Architecture as an Emergent Property of Requirements Integration

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, how to go from a set of functional requirements to an architecture that accommodates those requirements remains a challenging problem. Progress with this fundamental problem is possible once we recognize (1) that individual functional requirements represent fragments of behaviour, (2) a design that *satisfies* a set of functional requirements represents integrated behaviour, and (3) an architecture must accommodate the integrated behaviour expressed in a set of functional requirements. 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 go in a systematic way from a set of functional requirements to a design that will satisfy those requirements and an architecture that will support the implied integrated behavior. In practice, these two tasks are 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 it is safe to conclude that individual requirements express *constrained behaviour*. By comparison, a system that satisfies a set of functional requirements exhibits *integrated constrained behaviour*. The latter behaviour of systems is not inherently different.

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 and Message Sequence Charts (MSCs) [3,4], accommodate 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. Such ventures are generally not enthusiastically received – a consensus is that new notations 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 path for going first from a set of functional requirements to an integrated behaviour representation that will satisfy those requirements and then to an architecture and the set of accompanying component behaviour designs [5].

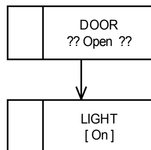
2. Behavior Trees

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, threads, and

constraints. Behavior trees are equally suited to capture behavior expressed in natural language functional requirements as to provide an abstract graphical representation of behavior expressed in a program. We may therefore ask can the same formal representation of behaviour be used for requirements and for a design? If it could it may clarify the requirements-design-architecture relationship.

Definition: *A Behavior Tree is a formal, composable, 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.*

Behavior trees provide a direct and clearly traceable relationship between what is expressed in the natural language representation and its formal specification. Translation is carried out on a sentence-by-sentence basis, e.g., the sentence "when the door is opened the light should go on" is translated to the behaviour tree below:



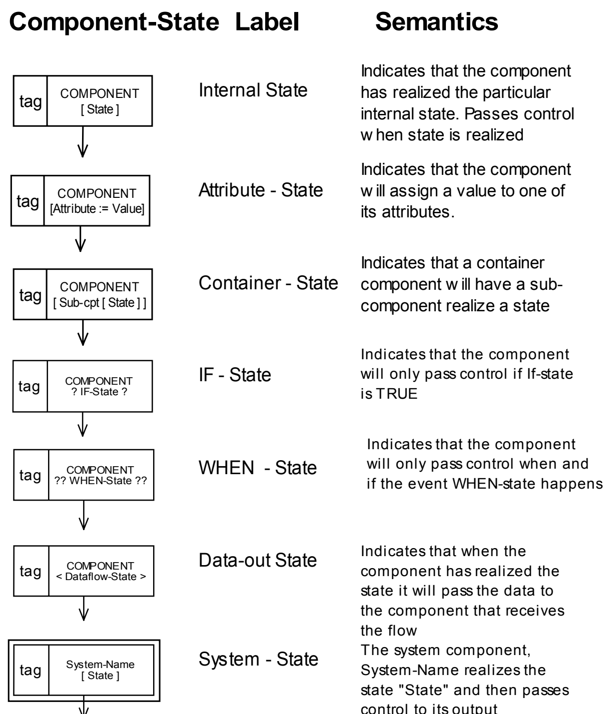


Figure 1. Behavior Tree Notation, key elements

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 relation, or [Attribute := expression | State]. 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. System states, are used to model high-level (abstract) behaviour and some preconditions/postconditions. Key elements of the notation are 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.

3. Genetic Software Engineering Method

Conventional software engineering applies the underlying design strategy of constructing a design that will *satisfy* its set of functional requirements. In contrast to this, a clear advantage of the behavior tree notation is that it allows us to construct a design *out of* its set of functional requirements, by integrating the behavior trees for individual functional requirements (RBTs), one-at-a-time, into an evolving design behavior tree (DBT). This very significantly reduces the complexity of the design process and any subsequent change process [5].

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

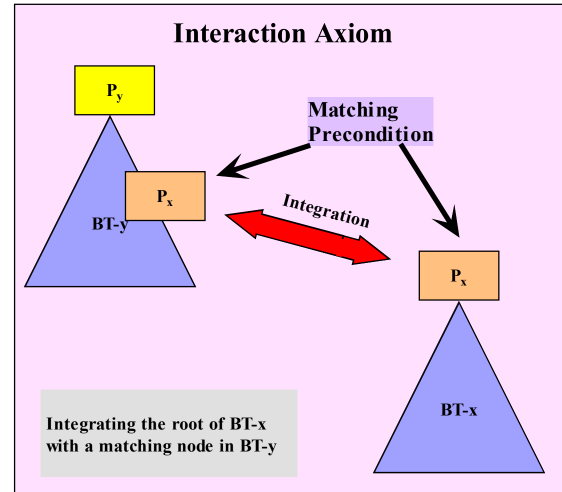


Figure 2. Interaction Axiom - graphic form

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 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 validation 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.
.

Comprehensive description, formalization, and justification of a software development method and notation, like the one here, requires significantly more than a conference paper length treatment To maximize communication we will only introduce the main ideas of the method informally and show how the architecture and component designs are obtained. The 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 Table I 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 is 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

After preliminary glossary/vocabulary processing and removal of aliases, etc, requirements translation is the first major 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 behavior trees. Translation identifies the *components* (including actors and users), the *states* they realise (including attribute assignments), the *events* and *decisions/constraints* that they are associated with, the *data* components exchange, and the *causal, logical* and *temporal* dependencies associated with component interactions.

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.

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 has good repeatability if translators forego the temptation to interpret, design, and introduce new things as they do an initial translation.

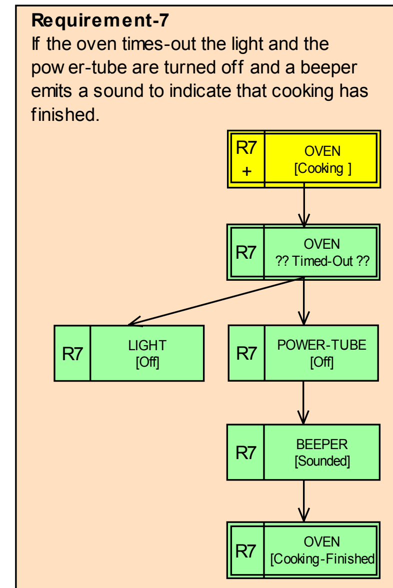


Figure 3. Behavior Tree for Requirement R7

3.2 Requirements Integration

When requirements translation is completed each individual functional requirement has been 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 only ever attempt to integrate two behavior 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/behavior missing from the set which prevents integration of a requirement. Attempts at integration uncover such problems with requirements at the earliest possible time.

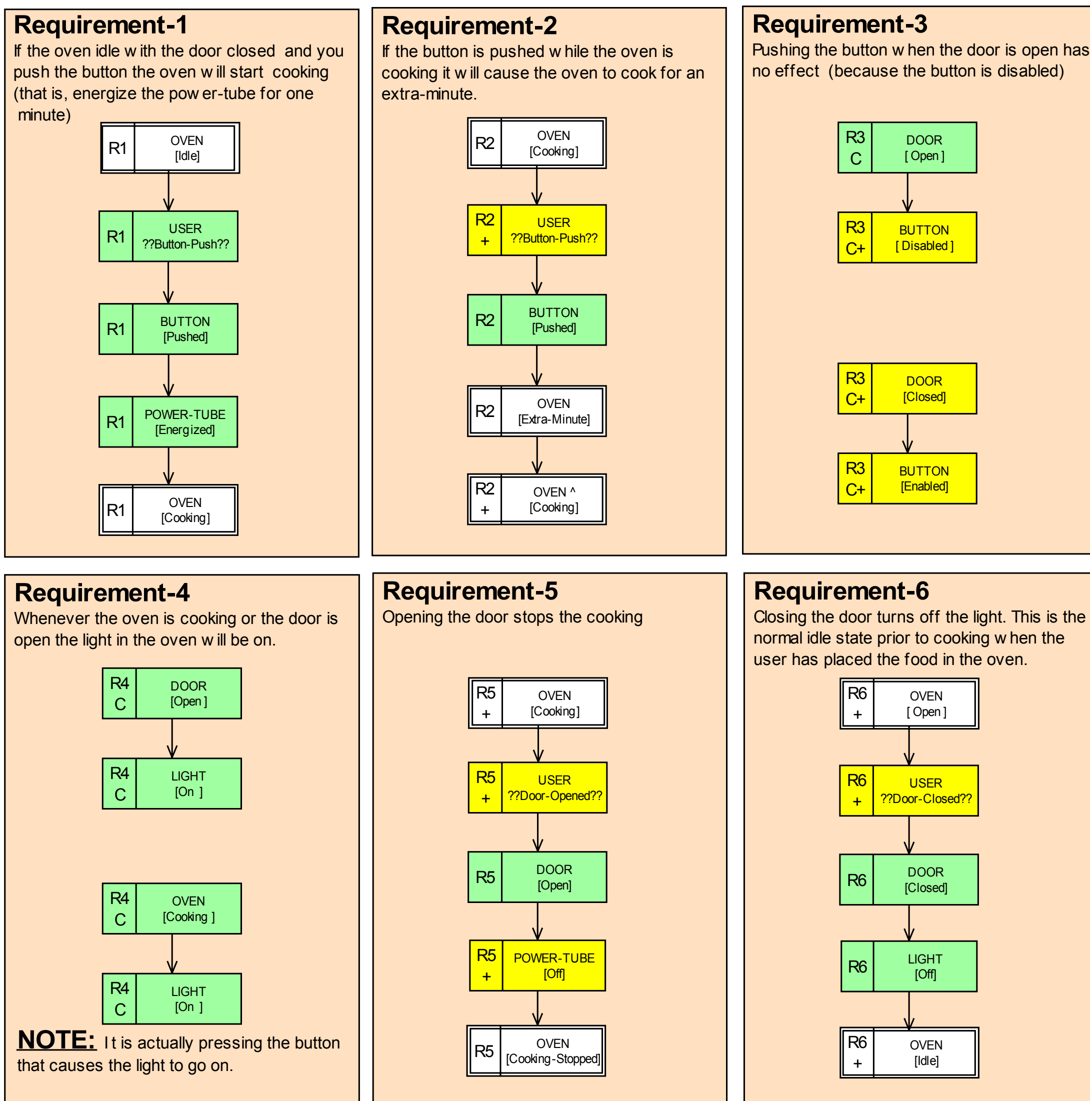


Figure 4. Behavior trees for Microwave Oven

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.

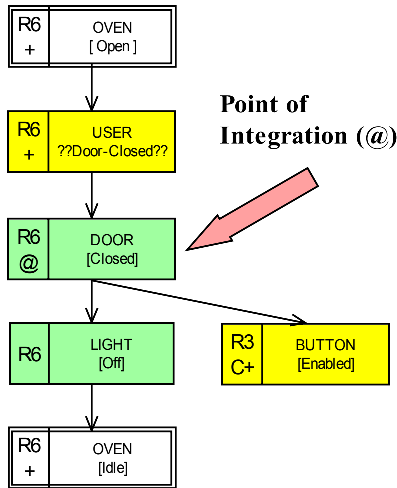


Figure 5. Result of Integrating R6 and R3C

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.

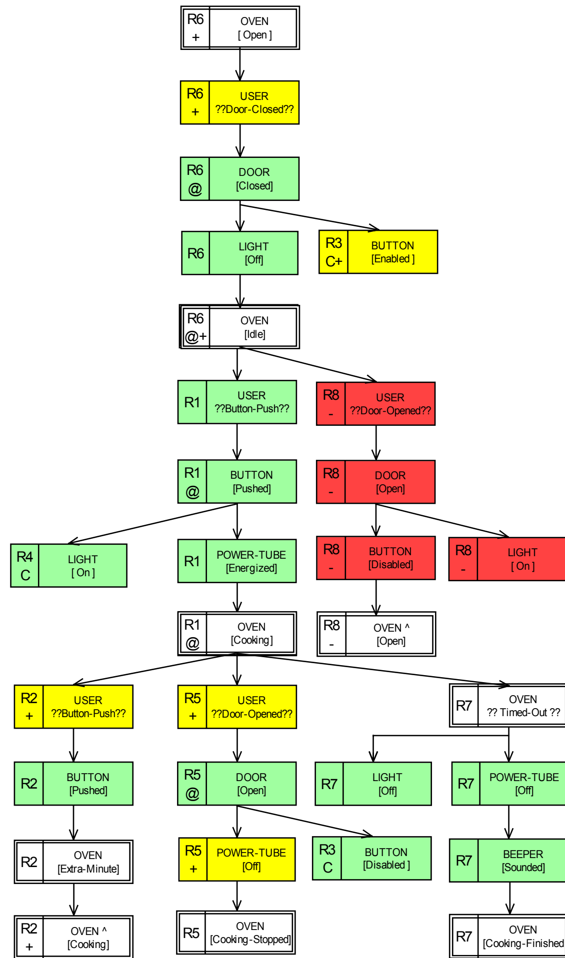


Figure 6. Integration of all functional requirements

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. 4 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

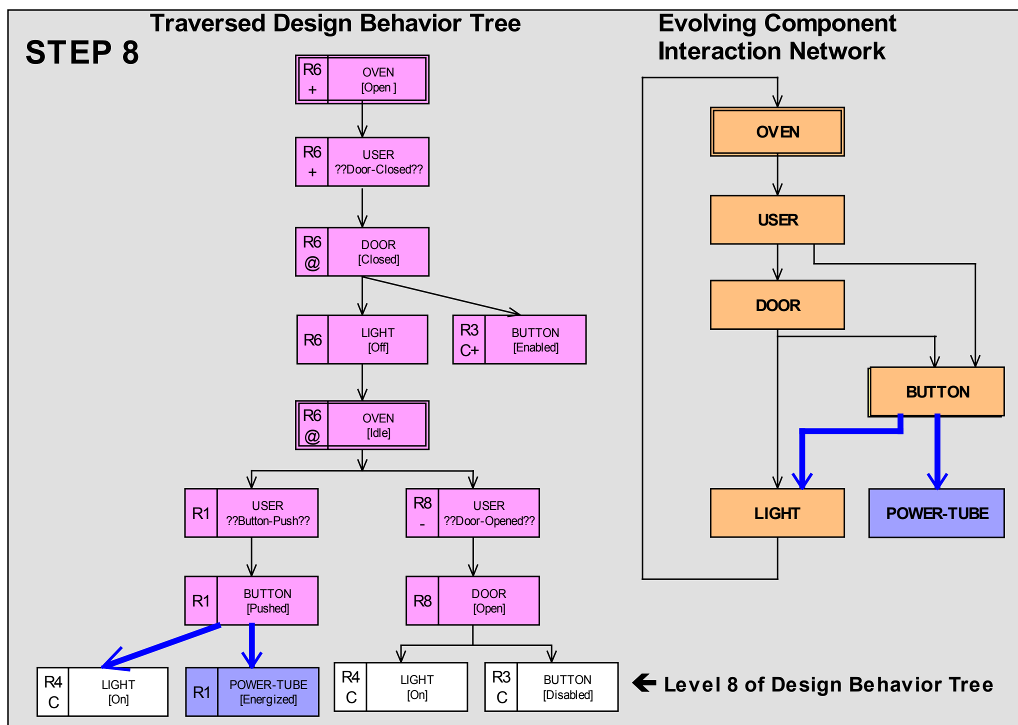


Figure 7. A step in the Tree-to-Network Transformation

parts of the tree in different states (e.g., 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 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. 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 network 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. The next important task is to isolate the behaviors of the individual components present in the architecture from the DBT using projection.

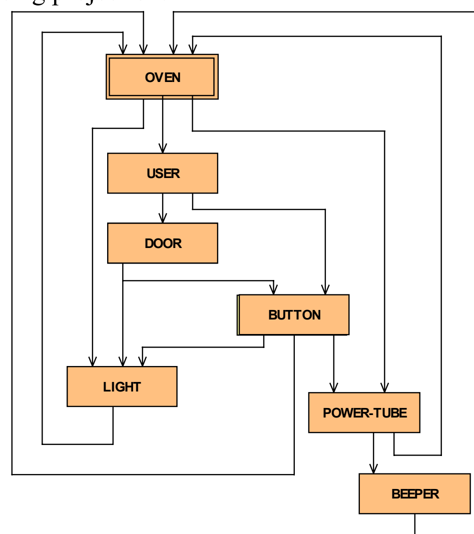


Figure 8. Component Interaction Network - (CIN)

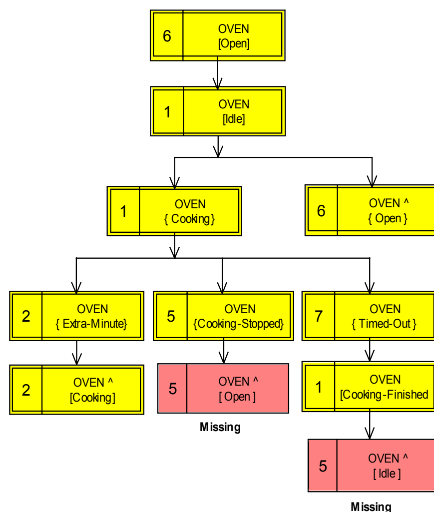
3.4 Component Behavior Projection

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

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.

OVEN COMPONENT - Projected Behavior



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.

In a number of reports and presentations at http://www.sqi.gu.edu.au/gse/papers 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.

Conclusion

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

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., 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

[4] S.Uchitel, J.Kramer, A Workbench for Synthesizing Behavior Models from Scenarios, 23rd International Conference on Software Engineering (ICSE'01), Toronto, Canada, 2001.

[5] R.G.Dromey, Genetic Software Engineering - Simplifying Design Using Requirements Integration, IEEE Working Conference on Complex and Dynamic Systems Architecture, S4, pp. 1-16, Brisbane, Dec 2001.

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

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