A HIERARCHICAL ARCHITECTURE TO MODEL COMPLEX SOFTWARE INTENSIVE SYSTEMS IN BEHAVIOR TREES

L.Wen, R.G. Dromey,
Software Quality Institute, Griffith University,
Nathan, Brisbane, Qld., 4111, AUSTRALIA.
l.wen@griffigh.edu.au, g.dromey@griffith.edu.au

Extended Abstract

1 The Problem: The need for a systematic approach to building complex software intensive systems

With the quick development of computer hardware, many complex systems are becoming software intensive. For example an aircraft, which used to be a pure hardware artifact, includes more and more software components for communication as well as flight control, so it becomes a software intensive system. How to build a high quality software intensive system has always been a big challenge, especially, with the fact that software intensive systems are tended to merge together to form higher level super systems, which are inevitable much more complex; e.g. a new airline management system may integrate the air ticket sales system and the airline plan system to provide a better management tool for an airline company.

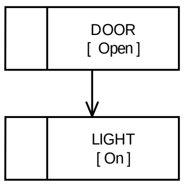
The reasons of difficulties to develop complex software systems have been addressed thoroughly in software engineering. Generally, there are three fundamental reasons. The first reason is because of the intrinsic nature of complexity in those systems. The second reason is the unavoidable changes of the software requirements. The third reason is about the defects of software requirements, which are frequently inconsistent, incomplete and ambiguous for large and complex systems; because for a large system, the requirements are assembled from different stakeholders who may be of different domains and may have different views of the system.

2 Approach proposed by this paper. A hierarchical architecture modeled by behavior trees

In this paper, we use behavior trees as a formal language to model a hierarchical architecture for software intensive systems. This architecture could be one of the most suitable architectures for large and complex software intensive systems because of its simplicity and scalability.

2.1 Behavior trees

Behavior trees, firstly proposed in 2000, are a tree-structured formal graphical language to model the behavior of software intensive systems. In the past ten years, behavior trees have been explored in a broad range of software engineering areas including: model checking, software change and evolution, software safety and security etc and received positive results. In this abstract, due to the limitation of space, we use only a small fraction of a behavior tree to illustrate the main ideas.



The figure above, which is a small behavior tree, shows that there are two components “DOOR” and “LIGHT” in a system; when the “DOOR” realizes the state of “Open”, it will cause the “LIGHT” to realize the state of “On”.

2.2 Simple systems and complex systems.

In the hierarchical architecture, we define two different types of systems: simple systems and complex systems. A simple system is defined as a system that contains no other lower level systems as its components; in other words, it is a lowest level system.

Contrary to a simple system, a complex system is a system with lower level systems as its components. For example, supposing that this is a “Light” as one of the simple systems, if there is a system “Oven” that contains a “Light” as its component, the “Oven” is a complex system. Similarly, the “Oven” can be used as a component in a “Kitchen” system and the “Kitchen” in a “Building” system etc.

2.3 Environment, system and component

In the proposed 3-tier architecture, each entity can be treated as a system, a component or even an environment depending on the observing viewpoints. For example, we consider a “Kitchen” system shown in Figure 1, which contains an “Oven” and a “Fridge” as its components, and “Oven” contains a “Light” and a “Button” as its components.

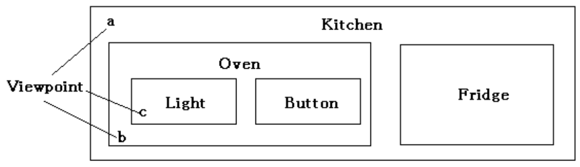


Figure 1 Three different viewpoints

If a person is observing at viewpoint **a**, “Kitchen” will be the system, “Oven” and “Fridge” are components, and “Light” and “Button” are invisible. If the observing point is at b, “Kitchen” will be the environment, “Oven” is the system, “Light” and “Button” are components and “Fridge” is invisible. If the viewpoint is at c, “Light” will be the system, there is no components as “Light” is a simple system, “Oven” is the environment and anything else is invisible.

2.4 Internal and external boundary

When a system is observed from inside, the system’s behavior can be modeled by a behavior tree. From this behavior tree, the internal boundary of the system is formally specified.

When we observe the system from its environment, which is a super system, the behavior of the super system can also be specified as a behavior tree. Similarly, from the new behavior tree, the boundary of the previous system, which now is treated as a component, is also formally specified.

The two boundaries of a system must match each other, so the system can execute its designed functions in its environment. This principle is called the *mutual boundary property*.

2.5 Tree-formed structure

In the proposed architecture, every system is connected to only its environment and its components. There is no direct connection between the components. Therefore the whole system is built in a strict tree-formed architecture. The integration of a system to its environment is based on the mutual boundary property.

3. Advantages of the proposed approach

The proposed architecture has the following advantages:

Decompose the complexity: It decomposes the complexity of a large system into many systems of different levels. The complexity of each single system can be reduced to a manageable scale because all the connections to the system are local.

Less number of couplings: Reducing the number of couplings between components is a normal method to simplify a complex system. In the proposed architecture, because of its tree-structure, the number of couplings has been reduced to minimum.

Easy to change: Whenever a component is changed, the impact is easy to trace and the ripple effects can be easily blocked because of the tree-structure, which has no circles.

Easy to reuse: For each system, all its components are independent to each other, so it is easy to reuse any components in other systems.

Easy to integrate: To integrate a system into a parent system, we only need to exam the child system’s boundary without concerning its internal compositions. Also, no matter how many systems are integrated and how many levels are formed, the overall tree-structured style will be kept.

Easy to test: In the hierarchical architecture, each system is self-sustaining if the environment can simulate the behavior of its external boundary. Therefore, starting from the low level systems, completed unit testing can be performed in the early stage of development and high level system testing can be simplified when all the included lower level systems have been tested or simulated.

4. Supporting tools and case studies

Based on this proposed hierarchical architecture, a software tool called BECIE (Behavior Engineering Component Integrated Environment) has been developed. We have used this tool to successfully model and simulate a number of software intensive systems such as Microwave Oven system, Car Park system. Especially, we successfully use BECIE to model and simulate a Universal Turing Machine. Because any computerable system can be simulated by a Universal Turing Machine, our results indicate the proposed architecture plus behavior tree modeling language are sufficient to model any software intensive systems.

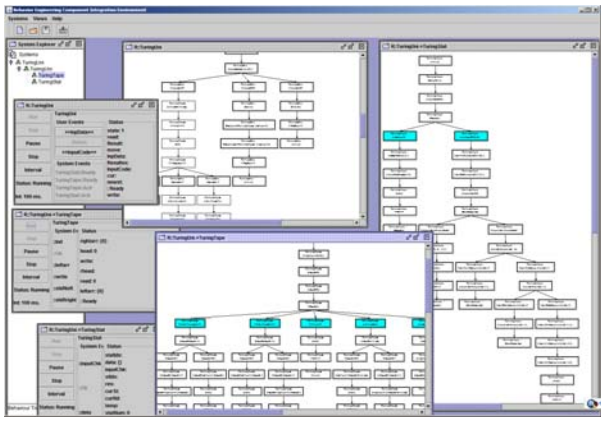


Figure 2 A universal Turing Machine simulated by BECIE

5. Conclusion

This paper proposes a hierarchical architecture to model large and complex software intensive systems. The main idea is that in the whole system, every entity is a system by itself and it connects only to its environment and its components. Each entity is specified by a behavior tree and the integration between two entities is based on the mutual boundary property. Because of its simplicity and scalability, the architecture is ideal for modeling large and complex systems. Finally, a software tool is reported to successfully model and simulate a number of software intensive systems including a Universal Turing Machine built in this hierarchical architecture.