

# Architecture Normalization for Component-based Systems

Lian Wen<sup>1</sup>

*Software Quality Institute  
Griffith University  
Nathan, Brisbane, Qld., 4111, AUSTRALIA*

Geoff R. Dromey<sup>2</sup>

*Software Quality Institute  
Griffith University  
Nathan, Brisbane, Qld., 4111, AUSTRALIA*

---

## Abstract

Being able to systematically change the original architecture of a component-based system to a desired target architecture without changing the set of functional requirements of the system is a useful capability. It opens up the possibility of making the architecture of any system conform to a particular form or shape of our choosing. The Behavior Tree notation makes it possible to realize this capability by inserting action-inert bridge component-state. For example, we can convert typical network component architectures into normalized tree-like architectures which have significant advantages. We can also use this “architecture change” capability to keep the architecture of a system stable when changes are made in the functional requirements. The results in this paper build on earlier work for formalizing the process of building a system out of its requirements and formalizing the impact of requirements change on the design of a system.

*Keywords:* Components, software architecture, formal methods, behavior trees, genetic software engineering.

---

## 1 Introduction

Software architecture is one of the critical issues in software engineering. In this paper, we will use the concept of component interaction network (CIN) [1,2] as our chosen architectural construct. A CIN is a graph that shows a software system’s components and the dependencies or interactions among them.

Generally, a lower coupled system is more portable and easier to maintain. In this paper, we propose a tree-like hierarchical structure as an optimized component

---

<sup>1</sup> Email: [l.wen@gu.edu.au](mailto:l.wen@gu.edu.au)

<sup>2</sup> Email: [g.dromey@griffith.edu.au](mailto:g.dromey@griffith.edu.au)

architecture because of the scalability and simplicity of trees. A tree is a connected graph with the least amount of coupling. Many architectural styles such as “Pipe & Filter”, “Shared Repository”, “Layered Abstract Machine”, “Bus”, “Client-Server” [6,7], and “C2” [12] can be abstracted as trees in special conditions. We call a software system with a tree-structured CIN a normalized system; the procedure for transforming a non-normalized system into a normalized system is called architecture normalization.

It is usually argued that software architecture is determined or at least strongly influenced by the functional requirements of the system. A complex system may inevitably produce a complex architecture. However, our research shows that the topological structure of a CIN can be made independent of the functional requirements that the system satisfies.

To prove this point, we use the Genetic Software Engineering (GSE) design process [1]. GSE provides a formal approach for designing component-based software systems. The underlying procedure of GSE includes three steps. Firstly, each individual functional requirement is translated (manually) into a corresponding tree-structured graph called a requirement behavior tree (RBT); then these trees are integrated into one large tree called a design behavior tree (DBT); finally from the DBT, other design diagrams includes the component architecture (CIN) are retrieved. In GSE, because the procedure for the last two steps is clearly defined, once the set of RBTs are fixed, the corresponding CIN is also fixed. Therefore, the focus of this problem is how we can have different sets of RBTs for the same set of functional requirements. To achieve this, the first method is to adjust the order of nodes in RBTs if the order has not be specified by the functional requirements; the second method is to insert bridge component-states, which are similar to hidden events in CSP [8]. The second method is more systematic that can transfer the CIN into any pre-defined form without affecting the functional requirements. In other words, the component architecture can be independent to the functional requirements.

Based on our previous work, GSE not only provides a systematic approach to construct component-based software design, it also provides a formal method to do change impact analysis [2]. When a software system has been adjusted due to the changes in the functional requirements, a traceability model has been proposed to show the change impacts on the component architecture as well as on other design documents. Sometimes, changes in a system’s functional requirements will affect the architecture. Repeated changes of a system may eventually ruin the system’s architecture. However, based on the result of this paper, it is possible for the designers to preserve the architecture or minimize the change impact when the functional requirements have been changed. If the component architecture of a large system can be kept stable during the system’s lifetime, it will undoubtedly reduce the maintenance costs of that system.

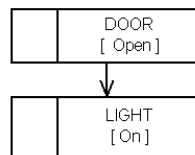
The paper is organized as following: Section 2 briefly introduces the concept of GSE. Section 3 introduces the architecture transformation theory. In Section 4, we propose the concept of software normalization, and a microwave oven case study

has been presented to illustrate the architecture transformation theory and the simplicity of a normalized system. Finally, the last section gives a brief conclusion.

## 2 Genetic Software Engineering

### 2.1 Behavior Trees

The Behavior Tree notation, which has been given a formal semantics [1], captures in a simple tree-like form of composed component-states. It provides a direct and clearly traceable relationship between what is expressed in the natural language representation and its formal specification. For example, 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], an ??Event?? or a ?Decision?. Exactly what can be an event, a decision, a state, are built on the formal foundations of expressions<sup>3</sup>. 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. For details of the latest GSE notation please browse the SQI paper site [3].

### 2.2 GSE Design Process

There are three major steps to construct a component-based architecture using the GSE design process. The first step is to translate each individual functional requirement into one or more corresponding requirements behavior trees (RBTs). The second step is to integrate all the RBTs into a single design behavior tree (DBT) and the third step is to project the component interaction network (CIN) and many other design documents. Further details of the GSE procedures are given elsewhere [1,3]. To maximize communication our intent is to introduce the main ideas of the design method in a relatively informal way. The whole design process is best understood in the first instance by observing its application to a simple example. Later, the same example will be normalized to explain how the proposed method manipulates the DBT so that the corresponding component architecture can be transformed to a tree structure. We use a design example for a Microwave Oven which has already been published in the literature [1,2] and [4]. The seven

<sup>3</sup> For general discussions, we may abstract everything as a state irrespective of whether it is an “even” or a “decision”.

stated functional requirements for the Microwave Oven problem are given in Table 1.

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 translation for the requirement 7 (R7) is shown in Fig. 1. From Fig. 1, 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. The “+” sign in the root state “OVEN [Cooking]” indicates these states are only implied in the original requirement. The behavior trees translated for the complete set of requirements can be found in [1].

When requirements translation has been completed, each individual functional requirement is translated to one or more corresponding requirement behavior tree(s) (RBTs). We can then systematically and incrementally construct a design behavior tree (DBT) that will satisfy all its requirements. The process of integrating two behavior trees is guided by the precondition and interaction axioms [1]. If an RBT’s root node exists in another RBT, the RBT can be integrated into the second tree at that point. For example, for the behavior trees of R3 and R6 shown in Fig. 2, it is found that the root node DOOR[Closed] of R3, exists in tree R6, so the RBT of R3 can be integrated with tree for R6 to create a new tree as shown in Fig. 3.

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 shown in Fig. 4 (R8 is a missing requirement from the original functional requirements, but can be easily identified through the common domain knowledge of a microwave oven). 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 task is to retrieve the component interaction network (CIN) and other design diagrams.

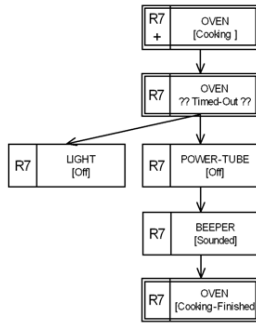


Fig. 1. Behavior tree for requirement R7

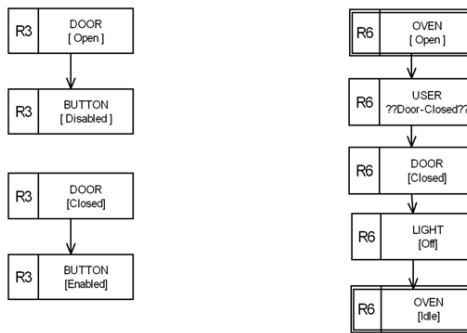


Fig. 2. Behavior trees for requirement R3 and R6

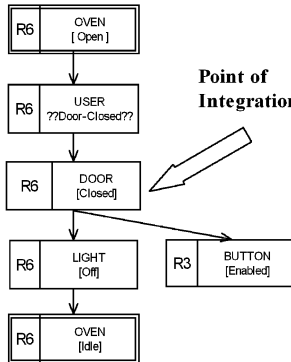


Fig. 3. Result of Integrating R6 and R3 (the second part)

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. 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

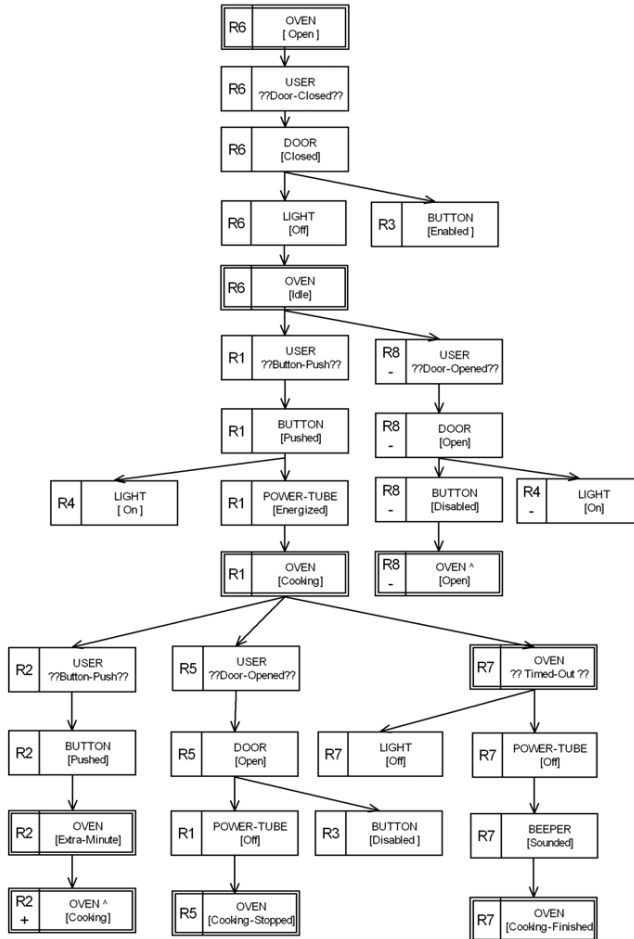


Fig. 4. Integration of all functional requirements

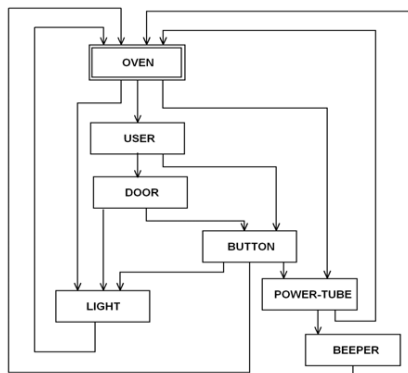


Fig. 5. Component Interaction Network - (CIN)

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), which shows the interaction relationships between components and presents the component architecture.

The CIN derived from the Microwave Oven design behavior tree is shown in Fig. 5. The algorithms to project other types of design diagrams are not related to the topics of this paper, so they will not be pursued here.

### 3 Architecture Transformation Theory

#### 3.1 Definitions

In the original definition of a CIN, a link is directional. If there are two links  $L_a$  and  $L_b$  that connect a pair of components  $C_i$  to  $C_j$  in different directions,  $L_a$  and  $L_b$  are treated as two separated links. In the section, in order to simplify the discussion, we merge  $L_a$  and  $L_b$  into one single link, without explication, any link is supposed to be bi-directional, and a one-way link is only a special case of a two-way link (this difference is unobservable if we abstract a CIN as a bidirectional graph).

**Definition 3.1** A **network** is a graph that includes links and components, each component only appears once in the network and between two different components, there exists at most one link. A link is identified by the two components such as  $(C_i, C_j)$ , where  $C_i$  and  $C_j$  are two components in the network.

**Definition 3.2** In a network  $N$ , if there exists a link between two components, we say that these two components are **directly connected**. Suppose  $C_1, C_2, \dots, C_m$  are  $m$  different components in  $N$ , if for all  $1 \leq i \leq (m-1)$ ,  $C_i$  and  $C_{i+1}$  are directly connected, we say  $C_1, C_2, \dots, C_m$  form a **path** and the length of this path is  $m-1$ .

**Definition 3.3** A network is called a **connected network**, if for all pairs of components  $C_i, C_j$ , which belong to this network, there exists a path starting from  $C_i$  and ending at  $C_j$  in this network.

**Definition 3.4** From a DBT  $T$ , we can project a CIN  $N$  through the algorithm defined in GSE; the CIN is called this DBT's **associated CIN** and it is denoted as  $N = M(T)$ .

**Proposition 3.5** A CIN is a connected network.

**Proof.** Let  $T$  be a DBT and  $N$  be the associated CIN, we have  $N = M(T)$ , and  $C_i, C_j$  are two components belonging to  $N$ . Suppose  $C_r$  is the component associated with the root node in  $T$ . According to the algorithm to project  $N$  from  $T$ , it is easy to prove that there is a path between  $C_i$  and  $C_r$  in  $N$ . Similarly, there is a path between  $C_j$  and  $C_r$ . Merging the two paths together, we have a path linking  $C_i$  to  $C_j$ , so  $N$  is a connected network.

The fact that a CIN must be a connected network is important for proving the paper's main theorem, which shows that the structure of a CIN can be manipulated

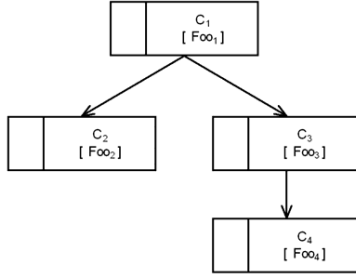


Fig. 6. A simple DBT of 4 components and 4 states

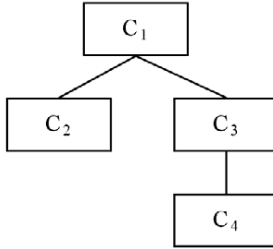


Fig. 7. The CIN  $N$  of  $T$  shown in Fig. 6

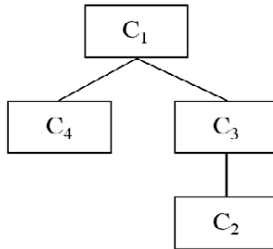


Fig. 8. The desired CIN  $\tilde{N}$

into any preferred form by inserting nodes in the associated DBT. Before we prove this theorem, in the next subsection, we will use a simple example to illustrate the basic ideas.

### 3.2 A Simple Example

Fig. 6 shows a simple DBT  $T$ , and the associated CIN  $N$  of  $T$  is shown in Fig. 7. We have removed the arrows in  $N$  to simplify the discussion.

Now suppose that the CIN  $\tilde{N}$  shown in Fig. 8 is more desirable. The problem is how we could insert bridge component-states in  $T$  to make the new tree's associated CIN become  $\tilde{N}$ .

The link set of  $N$  is  $L_N = \{(C_1, C_2), (C_1, C_3), (C_3, C_4)\}$ , and the link set of  $\tilde{N}$  is  $L_{\tilde{N}} = \{(C_1, C_4), (C_1, C_3), (C_3, C_2)\}$ . Because the links of  $(C_1, C_4)$  and  $(C_3, C_2)$  exist in  $L_{\tilde{N}}$  but not in  $L_N$ , we can add two nodes in  $T$  to create a new tree  $T'$  shown in Fig. 9.

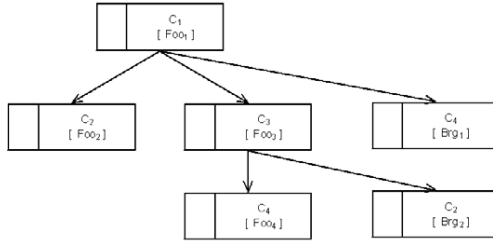


Fig. 9. Two bridge component-states are added into tree  $T$  to create tree  $T'$

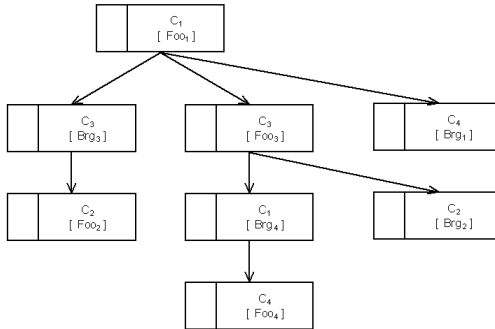


Fig. 10. Two more bridge component-states are inserted to get rid of the unwanted direct connections

Let  $N'$  be the associated CIN of  $T'$ , then it is obvious that the link set for  $N'$  is:  $L_{N'} = \{(C_1, C_2), (C_1, C_3), (C_3, C_4), (C_1, C_4), (C_3, C_2)\}$ . Comparing  $L_{\tilde{N}}$  with  $L_{N'}$  it is found that the links  $(C_1, C_2), (C_3, C_4)$  exist in  $L_{N'}$  but not in  $L_{\tilde{N}}$ . To get rid of the extra links, we need to insert bridge component-states between the unwanted direct connections. In Fig. 9, there is a direct connection from  $C_1[\text{Foo1}]$  to  $C_2[\text{Foo2}]$ . Because  $C_1$  and  $C_2$  are not supposed to be directly connected, we need to insert bridge state(s) between the two nodes. Checking  $\tilde{N}$ , we find the path to link  $C_1$  and  $C_2$  is  $C_1, C_3, C_2$ , so we should insert a bridge component-state of  $C_3$  between  $C_1[\text{Foo1}]$  and  $C_2[\text{Foo2}]$ ; by similar analysis, we know that a bridge component-state of  $C_1$  should be inserted between  $C_3[\text{Foo3}]$  and  $C_4[\text{Foo4}]$ . The result new tree is shown in Fig. 10. Inspecting this tree and we find that if we remove  $C_4[\text{Brg1}]$  and  $C_2[\text{Brg2}]$ , the associated CIN will not be affected. We therefore remove these two nodes to get the final  $\tilde{T}$  shown in Fig. 11.

It is easy to prove that  $\tilde{N} = M(\tilde{T})$ . If we ignore the bridge component-states in  $\tilde{T}$ , the behavior of  $\tilde{T}$  is exactly the same as the behavior of  $T$ . This simple example clearly illustrates how we can transform a component architecture into a new form by inserting bridge component-states into the DBT.

### 3.3 Behavior Invariance Theorem

**Definition 3.6** A **bridge component-state**, also called **bridge state** in short, is a special state in a behavior tree. It is visible when the tree is observed from the solution domain, but it becomes invisible when we observe the tree in the problem domain. It is similar to the concept of a hidden event in CSP [8]. When we observe

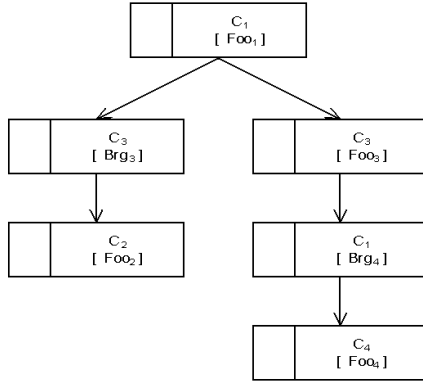


Fig. 11. Prune the unnecessary bridge component-states and get the final  $\tilde{T}$

a system from higher level, some low level details become unobservable

Generally, a design behavior tree (DBT) is a bridge to connect the two domains of a system: the problem domain and the solution domain. In the problem domain, a DBT should capture all the functional requirements and in the solution domain, many design decisions are properties that directly emerge from a DBT.

**Proposition 3.7** *When we insert bridge states in a DBT, the bridge states will not change the functional requirements captured by the behavior tree.*

**Theorem 3.8** *Let  $T$  be a DBT and  $N$  be its associated CIN, where  $N = M(T)$ . Suppose there are a total of  $s$  components  $C_1, C_2, \dots, C_s$  in  $N$  and  $\tilde{N}$  is an arbitrary connected network that includes and only includes those  $s$  components. Then, by adding extra nodes to  $T$ , we can produce a new DBT  $\tilde{T}$  with  $\tilde{N}$  as the associated CIN, where  $\tilde{N} = M(\tilde{T})$ .*

**Proof.** Let us compare  $N$  and  $\tilde{N}$ , because they have the same component set, if they are different, they must have different link sets. If there is a link  $(C_i, C_j)$  that only exist in  $\tilde{N}$ , we can simply add a node of  $C_j$  under a node of  $C_i$  in tree  $T$  to make the associated CIN have link  $(C_i, C_j)$ . So the problem is how we can remove links, which are not in  $\tilde{N}$ , from  $N$  by inserting nodes in  $T$ . If a link  $(C_l, C_k)$  only belongs to  $N$ , then in tree  $T$ , there must be nodes of  $C_l$  that are directly connected to nodes of  $C_k$ . Because  $\tilde{N}$  is a connected network, there must exist a path between  $C_l$  and  $C_k$  in  $\tilde{N}$ . Excluding  $C_l$  and  $C_k$ , supposing the rest part of the path is  $C_{n_1}, C_{n_2}, \dots, C_{n_t}$ , then at the each occurrence of a direct connection between a node of  $C_l$  and a node of  $C_k$  in  $T$ , we add a series of nodes of  $C_{n_1}, C_{n_2}, \dots, C_{n_t}$ . Then the modified behavior tree's associated CIN will not have the direct link of  $(C_l, C_k)$ . Because the inserted nodes are ordered according to an existing path in  $\tilde{N}$ , the insertion of the new states will not introduce extra links that are not in  $\tilde{N}$ .

**Theorem 3.9** *Let  $T$  be a DBT and  $N$  be its associated CIN.  $N$  has  $s$  components  $C_1, C_2, \dots, C_s$  and  $\tilde{N}$  is an arbitrary connected network that only includes those  $s$  components. Then, we can create a new DBT  $\tilde{T}$  that capture the same set of functional requirements as  $T$ , and has  $\tilde{N} = M(\tilde{T})$ .*

This theorem is the direct result from Theorem 3.8 and Proposition 3.7. It states that the component architecture can be independent to the functional requirements. Therefore, it is possible for us to investigate universal optimized software architecture regardless the functional requirements of a particular system. In the next section, we propose a tree-structured architecture as a possible universal optimized form for software architecture due to some unique features of trees.

## 4 Software Normalization

### 4.1 Trees and Normalized DBTs

There are a number of equivalent definitions of trees and a number of mathematical properties that imply this equivalence [9]. Since most of the properties are obvious, we will not repeat some of the proofs.

**Proposition 4.1** *A connected graph is a tree when and only when for each pair of nodes in the graph; there is only one unique path between them. [9]. A connected graph is a tree when and only when there is no circular path.*

**Proposition 4.2** *A connected graph with  $n$  nodes has at least  $(n - 1)$  links. It is a tree when and only when there are  $(n - 1)$  links. In other words, a tree is a connected graph with the least possible number of links [10].*

**Definition 4.3** A DBT is called a **normalized DBT** if the associated CIN is a tree. A software system with a normalized DBT is called **normalized software system** with **normalized architecture**.

**Theorem 4.4** *Any DBT can be normalized (transformed into a normalized DBT) without changing the functional requirements. (Direct result from Theorem 3.9).*

**Proposition 4.5** *For a CIN  $N$  with  $n$  components, the number of the links must be greater than or equal to  $(n - 1)$ . The number of links equals to  $(n - 1)$  if and only if the system is normalized.*

If we use the number of links among components as a measure of the complexity of the architecture of software systems, Proposition 4.5 indicates that a normalized software system has the simplest architecture.

**Proposition 4.6** *Let  $T$  be a DBT and  $N$  be its associated CIN.  $T$  is normalized when and only when for all pairs of components  $C_i$  and  $C_j$  in  $N$ , there exists only one path between the two components in  $N$  provided no node in the DBT is included twice in a path.*

This proposition is a direct result from Proposition 4.1 and the definition of a normalized system<sup>4</sup>. It indicates a very important feature of a normalized software system. For large software systems, we frequently face the problem of passing references, messages or attributes between different components. Because we cannot

<sup>4</sup> For a pair of components, it may have multiple types of information exchanged between them, for example, data flows or controls. However, in this paper, we assume that we can apply one type of abstract connection that can pass all the different types of information.

make each pair of components directly connected, we have to use some components as bridges to pass messages or references. If there are multiple paths between two components, we may not know which paths are used and which are not and it will make the change impact analysis [2] more difficult.

**Proposition 4.7** *If there are no mutual components in two tree-structured CINs, when the two CINs are connected by a link, the new CIN is also tree-structured.*

**Proposition 4.8** *Consider two tree-structured CINs  $N_1, N_2$ . If there is only one mutual component  $C$  in both CINs, the two CINs can be merged through the mutual component  $C$ ; then the merged CIN is also tree-structured.*

**Theorem 4.9** *If a normalized DBT  $T$  is broken into two DBTs  $T_1$  and  $T_2$  by cutting off a link; then  $T_1$  and  $T_2$  are also normalized DBTs.*

**Proof.** If  $T_1$  is not normalized, let  $N_1$  be the associated CIN of  $T_1$ .  $N_1$  is not tree-structured. According to Proposition 4.6, there exists at least a pair of components  $C_i, C_j$  in  $N_1$  that are connected by more than one path. When  $T_1$  and  $T_2$  are merged into the original  $T$ , because no link in the  $T_1$  is lost in  $T$ , the associated CIN of  $T$  has all the links in  $N_1$ . So the multiple paths linking  $C_1$  and  $C_2$  are also in  $T$ 's associated CIN, but this is contrary to the condition that  $T$  is normalized. Therefore, we know  $T_1$  is normalized, and similarly  $T_2$  must be normalized.

Proposition 4.7, Proposition 4.8 and Theorem 4.9 specify an important feature of trees. That is, if a tree is broken into two parts, each part is still a tree; if two trees are integrated into one graph, the graph is also a tree if the integration is based on some specified rules. This feature is important for building large scale systems because the normalization property can be hold in different levels.

## 4.2 Case Study

In the second section, we have used an example of Microwave Oven to explain the fundamental concepts of GSE. Here we will normalize it to demonstrate how the component architecture can be simplified through the normalization. Fig. 12 shows a normalized DBT. The normalized process is a mixture of inserting bridge states and adjusting the order of some states. The bridge component-states are filled with grey. The associated CIN of the DBT is shown in Fig. 13.

Comparing the normalized DBT with the original DBT in Fig. 4, we have found that differences between the two behavior trees are trivial and both DBTs capture all the functional requirements in Table 1. However the differences between the two CINs are significant. The CIN shown in Fig. 13 is much simpler than the original CIN in Fig. 5. Even though the Microwave Oven case study is a small system with only 7 components, the architecture normalization has dramatically simplified the component architecture. If the same process is applied in large systems, we expect that the impact of simplification on the component architecture will be more significant.

The tree shown in Fig. 5 has only two levels. This does not mean that a normalization process can only produce a CIN of two levels. Theoretically, we can have the

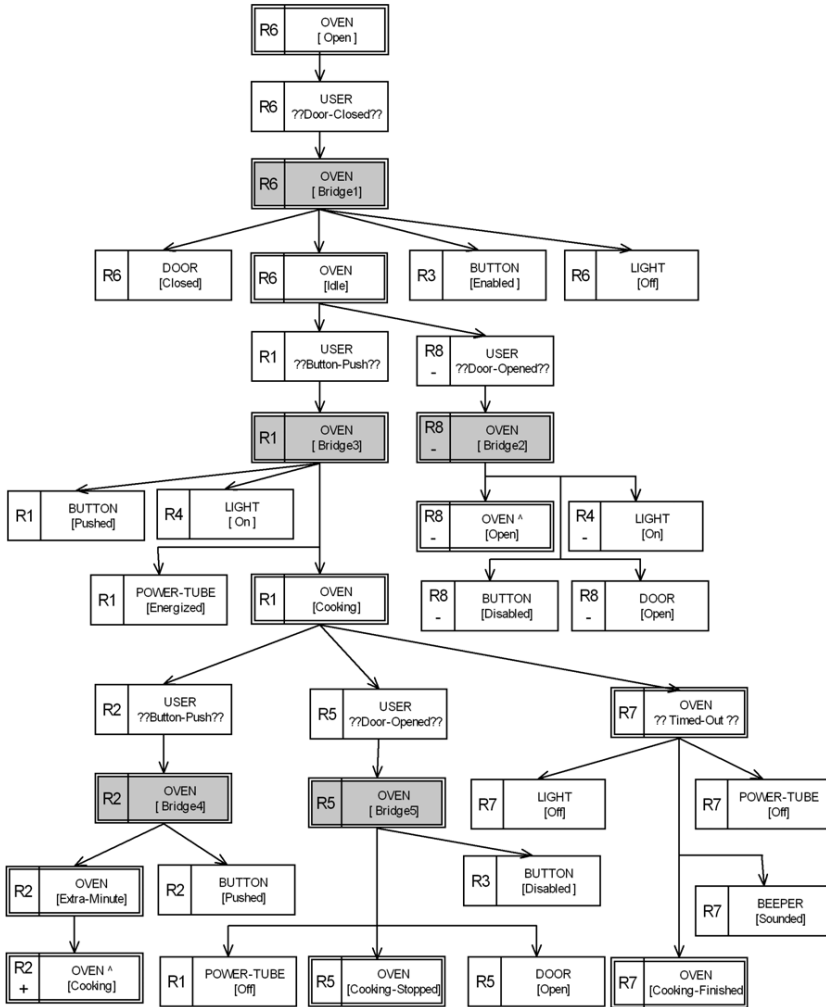


Fig. 12. A normalized DBT for the Microwave Oven case study

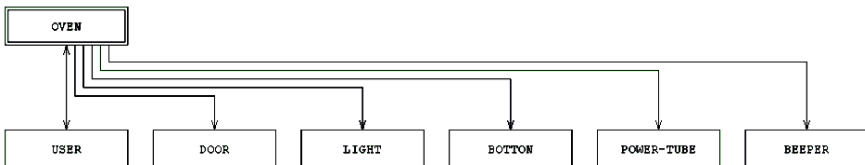


Fig. 13. The tree-structured CIN associated with the DBT in Fig. 12

CIN as any preferred forms, but due to the limitation of space, no further examples can be given in this paper.

## 5 Conclusion

This paper has addressed two things: the relationship between the functional requirements and the component architecture of a system, and the control of changes on the architecture of a system. A consequence of this work shows the advantages of using tree-like architecture as an optimized form due to its simplicity and scalability.

The component architecture of a system must support the implementation of all the integrated behaviors of a system. The latter are in turn implied by the set of functional requirements for the system. Current software engineering practice suggests that, for a given problem, there exist many different approaches to designing a solution to the problem [11] each of which may lead to a system with a different component architecture. What we have sought to do is establish the relationship between a set of functional requirements and the component architecture of a system and then shown how systematic change of the architecture can be achieved without affecting the set of functional requirements that the system satisfies.

Once we have the means to systematically change the component architecture of a system we can equally effectively use this power to resist the consequences of changes on the architecture of a system. It is a well known observation of software engineering practice that repeated change to the functional requirements of a software system tends to gradually degrade the original component architecture and increase the cost of the maintenance. The results in this paper prove that we can usually keep the component architecture stable when a system is changed. This has significant implications for reducing the cost of software maintenance.

## References

- [1] Geoff, R. G., *From Requirements to Design: Formalizing the Key Steps*, (Invited Keynote Address), SEFM'2003, pp. 2-11, Brisbane, September, 2003.
- [2] Wen, L., Dromey, R. G., *From Requirements Change to Design Change: A Formal Path*, SEFM 2004, pp. 104-113, 2004.
- [3] SQI Paper, <http://www.sqi.gu.edu.au/gse/papers>.
- [4] Shlaer, S., Mellor, S.J., "Structured Development for Real-Time Systems", Vols. 1-3, Yourdon Press, 1985.
- [5] Bass, L., Clements, P. and Kazman, R., "Software Architecture in Practice", Addison Wesley Longman, Inc. 1998.
- [6] Stafford, J. A., Wolf, A. L., "Software Architecture", Component-Based Software Engineering, putting the pieces together, Chapter 20, 2001.
- [7] Perry, D., Wolf, A., *Foundations for the Study of Software Architecture*, SIGSOFT Software Engineering Notes, Vol. 17, No. 4, Oct., 1992.
- [8] Hoare, C.A.R., "Communicating Sequential Processes", Prentice-Hall, 1985.
- [9] Knuth, D. E., "The Art of Computer Programming, Fundamental Algorithms", 3rd edition, Vol 1, Addison Wesley Longman, 1992
- [10] Sedgewick, R., "Algorithms", Addison-Wesley Publishing Company, Inc, 1989.
- [11] Glass, R. L., "Facts and Fallacies of Software Engineering", Pearson Education, Inc, 2003.
- [12] Medvidovic, N., "One the Role of Middleware in Architecture-Based Software Development", SEKE'02, 2002.