it is necessary to "concentrate" each component's behavior. We can achieve this by systematically *projecting* each component's behavior tree (CBT) from the design behavior tree. We do this by simply 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.
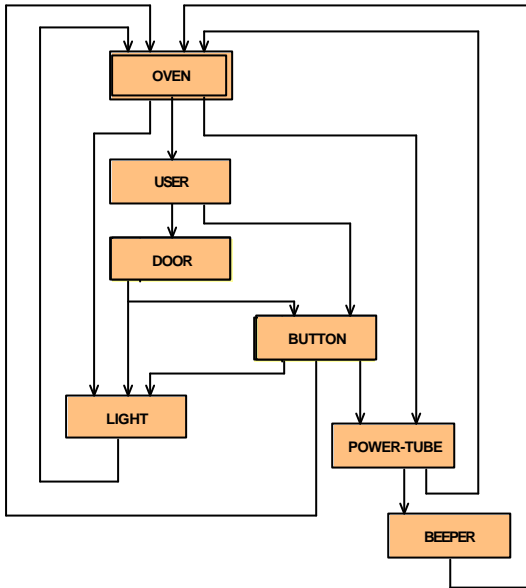

Figure 8. Component Interaction Network - (CIN)

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 9 below the OVEN component is highlighted in the DBT on the left of the figure and is projected on the right of the figure. 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. To complete the component-based design, we embed the behaviors of each component into the architectural design provided by the component interaction network (CIN) – see, for example figure 8 above. The tasks that then remain are to rationalize the component interfaces and to implement the component interaction network which supports the component interactions that, in turn, implement the system behaviors. And finally, we must provide implementations to support the behaviors exhibited by each of the components. Component integration can be done using either the facilities of a component framework [1] or by using a

standard code implementation that maps the graphic network into code.

Component behavior projections frequently show up incompleteness and other defects, as is the case with the OVEN component projection – figure 9. Missing is the behavior that should happen next when the cooking is stopped by opening the door and 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 behaviour in order for the component's behaviour 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 behaviour complete.

3.5 Comparison With Other Methods
The Microwave Oven problem has been previously studied by Shlaer and Mellor [7]. They employ a state transition diagram and a state transition table to model the behaviors of the Microwave Oven. The state transition diagram (STD) bears some similarity to the GSE projected behaviour for the OVEN system component. However the STD is a network rather than a tree. Events involving other components cause transitions between STD states. In contrast, using the GSE method, the behaviour of all other components in the system is incorporated directly in the DBT. Using STDs traceability to the original requirements is not direct and transparent. In going from original requirements to an STD additional behaviour not specified in the original requirements has been added without comment (e.g. the behaviour to allow the oven to be opened when it is idle). In GSE, 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 behaviour 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, GSE leads to an architecture and component behaviour designs that are problem-dependent rather than generic. GSE has similar advantages over MSCs [4] and Statecharts [3]. In a number of reports and presentations at we provide a more detailed account of the GSE method, the notation and its application to a diverse set of problems including contract automation and much larger applications. We also provide examples that show how to translate the designs that the method produces into object-oriented and component-based implementations in Java.

Figure 9. Component Behavior Projection from the Design Behavior Tree

4. Discussion

In contrast to other methods that are used for object-oriented design the GSE method relies on the use of a single notation, behavior trees, to represent (1) behavior in individual functional requirements (or use-cases), (2) the integrated behavior of a set of requirements and (3) the projected behavior of individual components. Behavior trees capture what is usually spread across sequence diagrams, activity diagrams, class diagrams and state-machines. The initial focus in GSE is on translation, integration of functional requirements and defect detection. This contrasts with the strong focus on identifying objects and classes in more traditional object oriented design. It is important to note that the Behavior Tree Notation which forms the backbone of the GSE method can be formalised using an extension to Dijkstra's Weakest Precondition theory [8]. Our intention, in the limited space here, is to show this formalization only for a key element of the notation. Gries [8], used the following weakest precondition formalization for an assignment "x:=e" establishing a postcondition R, i.e.,

$$wp (x := e , R) \equiv R[e/x] \wedge def(e) \quad \text{-- (1)}$$

In the behaviour tree notation for a component C which realises a defined and allowable state s, (i.e, textually C[s]) and establishes a postcondition R , at time t, we have:

$$wp(C[s] , R , t) \equiv R[s / C.t] \wedge def(s) \quad \text{-- (2)}$$

Applying (2) for a component state X[s1], a postcondition X.t1 = s1, and assuming s1 is defined we have:

$$wp (X[s1] , X.t1 = s1 , t1)) \equiv X.t1 = s1[s1 / X.t1]$$
$$\equiv (s1 = s1) \equiv TRUE$$

This shows a component X, realizing a state s1, at a particular time t1, corresponds to the establishment of a logical equality X.t1 = s1. Weakest preconditions can

characterize the semantics of the other component-states from the behaviour tree notation and accommodate sequential composition, etc, associated with composing component states. This permits association of formal specifications with DBT nodes, e.g. for OVEN[Idle]:

OVEN[Idle] ≡ DOOR[Closed] ∧ LIGHT[Off] ∧ BUTTON[Enabled].

What we have shown here is just a simple, informal "textbook" application of GSE. An obvious question to ask is does the GSE method scale-up to larger applications? At the website given above we have documented the construction of the DBT for a Satellite Control System that has been studied by Lockheed-Martin using Objectory and Prowell, et al, from the SEI [9] using the Cleanroom method. It has a 15-page functional requirements specification. We have also successfully applied the "large-scale" adaptation of the GSE method (see website) to a web-based car-fleet management system whose implementation required over 160 web-pages. Currently we are applying the method to a much larger system – part of the Orion AP-3C reconnaissance aircraft software functional requirements. The results have been promising particularly for the control of complexity that is achieved and for the effectiveness in the detection of integration, redundancy and other defects in the original requirements.

Another obvious question is, does the behavior tree notation have enough expressive power to compete with extensive composite notations like UML? No half-page argument can settle this question. Our response is to claim that behavior trees can capture what is expressed in use-cases, class diagrams, sequence diagrams, activity diagrams and state machines. The incorporation of graphic logical forms, including a disjunctive normal form, provides enough power to accommodate case analyses, decisions, and associated and/or complexity.

Considerable thought has gone into whether it is appropriate to use the term "genetic software engineering" 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 Woolfson's recent book [6] 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.

Conclusion
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. However, like any method, Genetic Software Engineering, will only work well when applied rigorously. Derivation of the software component

architecture from the design behavior tree and projection of the set of component behavior trees from a design behavior tree are both repeatable, algorithmic processes, that can be automated if we choose to do so. The greatest chance for variation with work products comes in the translation of natural language descriptions of functional requirements to requirements behavior trees (RBTs). Variance can be minimized by making sure all components and all behaviors in a given textual description are accommodated in the translated RBT. Best results are obtained by getting two people to translate each requirement and then resolve their differences. Requirements integration is also potentially algorithmic and automatable, but defects in requirements need software engineers to resolve such problems.

Returning to our original motivations from genetics and jigsaw puzzles, we have seen that Genetic Software Engineering 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 behaviour 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*".

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] D.Harel, Statecharts: Visual Formalism for Complex Systems, Sci. Comp. Prog., 8, 231-274 (1987)

[4] D. Harel., W. Damm, LSCs: Breathing Life into Message Sequence Charts, 3rs IFIP Conf. On Formal Methods for Open Objected-based Distributed Systems, New York, 1999, Kluwer

[5] D. Jackson, Alloy: A Lightweight Object Modelling Notation, MIT Lab. for Comp. Sci. Report (1999)

[6] A. Woolfson, Life Without Genes, Flamingo, (2000).

[7] S. Shlaer, S.J. Mellor, Object Lifecycles, Yourdon Press, New Jersey, 1992.

[8] D. Gries, The Science of Programming, Springer-Verlag, (1981).

[9] S.J. Prowell, C.J.Trammell, R.C. Linger, J.H. Poore,
Cleanroom Software Engineering: Technology and Process,
Addison-Wesley, Reading Mass., (1999).