

Chapter IV

Making Real Progress with the Requirements Defects Problem

R. Geoff. Dromey
Griffith University, Australia

Abstract

Requirements defects remain a significant problem in the development of all software intensive systems including information systems. 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 (DBT). Different classes of defects are detected at each stage of the development process. Defects may be found at the translation to behavior trees, and then at the integration of behavior trees and when individual component behavior trees are projected from the DBT. Other defects may be found by inspection and model-checking of the DBT.

Introduction

There are seven well-known potential problems (Davis, 1988) with the functional requirements and their use in the development of modern software intensive systems:

- they may be incomplete, inconsistent, and/or contain redundancy
- they may not accurately convey the intent of the stakeholders
- in transitioning from the original requirements to the design, the original intention might not be accurately preserved
- over the course of the development of the system, the requirements may change
- the system the requirements imply may not be adequate to meet the needs of the intended application domain
- the number and complexity of the set of requirements may tax people's short-term memory beyond its limits
- the alignment between the requirements for a system, its design, and the implementation may be not preserved

Confronted with these challenges, existing methods for requirements analysis, inspection, representation, and then design are often not up to the task — we end with multiple partial views of a system that have a degree of overlap that makes it difficult to see/detect many types of defects, particularly those that involve interactions between requirements (see Booch et al., 1999; Harel, 1987; Schlaer & Mellor, 1992).

Given all this, is there a more practical way forward? Our chief concerns are:

- to get the complexity of the requirements under control,
- to preserve the intention of the stakeholders, and where there are ambiguities or other problems, clarify the original intention,
- to detect requirements defects as early as possible, and
- to ease the consequences of needing to change requirements as development proceeds and our understanding of the problem at hand improves.

We suggest there is a way to deliver these benefits and consistently make real progress with the requirements problem. It demands that we use the require-

ments of a system in a very different way. Traditionally the goal of systems development is to build a system that will satisfy the agreed requirements. We suggest this task is too hard, particularly if there is a large and complex set of requirements for a system. The Principle of Totality (Mayall, 1979) gives a clear insight into what the difficulty is. It tells us “*all design requirements are always interrelated and must always be treated as such throughout a design task*” (the degree of relationship can vary from negligible to critical between any pair of requirements). As soon as we have to deal with a large set of requirements, we have to take into account all their potential interactions. This almost always taxes the capacity of our short-term memory beyond its limits because we do not have a practical way of ensuring that all requirements, and all requirements interactions, are properly accounted for as we proceed with each design decision. The problem is further exacerbated by failure to detect defects that result from interactions between requirements.

A much simpler and easier task is to seek to build a system out of its requirements (Dromey, 2003). Building a design out of its requirements means that we only need to focus on the localized detail in one requirement at a time. This is cognitively a much more manageable task and less likely to cause our short-term memory to exceed its capacity (e.g., we only have to consider *one requirement at any time* rather than try to consider one hundred or a thousand requirements at a time which could easily be needed for a large system). It also means that requirements interactions are systematically accommodated because each requirement has a precondition associated with it which determines how and where it integrates into a design (c.f. how a given jigsaw puzzle piece slots into the solution to the overall puzzle — except that inserting requirements is easier than placing jigsaw puzzle pieces). If we opt to build a system out of its requirements, it implies:

- we have a representation that will actually represent the behavior in individual requirements;
- we have a way of combining individual requirements to create a system that will satisfy all requirements.

Existing notations like UML (Booch et al., 1999), state-charts (Harel, 1987) and others (Schlaer & Mellor, 1992) do not make it easy to combine individual requirements to create a system design. To explore the design strategy of building a system out of its requirements, it has been necessary to develop and extensively trial on a diverse set of both large and small systems a notation called Behavior Trees. 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 problem domain representation of design that will

satisfy those requirements. Individual requirements are first translated, one at a time, to behavior trees. Behavior trees are then integrated to create a design behavior tree (DBT). From the design behavior tree it is possible to derive the architecture of a system and the behavior designs of individual components. The processes of requirements translation, requirements integration, and the processes of model checking and inspecting the integrated design behavior tree provide a powerful means for finding many different classes of requirements defects.

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. The vital question that needs to be settled, if we are to build a system out of its requirements, is: can the same formal representation of behavior be used for requirements and for a design? Behavior trees make this possible, and as a consequence, clarify the requirements-design relationship.

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 basis, for example, the sentence “when the button is pressed, the bell sounds” is translated to the behavior tree below:

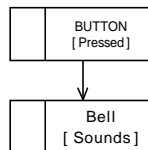
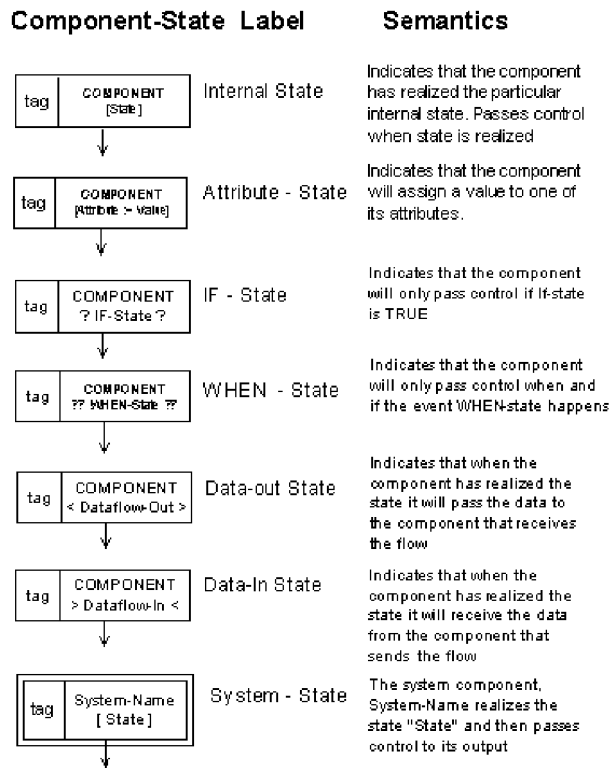


Figure ????



The principal conventions of the notation for component-states are the graphical forms for associating with a component a [State], ??Event??. ?Decision?, <Data_out>, or [Attribute := expression | State], and reversion “^”. Exactly what can be an event, a decision, a state, and so forth is 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) 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.

Genetic Design

Conventional software engineering applies the underlying design strategy of constructing a design that will *satisfy* its set of functional requirements. In contrast to this, a clear advantage of the behavior tree notation is that it allows us to construct a design *out of* its set of functional requirements, by integrating the behavior trees for individual functional requirements (RBTs), one-at-a-time, into an evolving design behavior tree (DBT). 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. We call this the *genetic design* process because of its links in similarity to what happens in genetics. Woolfson (2001), in the introduction of his book, provides a good discussion of this.

What we are suggesting is that a set of functional requirements, represented as behavior trees, in principal at least (when they form a complete and consistent set), contains enough information to allow their composition. This property is the exact same property that a set of pieces for a jigsaw puzzle and a set of genes possess (Woolfson, 2000). 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 design 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 behavior 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 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 (solution domain)
- Component behavior projection (solution domain)

Each overarching step needs to be augmented with a verification and refinement step designed specifically to isolate and correct the class of defects that show

Table 1. Functional requirements for Microwave Oven System

| |
|---|
| <p>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).</p> <p>R2. If the button is pushed while the oven is cooking, it will cause the oven to cook for an extra minute.</p> <p>R3. Pushing the button when the door is open has no effect (because it is disabled).</p> <p>R4. Whenever the oven is cooking or the door is open, the light in the oven will be on.</p> <p>R5. Opening the door stops the cooking.</p> <p>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.</p> <p>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.</p> |
|---|

up in the different work products generated by the process. A much more detailed account of this method is described in Dromey (2003).

To maximize communication our intent here is to only introduce the main ideas of the method relevant to requirements defect detection and do so in a relatively informal way. The whole process is best understood in the first instance by observing its application to a simple example. For our purposes, we will use requirements for a Microwave Oven System taken from the literature. The seven stated functional requirements for the Microwave Oven problem (Schlaer & Mellor, 1992, p. 36) are given in Table 1. Schlaer and Mellor have applied their state-based object-oriented analysis method to this set of functional requirements.

Requirements Translation

Requirements translation is the first formal step in the Genetic Design process and the first place where we have a chance to uncover requirements defects. 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 realize (including attribute assignments), the *events* and *decisions/constraints* they are associated with, the *data* components exchange, and the *causal*, *logical*, and *temporal* dependencies associated with component interactions.

The behavior trees resulting from translations for the first six functional requirements for the Microwave Oven System given in Table 1 are shown in Figure 1.

Figure 1. Behavior trees for Microwave Oven functional requirements

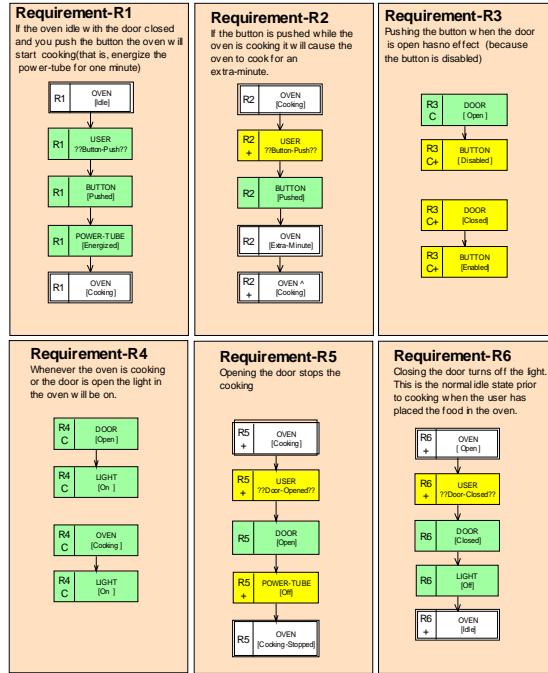
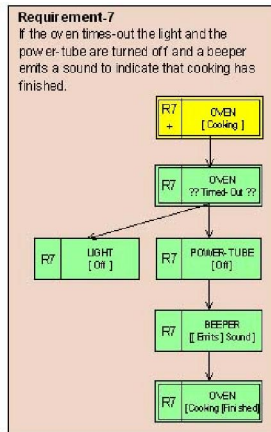


Figure 2. Behavior tree produced by translation of requirement R7 in Table 1



Example Translation

Translation of R7 from Table 1 will now be considered in slightly more detail. It involves identifying the **components** (including actors and users), the *states* (*italics*) they realize (including attribute assignments), and the order indicators (underlined), that is, the *events* and *decisions/constraints* they are associated with, the *data* components exchange, and the *causal, logical, and temporal* dependencies associated with component interactions. In making translations, we introduce no new terms, translate all terms, and leave no terms out. When these rules are followed, translation approaches repeatability. Functional requirement R7 is marked up using these conventions. “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 2 gives a translation of requirement R7, to a corresponding requirements behavior tree (RBT). In this translation, we have followed the convention of associating 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 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.

Translation Defect Detection

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

- Aliases, where different words are used to describe a particular entity or action, and so forth. For example, in one place the requirements might refer to the *Uplink-ID* while in another place, it is referred to as the *Uplink-Site-ID*. 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.

- 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.

A final point should be made about translation. It does not matter how good or how formal the representations are that we use for analysis/design, 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. And importantly, the behavior tree notation is simple enough for clients and users to understand without significant training. This is important for validation and detecting translation defects.

When requirements translation has been completed, each individual functional requirement is translated to one or more corresponding requirements behavior trees (RBTs). We can then systematically and incrementally construct a design behavior tree (DBT) that will satisfy all its requirements *by integrating the individual requirements' behavior trees* (RBT) one at a time. This step throws up another class of defects.

Requirements Integration

Integrating two behavior trees turns out to be a relatively simple process that is guided by the precondition and interaction axioms referred to previously. 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 generalizes 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 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 defects with the requirements at the earliest possible time. Many defects with requirements can only be discovered by creating an integrated view because examining requirements individually gives us no clue that there is a problem. In Figure 4, we show the result of integrating the seven RBTs that result from requirements translation (plus the missing requirement R8 — see subsequent discussion). It is easy to see because of the tags, R1, R2, and so on, where each functional requirement occurs in the integrated DBT. “@@” mark integration points.

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 3.

Figure 3. Result of integrating R6 and R3C

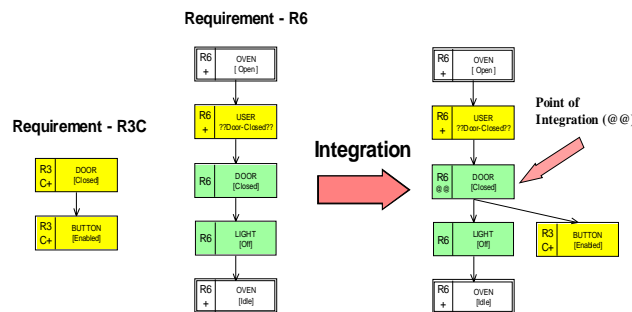
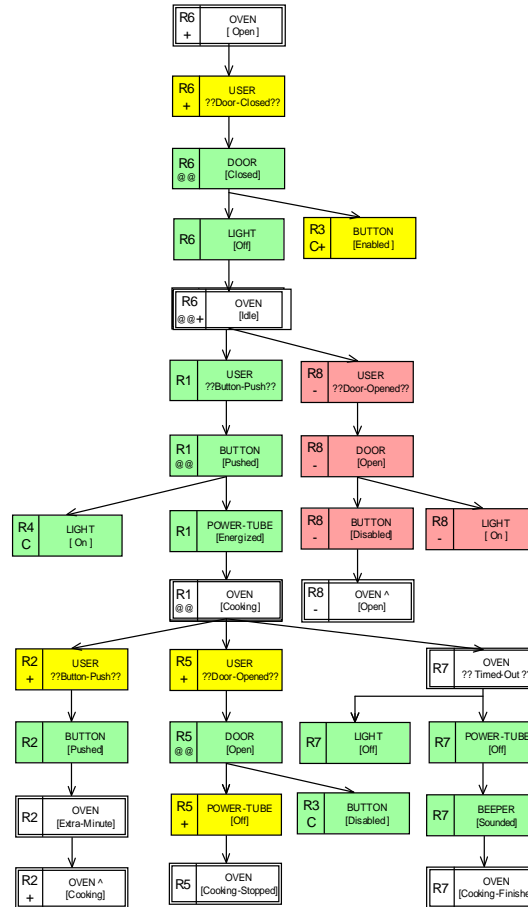


Figure 4. Integrated design behavior tree (DBT) for Microwave Oven System



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 and evolves incrementally by integrating RBTs and/or DBTs pairwise until we are left with a single final DBT (see Figure 4).

This is the ideal for design construction, 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 need 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.

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 precondition (it may be too weak or too strong or domain-incorrect) 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 RBT needs to be integrated is missing or incorrect.
- Both of the first two types of defects occur at the same time. Resolving this type of problem may sometimes require domain knowledge.
- 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, however, can 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 problems early on 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.

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

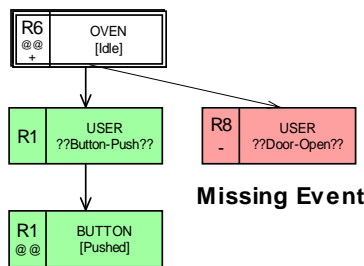
detection steps. The design behavior tree turns out to be a very effective representation for revealing a range of incompleteness and inconsistency defects that are common in original statements of requirements. The Microwave Oven System case study has its share of incompleteness and other defects.

The DBT can be subject to a manual visual formal inspection, and because behavior trees have a formal semantics (Winter, 2004) we can also use tools (Smith et al., 2004) 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 it 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

Figure 5. Missing event detected by the event completeness check rule



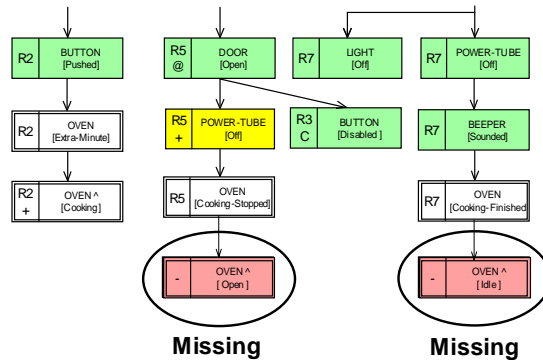
other events that we have listed occur at this point?” In the case where the OVEN[Idle] occurs, the only event in the original requirements is that the user-event of pushing the button to start the cooking can occur (see Figure 4).

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. It is interesting to note that when Schlaer and Mellor (1992) transition from the stated requirements to a state transition diagram, the missing behavior has been added without comment. Because of this sort of practice, traceability to the original requirements is lost in many design methods. Behavior trees which apply explicit translation and integration of requirements maintain traceability to the original requirements.

Missing Reversion Defects

Original statements of requirements frequently miss out on including details of reversions of behavior that are needed to make the behavior 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 behavior is possible – if such a situation arises, the integrated requirements have a reversion defect. Take the case of the Microwave Oven DBT in Figure 4. We find that if the Oven reaches either an OVEN[Cooking_Stopped] or an OVEN[Cooking_Finished] state, then no further behavior 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 6.

Figure 6. Reversion “^” nodes added to make DBT behavior 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 (Smith et al., 2004). From the XML we generate a CSP (Communicating Sequential Processes) representation. There are several translation strategies that we can use to map behavior trees into CSP. Details of one strategy for translating behavior trees into CSP are given in Winter (2004). A similar strategy involves defining subprocesses 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:

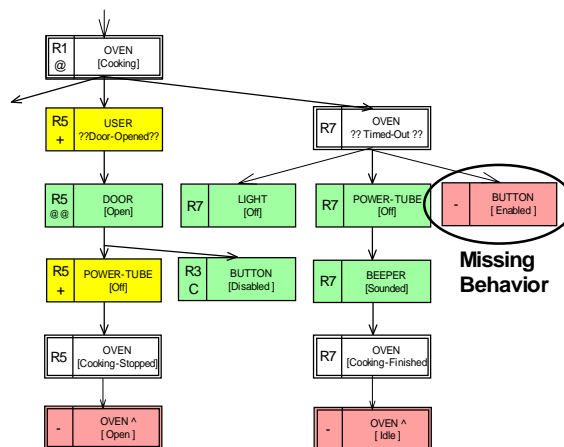
$$\text{DoorOpen} = \text{userDoorClosed} \rightarrow \text{doorClosed} \rightarrow \text{DoorClosed}$$

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 (Winter, 2004).

Reversion Inconsistency Defect Detection

The current tool does a number of consistency checks on a DBT. One important check to do is a reversion “^” check where control reverts back to an earlier

Figure 7. Missing behavior detected by checking $OVEN[Idle]/OVEN^{\wedge}[Idle]$ component state consistency



established state. For example, for the Microwave Oven example in Figure 4, 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 6 shows the bottom part of the Oven DBT from Figure 4.

We see that requirement R7 (and the DBT in Figure 4) is silent on any change to the state of the **BUTTON** component. This means we get from R1 that $BUTTON[Pushed]$ still holds when $OVEN^{\wedge}[Idle]$ is realized. 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:

$$OVEN[Idle] \equiv DOOR[Closed] \wedge LIGHT[Off] \wedge BUTTON[Enabled] \wedge \dots$$

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

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 to give a flavor 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 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. We will not pursue these design steps here or the associated error detection. They are dealt with elsewhere (Dromey, 2003), as is the necessary work on creating an integrated view of the data requirements and compositional requirements of a system which reveal still other defects.

Having provided a detailed description of how behavior trees, in combination with genetic design, may be systematically used to detect a wide range of defects

it is important to position this approach relative to other methods that are used to detect requirements defects. Here we will only do this at a high-level and in a qualitative way, because it is on the big issues where behavior trees differ from other options for requirements defect detection.

Comparison with Other Methods

Methods for detecting requirements defects can be divided into three broad classes: those that involve some form of human inspection of the original *informal* natural language statement of requirements, Fagan (1976), and more recently, a perspective-based approach (Shull & Basili, 2000); those that seek to create *graphic* representations of the requirements information (Booch et al., 1999; Harel, 1987); and those that seek to create a *formal* (usually symbolic) representation of the requirements information (Prowell et al., 1999) in an effort to detect defects.

There are three weaknesses with the informal inspection methods for requirements defect detection. They make no significant attempt to tackle the problem of complexity because they make no changes to the original information and therefore they come up against the memory overload problem. They also do nothing of significance to detect problems associated with the interaction between requirements because they do not create an integrated view. And thirdly, because they do not seek to map the original requirements to another representation they are less likely to uncover ambiguity and other language problems (aliases, etc.) with a large set of requirements. In contrast behavior trees tackle complexity by only needing to consider one defect at a time, they create an integrated view of requirements, and they employ a translation process that seeks to preserve original intention.

Graphic analysis and design methods, like UML (Booch et al., 1999), state-charts (Harel, 1987), and the method proposed by Schlaer and Mellor (1992), provide an analysis strategy that involves the creation of a number of partial views of the requirements information. The main difficulties with these approaches is that different analysts are likely, for a given problem and method, to produce widely different work products including the choices they make about which types of diagram to produce. The problem is compounded by overlap between some of the different types of diagram, and in some cases, these diagrams lack a rigorous semantics and are therefore open to more than one interpretation. There is also no direct or clear focus on how to use these methods and the various views/diagrams to detect defects. Because a variety of diagrams may be used, and the mapping does not involve rigorous translation, it is very difficult to guarantee

preservation of intention; this introduces yet another potential source for creating defects. We also commonly find, when these methods are used by practitioners, that things in the original requirements get left out, new things get added, and the vocabulary changes as the transition is made from textual requirements to a set of graphic views (Dromey, 2005). In contrast to these problems with graphic methods, the Behavior Tree Notation allows us to create a single integrated view of the behavioral requirements and a single integrated view of the static compositional and data requirements (not discussed in this chapter) of a system. Creation of behavior tree work-products approach repeatability of construction when produced by different analysts because they are based on rigorous translation which always has the goal of preservation of intention. As we have shown, behavior trees also employ a number of clearly defined strategies for detecting a number of different classes of defects. Behavior trees have also been given a formal semantics, which makes it possible to implement formal automated model-checking.

The use of formally based notations and methods has always been seen as a way of detecting defects with a set of requirements because of the consistency and completeness conventions they enforce. While there is no doubt in principle that this is true, there are three common problems with the use of formal notations. They require a transition from an informal statement of requirements to a formal representation. The problem here is whether this transition is done in a way that preserves the original intention. Most formal notations, for example, the Cleanroom method, do not employ a process that approaches the repeatability of requirements translation in making the transition from the informal to the formal representation. Another difficulty with formal notations is that very often they are not easily understood by clients and end users. This can make it much more difficult to guarantee intention has been preserved. In contrast, clients and end users have little difficulty in understanding behavior trees because the notation is relatively simple. One of the greatest difficulties with formal notations is that they do not usually easily scale up to dealing with larger and more complex systems. This is much less of a problem with behavior trees provided one has the necessary tool support.

Conclusion

Control of complexity is key to detecting and removing defects from large sets of functional requirements for all software-intensive systems, including information systems. Genetic design facilitates the control of complexity because it allows us to consider, translate, and integrate only one requirement at a time. At

each of the design steps, a different class of requirements defects is exposed. It is also much easier to see many types of defects in a single integrated view than spread across a number of partially overlapping views as is the case with methods based on representations like UML or state-charts. Because the design is built out of individual requirements, there is direct traceability to original natural language statements of requirements. Translation of individual requirements rather than some form of abstraction goes a long way to preserve and clarify intention as well as to find defects in original statements of requirements. This approach to design and defect detection has been successfully applied to a diverse range of real (large) industry applications from enterprise applications, to distributed information systems, to workflow systems, to finance systems, to defense-related command and control systems. In all cases the method has proved very effective at defect detection and the control of complexity. The utility of the method will increase as we develop more powerful tools that make it easy to control vocabulary, support multiple users, and perform a number of formal checks at each of the design stages.

References

- Booch, G., Rumbaugh, J., & Jacobson, I. (1999). *The unified modelling language user guide*. Reading, MA: Addison-Wesley.
- Davis, A. (1988). A comparison of techniques for the specification of external system behavior. *Communications of the ACM*, 31(9), 1098-1115.
- Dromey, R.G. (2003, September). From requirements to design: Formalizing the key steps (Invited Keynote Address). *IEEE International Conference on Software Engineering and Formal Methods*, Brisbane, Australia (pp. 2-11).
- Dromey, R.G. (2005). Genetic design: Amplifying our ability to deal with requirements complexity. In S. Leue & T.J. Systra (Eds.), *Scenarios, Lecture Notes in Computer Science*, 3466, 95-108.
- Fagan, M. (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3), 182-211.
- Harel, D. (1987). Statecharts: Visual formalism for complex systems. *Sci. Comp. Prog.*, 8, 231-274.
- Levenson, N. (2000, January). Intent specifications. *IEEE Transactions on Software Engineering*, SE-26(1).
- Mayall, W.H. (1979). *Principles of design*. London: The Design Council.

- Prowell, S., Trammell, C.J., Linger, R.C., & Poore, J.H. (1999). *Cleanroom software engineering: Technology and process*. Reading, MA: Addison-Wesley.
- Schlaer, S., & Mellor, S.J. (1992). *Object lifecycles*. NJ: Yourdon Press.
- Shull, I., & Basili, V. (2000, July). How perspective-based reading can improve requirements inspections. *IEEE Computer*, 33(7), 73-79.
- Smith, C., Winter, K., Hayes, I., Dromey, R.G., Lindsay, P., & Carrington, D. (2004, September). *An environment for building a system out of its requirements*. Proceedings of the 19th IEEE International Conference on Automated Software Engineering, Linz, Austria.
- Winter, K. (2004). Formalising behavior trees with CSP. *Proceedings of the International Conference on Integrated Formal Methods, LNCS, 2999*, 148-167.
- Woolfson, A. (2000). *Life without genes*. London: Flamingo.