From Requirements Change to Design Change: A Formal Path

Lian.Wen, R.Geoff. Dromey,
Software Quality Institute, Griffith University,
Nathan, Brisbane, Qld., 4111, AUSTRALIA.
+61 7 3875 5040
l.wen@griffith.edu.au, g.dromey@griffith.edu.au

Abstract

The ideal we seek when responding to a change in the functional requirements for a system is that we can quickly determine (1) *where* to make the change (2) *how* the change affects the architecture of the existing system (3) *which* components of the system are affected by the change (4) and, *what* behavioral changes will need to be made to the components (and their interfaces) that are affected by the change. The change problem is complicated because requirements changes are specified in the problem domain, whereas the design response and the implementation changes that need to be made are in the solution domain. Requirements and design representations vary significantly in the support they provide for accommodating requirements changes. An important way of cutting down the memory overload and difficulties associated with making changes is to use the same representation for requirements and the initial design response to the change. In this paper we use a formal component-state representation called *behavior trees* for this purpose. It allows individual functional requirements to be translated into their corresponding behavior trees; these trees are composed, one at a time, to create an integrated design behavior tree (DBT). The architecture, the component interfaces and the component behaviors of each component in the system are all emergent properties of the DBT. We extend this design approach, by proposing a formal method for mapping changes in a system's functional requirements, to changes in the architecture, the behavior of individual components and their interfaces. Such changes are shown visually on the work products of the design process that are affected. A tool is used to implement the change process.

Keywords

Software change, behavior trees, traceability analysis, software automation, software evolution, genetic software engineering, requirements engineering.

.

1. Introduction

The functional requirements of software systems being developed and software systems being used are typically subject to frequent change. Mapping these functional requirements changes (in problem domain) to the existing design (in the solution domain) and keeping all design documents consistent and up-to-date can be a difficult, tedious, and costly job. Traditional traceability analysis solutions apply hypertext systems [6, 9-11] and relational databases [7] to build an environment in which all the software documents are linked into a web. In this web, if one document is changed, the other documents that might be affected can be easily retrieved and browsed. However, such solutions usually do not provide facilities to automatically update the affected designs and related documents. It is a manual job to keep the whole set of documents consistent and up-to-date.

The previous discussion motivates the need to seek a way of automating the change process. Behavior Trees make this possible. The underlying strategy with behavior trees is to build a design out of its requirements. Each individual functional requirement is translated (manually) into its corresponding behavior tree(s). The resulting set of requirements behavior trees are then integrated one at a time to produce a design behavior tree (DBT). The design behavior tree captures all the functional requirements and shows their logical and behavioral relationships. The component-based architecture and the component designs of each of the components in the design are emergent properties of the DBT. The integrated design behavior tree, the architecture and the individual component designs form the baseline we need to manage subsequent changes to functional requirements. The general idea is to create the new design behavior tree based on the changed functional requirements. We then compare and merge the new design behavior tree with the original design behavior tree to create an edit behavior tree (EBT) that records all deleted, added, modified and unchanged functional requirements. Because the edited behavior tree is still built out of a set of functional requirements, and because the architecture, the component interfaces and the component designs are emergent properties of the EBT we can process it to generate the required changes to these work products.

Before discussing the formalization of the change process we will summarize the main ideas behind behavior trees

and their use in the genetic software engineering design method.

2. Genetic Software Engineering

2.1 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.

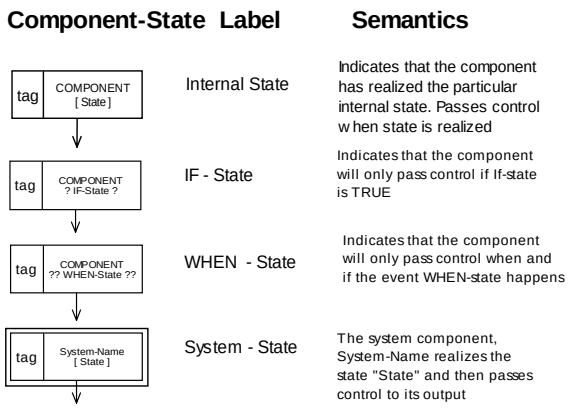
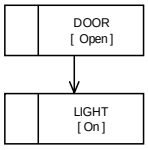


Figure 1. Behavior Tree Notation, key elements

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.*

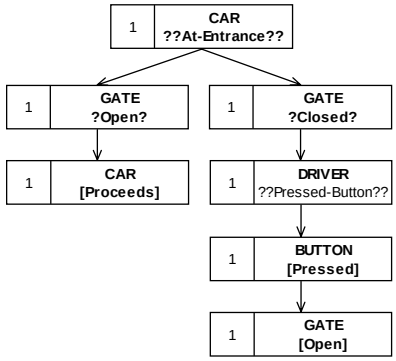
It provides a direct and clearly traceable relationship between what is expressed in the natural language representation and its formal specification. Translation is carried out on a sentence-by-sentence, word-by-word basis, e.g., the sentence "whenever the door is open the light turns on" is translated to the behavior tree below:



The principal conventions of the notation for component-states are the graphical forms for associating with a component a [State], ??Event??, ?Decision?. Exactly what can be an event, a decision, a state, are built on the formal foundations of expressions. 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. They are represented by rectangles with a double line (===) border. A selected list of key elements of the notation is given in Figure 1; for the whole set of the GSE notation please refer [1] or browse the web-site http://www.sqi.gu.edu.au/gse/papers.

2.2 Requirements Translation

Requirements translation is the first formal step in the Genetic Software Engineering (GSE) design process and it is the only step that cannot be fully automated. Its purpose is to translate each natural language represented functional requirement, one at a time, into one or more behavior trees. Translation involves identifying the **components (bold**)(including actors and users), the *states* (*italics*) they realize (including attribute assignments), and the order indicators (underlined) that is 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. In making translations we introduce no new terms, translate all terms and leave no terms out. When these rules are followed translation approaches repeatability. Consider the following functional requirement that is marked up using these conventions: "When *a* **car** *is at the entrance,* if *the* **gate** *is open* (seq) *the* **car** *proceeds* otherwise if *the* **gate** *is closed* when *the* **driver** *presses the* **button** (seq) *the* **gate** *becomes open"* The translated behavior tree is as follows:



To maximize communication our intent here is to only introduce the main ideas of the design 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 but complete example. Later, the same example will be modified to explain the proposed method that maps requirements changes to design changes. We use a design example for a Microwave Oven that has already been published in the literature [1, 12]. The seven stated functional requirements for the Microwave Oven problem are given in the table 1.

The translation for the requirement 7 (R7) in Table 1 is shown in Figure 2. From Figure 2, we can see that, initially, the OVEN is in the "Cooking" state. When the OVEN times-out, the LIGHT is off, POWER-TUBE is off, BEEPER sounds etc. In Figure 2, there is a "+" sign in

the root state "OVEN [Cooking]". This means this state is only implied in the original requirement.

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.

The behavior trees translated from the other six requirements can be found in [1]. In this paper, we only present the trees for requirement 3 and requirement 6 in Figure 3. Requirement 3 has two behavior trees because the statement for requirement 3 implies that when the DOOR is closed, the BUTTON is enabled.

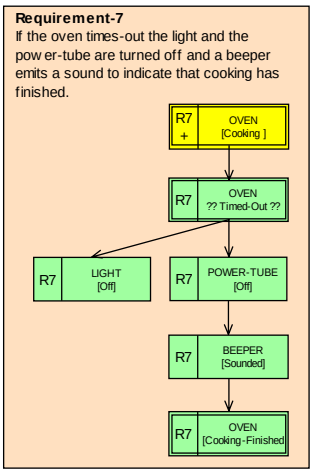


Figure 2. Behavior Tree for Requirement R7

2.3 Requirements Integration

When requirements translation has been completed each individual functional requirement is translated to one or more corresponding requirements behavior tree(s) (RBT). We can then systematically and incrementally construct a design behavior tree (DBT) that will satisfy all its requirements *by integrating the requirements' behavior trees* (RBT). The process of integrating two behavior trees is guided by the precondition and interaction axioms referred to below.

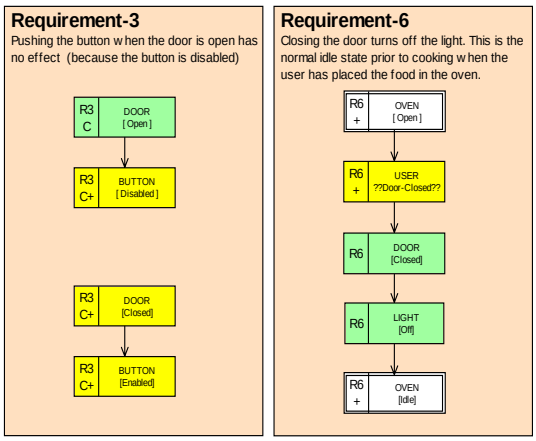


Figure 3. Behavior Trees for Requirement R3 and R6

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.

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).

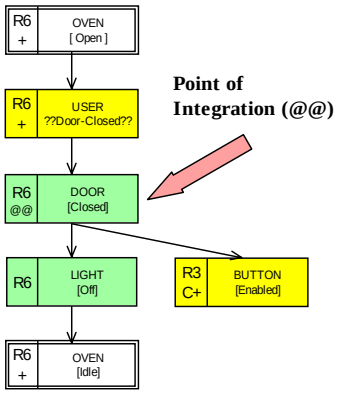


Figure 4. Result of Integrating R6 and R3C

Checking the behavior trees in Figure 3, it is found that the root node DOOR closed, exists in tree R6, so the RBT of R3 can be integrated with tree for R6 to create a new tree as shown in Figure 4.

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 5 below). This is the ideal for design construction that is realizable when all requirements are consistent, complete, composable and do not contain redundancies.

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

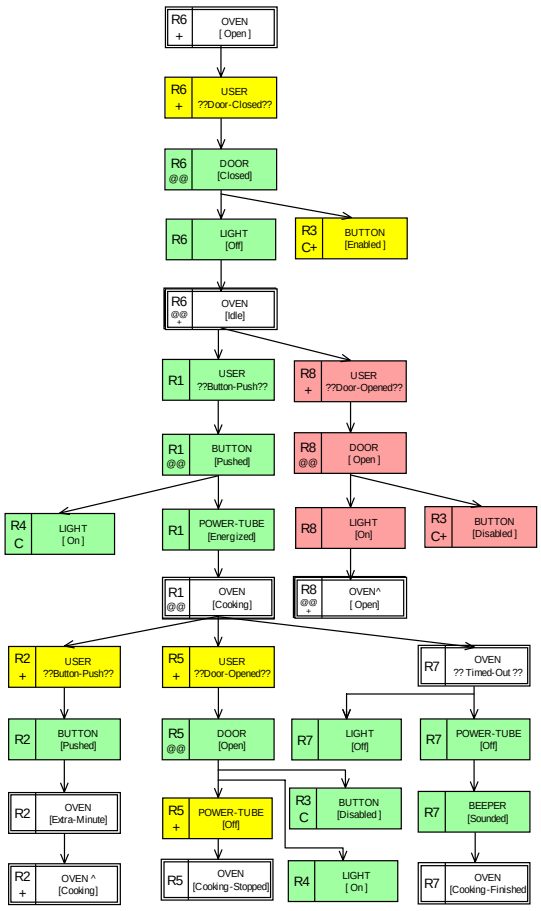


Figure 5. Integration of all functional requirements

2. 4 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), modeled 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 three key emergent properties of a design: (1) the component-architecture of a system, (2) the behaviors of each of the components, and (3) the interfaces of each of the components in the system* [1].

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 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 [1]. *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, which is simply a component dependency network for all the components in the requirements

The complete Component Interaction Network derived from the Microwave Oven design behavior tree is shown below in Figure 6. 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 interfaces may be systematically and significantly simplified but we not pursue that step here, e.g. light only needs one input).

2. 5 Component Behavior Projection

In the design behavior tree, the behavior of individual components tends to be dispersed throughout the tree (for example, see the 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.

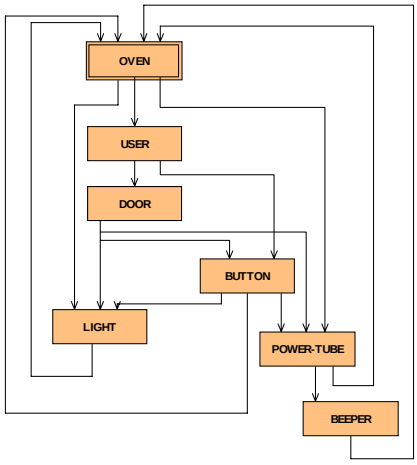


Figure 6. Component Interaction Network - (CIN)

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. 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.

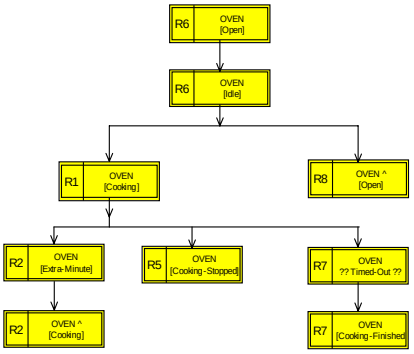


Figure 7. OVEN Component – Projected Behavior

2. 6 Component Interface Diagram

A component interface diagram (CID) shows the interface of a component and what other components link to the component and what other component it links to. A CID can be directly projected from the design behavior tree. The first step to project a component's CID is to highlight all the nodes in the DBT of the given component. We then have a list of all the links (state realizations, conditions and/or events) of the component. This yields the "input" components and "output" components of any component.

Figure 8 shows the CID of the OVEN component projected from the design behavior tree in Figure 5. A component interface diagram acts as a blueprint for the implementation of a component (these interfaces can be subject to a systematic simplification process).

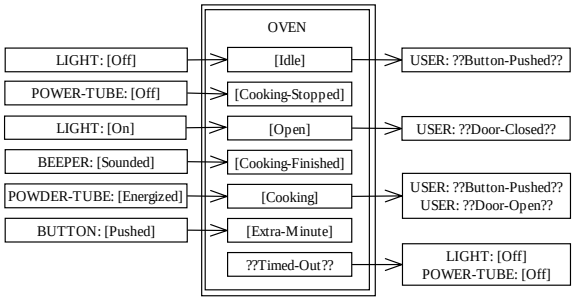


Figure 8. The CID of the component OVEN

3. From Requirements Change to Design Change

3. 1 Traceability in GSE

In traditional software engineering, most design documents are generated manually by the design team based on the designers' understanding and personal experience. In contrast with GSE, while the first step, translating individual functional requirements into RBTs, needs human *understanding*, the other steps have the potential to be either fully or at least partially automated. This potential for automation of key steps, together with the clear bi-directional traceability between the work-products of the design process (see Figure 9) provide important assistance for designing and implementing processes to support change of the functional requirements and the formal mapping of those changes onto the design.

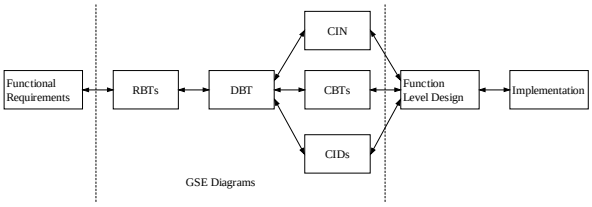


Figure 9 Traceability between work-products of GSE.

GSE's strong traceability works as a bridge to connect functional requirements, to the design.

3.2 Mapping Requirements Changes to Design Changes

Consider a software system that has been designed based on a set of functional requirements. Once the requirements are changed, the question is how to change the design to

match the new requirements. Existing design methods, including the original GSE method [1], do not provide a clear process, and supporting representations, for adjusting the design to accommodate the change in the functional requirements.

The present proposal addresses this problem of formalizing the impact of change on the design. The output of the method is a set of edited design diagrams which show the impact of the changed requirements on the design. More specifically, the edited design diagrams not only show the new design, but also mark which parts are new in the design, which parts existed in the old design but have been removed and which parts are unchanged. Currently, the method is only suitable for projects originally designed by the GSE method, because GSE provides a systemic process to translate and integrate functional requirements into the design. However a similar concept may be applicable to projects designed using other methods.

To understand the formalization of change, suppose we have a design originally constructed using GSE. To map subsequent changes to the functional requirements onto the existing design (captured by the DBT), we use the following major steps:

1. From the changed requirements, we translate any new/additional requirements to behavior trees.
2. We then use requirements integration and editing of the old DBT to produce a new DBT that accurately reflects the changed requirements.
3. *The new DBT and the old DBT are then merged to produce an Edit Behavior Tree (EBT).*
4. The other diagrams are then derived from the EBT using modified GSE processes (section 2).

The overall process is similar to the original GSE process, but it introduces a very important step: that of comparing the old DBT and the new DBT and merging them into an EBT (the detail of the merging algorithm is described in the next sub-section). The key point is that the EBT contains all the behaviors of the original DBT and new DBT and it also contains the editing information, which marks the change impact of the changes in the functional requirements.

The last step is to derive from the EBT the other edited design diagrams: the ECIN (edited component integration network, which shows the change impact on the architecture), the ECBTs (edited component behavior trees) and ECIDs (edited component interface diagrams). The method of projection is similar to that used in GSE except it also maintains the edit information. Details of the projection rules are discussed in the following sections.

3.3 Algorithm to Compare and Merge Behavior Trees

The purpose of comparing the new DBT and the old DBT is to identify the changes; to find out the new behaviors that are introduced into the new tree, the behaviors in the old tree but not in the new tree and the behaviors unchanged in the two trees. This information is stored in the EBT. As an example, suppose that T_1 and T_2, shown in Figure 10, are the old DBT and the new DBT respectively.

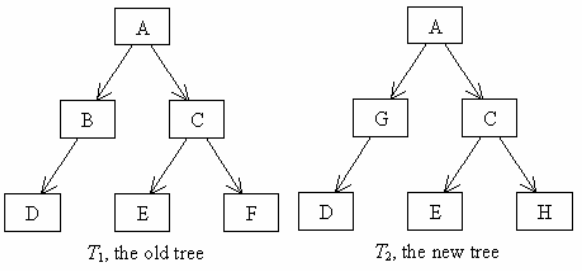


Figure 10 The old tree T1 and the new tree T2

To compare T_1 and T_2 and generate the Edit Behavior tree, we use the following algorithm:

1. Start the comparison[1] with the root nodes (in this example, node A). Because the root node exists in both trees, it is created in the edit behavior tree as an unchanged node.
2. Find the compared node's child-node set in both trees. (In this example, the child-node set in the old tree is {B, C} and the child-node set in the new tree is {G, C}.
3. If a node exists in the old tree's child node set but not in the new tree's child node set, this node will be marked in the edit behavior tree as an old node. (In this example, B is such a node)
4. In the old tree, the subtrees under the old node will be generated in the EBT as old. (In this example, the node D under node B in T_1 is such a case)
5. If a node exists in the new tree's child-node set but not in the old tree's child node set, this node will be created in the EBT as a new node. (In the example, G is such a node)
6. In the new tree, the subtrees under the new node will be generated in the EBT as new. (In this example, the node D under node G in T_2 is such a case)
7. If a node exists in the child node sets of both trees, it will be generated in the EBT as an unchanged node. (In the example, the node C is such a case)
8. An unchanged node will be a new comparison node and the algorithm will go back recursively to step 2.

[1] In this algorithm, we assume the two trees have an identical root node. If the two trees have different root nodes, one possible solution is to add an artificial root in both trees or adopt more sophisticated algorithms.

The edit behavior tree T_e produced from T_1 and T_2 is shown in Figure 11. The new part in the tree is drawn with bold lines and the old part in the tree is drawn with dotted lines and the unchanged part is drawn in the normal style.

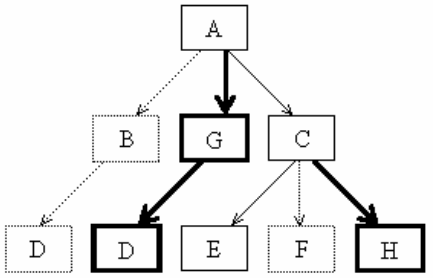


Figure 11 The edit behavior tree Te

One interesting thing in Figure 11 is node D. It is both old and new, which means it should be an unchanged node. However, the algorithm cannot resolve this fact at this stage. In the next stage, when projecting other diagrams from the EBT, the true status of node D will be determined.

3.3 The Projection and Transformation Rules

The rules to project the edited design diagrams from an EBT are similar to the rules to project design diagrams from a DBT that have been introduced in sections 2.4-2.6. The only difference is that the rules used for an EBT have to carry through the editing information.

As we have discussed before, during the process of projecting diagrams from a DBT, the DBT is decomposed into many atomic elements, while each element is either a node (a state, a condition or an event) or a link, and each element maps to a corresponding part in the target diagram. When a design diagram (a CIN, a CBT or a CID) is projected (or in the case of a CIN, obtained by transformation) from a DBT, any atomic part in the design diagram can be traced back to a link (or several links) or a node (or several nodes) in the DBT. If the projection/transformation source is not in the original DBT but in the EBT, each atomic part in the design diagram will inherit the editing information from its counterparts in the EBT.

For example, with the EBT in Figure 11, because node *H* is marked as "new", in a design diagram, if a particular part is projected or transformed from node *H*, that part will also be marked as "new". The same rule applies to entities of "old" and "unchanged" status. Note "old" nodes are marked for deletion.

In addition to the straightforward mapping rule, there is one exception. The transformation from an EBT to the CIN or a CID can be a many-to-one projection. This means several nodes (or links) in the EBT may project/transform to one single part in the design diagram, just as a particular state of a component have more than one node in an EBT, but when the EBT is transformed to a CIN, these nodes will merge to a single state within a component projected behavior tree. Therefore, a single atomic part in an edited design diagram may have more than a single edit source in the EBT.

The rules to merge this different editing information turn out to be straightforward. Referring to Figure 11 again, there are two node D's, one is marked as "new", which means the node D exists in the new requirement and another is marked as "old", which means node D also exists in the old requirement. Because node D exists in both the original requirements and the modified requirements it must be treated as unchanged in the edited diagram. From this simple analysis, we know that whenever an entity of "old" status merges with one of "new" status, it becomes "unchanged". Similarly, when "old" merges with "unchanged" it will be treated as "unchanged". For the case of "new" merging with "unchanged" it is also resolved as "unchanged". We may therefore summarize all the projection/transformation rules for dealing with editing information as follows:

1. "New" to "new".
2. "Old" to "old".
3. "Unchanged" to "unchanged".
4. "New" merged with "new" equals "new".
5. "Old" merged with "old" equals "old".
6. "New" merged with "old" equals "unchanged".
7. "New" or "old" or "unchanged" merged with "unchanged" equals "unchanged".

4. An Example

In section 2, we used a simple example to explain the basic concepts of GSE. If the functional requirements are now changed, the following example will show how, when using the method in section 3, the change impact is captured and reflected in the traceability analysis model.

Suppose a new component TIMER is introduced. This may cause the original requirements 1, 2 and 6 to be changed as described below (the modifications to the three requirements are underlined).

Modified requirement 1: There is a single control button available for the user of the oven. If the oven is idle state and you push the button, the timer will be set to one minute and the oven will cook (that is, energize the power-tube)

Modified requirement 2: If the button is pushed while the oven is cooking it will cause the timer to add one extra minute

Modified requirement 7: If the timer times-out, the light and power-tube are turned off and then a beeper emits a sound to indicate that the cooking is finished.

Figure 12 shows the new requirements behavior tree of the modified requirement 7 and the edited behavior tree (EBT) is shown in Figure 13. It was constructed using a tool that employs the rules described in section 3.

In Figure 13, the new fragments of behavior are drawn in bold lines and filled with dark gray, the old fragments of behavior, which are not in the modified system, are drawn in light grey lines and the unchanged parts are drawn in the normal style. This diagram shows clearly the change impact of the modified requirements on the behavior tree.

From the EBT, other diagrams (the ECIN in Figure 14, the ECID of OVEN in Figure 15 and the ECBT of OVEN in Figure 16) are projected. Because of space limitations, only the editing component diagrams of component OVEN are shown.

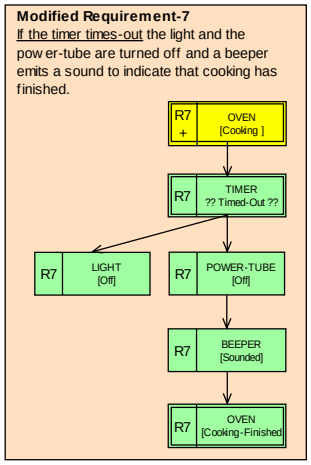


Figure 12. The RBT for Modified Requirement R7

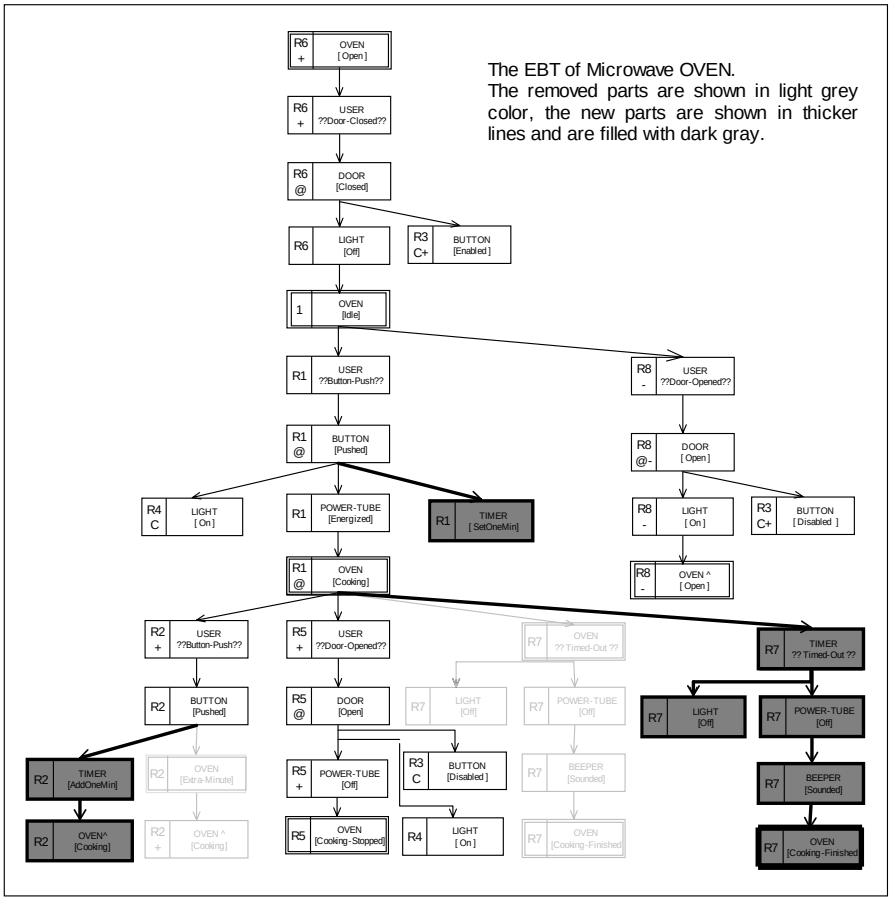


Figure 13. The edited behavior tree of the Microwave Oven

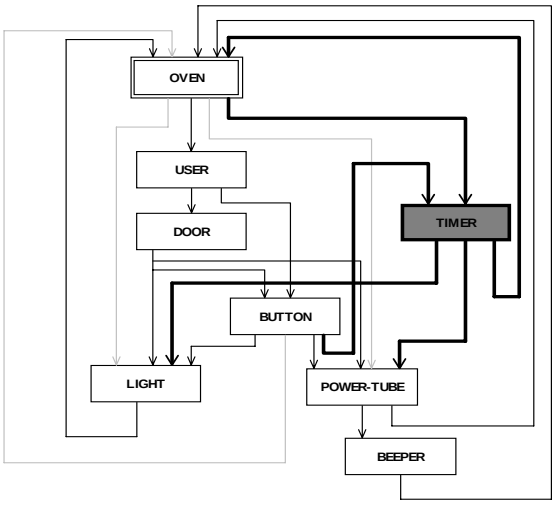


Figure 14. The ECIN of the new Microwave OVEN

From the ECIN (Figure 14), the change impact on the software architecture is clearly marked. Figure 14 shows that several interaction relationships between the component OVEN and other components are removed and a new component TIMER is added as well as several component interaction relationships with TIMER.

Figure 15 is the ECID (Edited Component Interface Diagram) of the component OVEN. In this diagram, the new text is bolded and filled with dark gray and the old part is drawn in light gray. It shows that the interface ??TimeOut?? and [Extra-Minute] are removed from OVEN component and the new component TIMER, which is called from the [Cooking] interface is added.

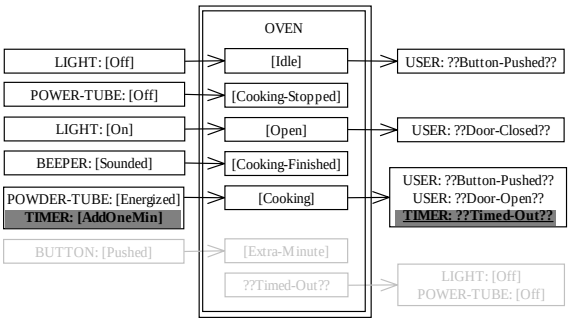


Figure 15. The ECID of the OVEN component

Figure 16 is the ECBT (Edited Component Behavior Tree) of the component OVEN. This figure shows the change impact on its internal behavior.

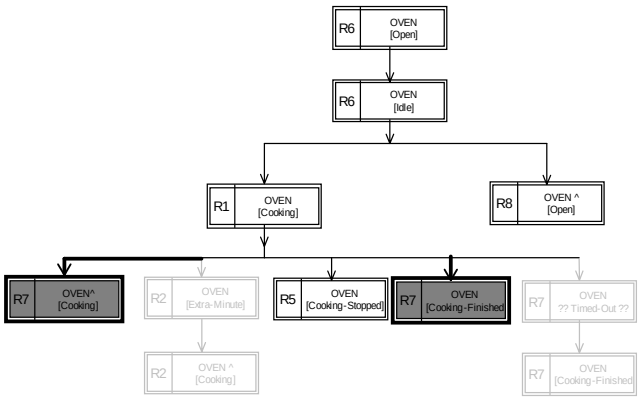


Figure 16 The ECBT of component OVEN

This example demonstrates how the traceability analysis model can be used to identify the change impact on different artifacts in the system, not only at the architecture level, but also at the component internal structure and interface level as well. This information can be used to adjust the implementation to make the system match the new/modified functional requirements.

5. Comparison

Other research on software change differs from the method proposed in this paper. The goal here has been to find a systematic process to map the changes in functional requirements to the changes in the design and the implementation. The closest approach we have found to our work is software change impact analysis [2, 3 and 4], which aims to estimate what will be affected in software and related documentation if a proposed software change is made. In software change impact analysis, one approach, called dependency analysis [2], mainly targets low level software artifacts such as source code. Another approach, called traceability analysis [6, 7, 8, 9 and 10], tries to establish traceability among high level software documents, closer to our proposal. Overall, current research in this field mainly focuses on developing software environments that can manage change to different types of software documents. These environments provide different facilities such as those for defining document structures, document templates, relationships among documents, document revision control, and key word matching etc. These approaches allow users to trace changes and identify change impact on different documents more efficiently. However most of these environments are not built upon repeatable or well-defined methods, which implement logic-based rules that can link formally different types of software documents, As a result, users have to manually change each impacted documents.

In our approach, except for the first step of translating functional requirements into behavior trees, all the other steps are based on well-defined rules and processes. This means they can be implemented by automated or at least semi-automated tools. A further advantage of this automated support is that functional requirements can be integrated into the edit behavior tree one by one. As these changes are made the corresponding design diagrams can be automatically re-generated on the fly to reflect each change as it is made. Therefore, the impact of each individual requirement on the design can be traced. This unique feature gives the method a powerful and systematic means for controlling the impact of change on a design.

6. Conclusion

The representations we have presented here show considerable promise as the basis for a fundamental theory that could underpin the creation of powerful software design and software maintenance tools. The prototype tool we have developed confirms the feasibility of this approach. It was used to generate the edited results used in this paper.

There has always been a wide gap between a set of functional requirements and a software design. GSE [1] provides a bridge to link requirements to a corresponding design that will satisfy those requirements. The original GSE method did not answer the question "if one side of the bridge changes, how should the other side change to make the two parts correspond?". The method introduced in this paper directly addresses this question. A clear advantage of using a representation that allows us to build a system out of its functional requirements is that the accompanying change process is relatively easy to formalize and therefore support with automated tools. This representation also helps us answer the question, as to where to make the change, and what impact does the change have on the architecture, the component designs and the component interfaces.

The proposed model, as presented, is only suitable for software projects that use behavior trees and the GSE design methodology. The concepts employed in this method might however also be adapted for other software design methods, such as the traditional OO design approach based on UML [5]. Actually, some diagrams in UML have some similarities with the diagrams in GSE. For example the activity diagram with the RBT, the class diagram with the CIN and the state diagram with the CBT etc. However, the lack of strict dependency relationships among different types of diagrams limits the possibility of automatically updateing other design diagrams if one diagram is changed. In GSE, the fundamental diagram is the DBT, which describes all the behaviors of the targeted system and includes all the information for any other diagrams. If we could introduce the DBT into UML, it would not be difficult to invent corresponding methods to automatically update many different types of design diagrams.

References

[1] Dromey, R.G., "*From Requirements Change to Design Change: Formalising the Key Steps"*, (Invited Keynote Address), IEEE International Conference on Software Engineering and Formal Methods, SEFM'2003, pp. 2-11, Brisbane, September, 2003.

[2] Bohner, S. A., Arnold, R. S., "*Software Change Impact Analysis*", IEEE Computer society Press Los Alamitos, California, 1996 QA 76 .76 .E93 B65

[3] Bohner , S. A., "*Software Change Impact Analysis for Design Evolution*", Proc. 8th Int'l conf. on software Maintenance and Re-engineering, IEEE CS Press, Los Alamitos, Calif., 1991

[4] Bohner, S. A., "*Impact Analysis in the Software Change Process: A Year 2000 Perspective*", Software Change Impact Analysis, 1996

[5] Fowler, M., Scott, K., , "*UML Distilled A Brief Guide to the Standard Object Modeling Language*", Addison-Wesley Publishers Ltd., 2000

[6] Garg, P. K., Scacchi, W., "*A Hypertext System to Manage Software Life-Cycle Documents*", IEEE Software, Vol. 7, No. 3, May 1990, pp. 90-98.

[7] Horowitz, E., Williamson, R. C., "*SODOS: A Software Documentation Support Environment - Its Definition*", IEEE Trans. Software Eng., Vol. SE-12, No. 8, Aug. 1986

[8] Cimitile, A., Lanuble, F., Visaggio, G., "*Traceability Based on Design Decisions*", Proc. Conf. On Software Maintenance, IEEE CS Press, Los Alamitos, Clif. 1992

[9] Conklin, J., "*Hypertext: An Introduction and Survey*", Computer, Sept. 1987,

[10] Trigg, R. H., and Weiser, M. "*Textnet: A Network-Based Approach to Text Handling*", ACM Trans. Office Information Systems, Jan. 1986

[11] Bigelow, J., "*Hypertext and CASE*", IEEE Software, March 1988

[12] Shlaer, S., Mellor, S.J., "*Structured Development for Real-Time Systems*", Vols. 1-3, Yourdon Press, 1985.