ICIC Express Letters ICIC International ©2017 ISSN 1881-803X
Volume **11**, Number **1**, January **2017** pp. 1–9

BT EXPANSION FOR EXTRACTING SOFTWARE ARCHITECTURE FROM REQUIREMENTS

Hongyan Yao, Xuebo Sun and Han Bao

School of Software
University of Science and Technology Liaoning
No. 185, Qianshan Middle Road, Lishan District, Anshan 114051, P. R. China
abroat@163.com

Received June 2016; accepted September 2016

Abstract. *How to translate the functional requirements into software architecture in a systematic and constructive way is a problem that is not addressed in the scope of software requirement engineering. To our knowledge, the work for building software architecture is almost empirical. Existing works recently employed the Behavior Tree, BT, to solve this problem but the achieved software architecture is not integrated, lack of events-handler & trunk data and interface definition, which are valuable information for guiding next-step design. In this paper we strengthen the BT based approach and present a proposal that is capable of building a complete software architecture which consistently reflects the functional requirements as well as provides sufficient formal information for guiding subsequent design. Simulation result and relevant analysis both indicate the proposal is feasible and effective.*

Keywords: Requirement engineering, Software architecture, Behavior Tree (BT), Interface

1. Introduction.

Software Architecture (SA) is the first artifact at the early stage of software design, and it is significant in the following aspects [1]: 1) SA is a global solution for expressing all requirements including Functional Requirements (FR) and Non-Functional Requirements (NFR), and all the other necessary properties or constraints; 2) It is useful as a manner for communication between stakeholders; 3) The changed requirements could be reflected conveniently on SA; 4) SA is a good start to make subsequent design; 5) Many tests are performed based on SA.

Since SA is so important for software design, it is better if there exists a systematic and constructive way to guide a process proceeding in transformation from FR to SA design. However, to our knowledge, SA design is mostly an empirical study and lack of effective and intuitive method. Even UML (United Model Language), an industry standard for object-oriented design, does not show a manner to translate directly from functional requirements to SA, and this opinion had been expressed and discussed in a series of papers written by Wen et al. [2-5]. We agree to their opinions. Actually, according to our previous work on SA [6,7] and our empirical study using UML to design, it has been realized UML is the most effective if we have an SA in advance. The following design steps will expand fluently if SA exists. So in short, SA is the key for subsequent design.

About how to transform functional requirements to SA Dromey had shed a new light on it and had got some meaningful achievements, which is significant for further study. They employed the Behavior Tree (BT) to formalize functional requirements, and proposed a sequential process to form at last an SA. They bring a way feasible to design SA. However, in our opinion their SA as a result is not complete and lack of events-handler & trunk data and interface definition. The artifact they achieved just indicates what components are composed of. We cannot get explicitly the component interfaces for guiding subsequent iterative design with UML. After all, an SA without interfaces signatures is little valuable. Just alike to Robert Martin who states in his book: a class diagram without function signatures is little valuable [8].

In this paper, we firstly explore the relationship between BT and SA to show what a qualified SA should hold; after that, we continue to work on the transformation from FR to SA, and propose an expansion version of BT-based approach that presents the process for reaching an interface-integrated SA. Through the simulation and relevant analysis it could be proven that our proposal is feasible and effective; furthermore, the SA attained by our proposal is more complete than that of the original method. Rest of the paper is organized as the following: next section will analyze the relationship between SA and BT; Section 3 will present our proposal for designing a complete SA, and then in Section 4 a simulation and a relevant analysis are provided to present the feasibility and effectiveness of our proposal. Section 5 concludes the paper.

2. **Analysis for Relationship between SA and BT.** SA [9] is the first artifact developed within the domain of design, right after the requirements analysis. SA contains a set of views concerning about the system, like function view, component view, process view, and deployment view. Among those views the component view is the foundation. It relates closely to the subsequent design process. Figure 1 shows the responsibility of SA in an engineering process. In Figure 1 the light color layer represents the artifact achieved in each engineering stage; the dark color layer denotes the actions performed for transformation between two adjacent layers. Figure 1 indicates that initial requirements described in natural language could become explicitly specified FR and NFR through the manner of Refine & Define. Refine refers to smooth the relations existing within FRs in case there are details missing; Define refers to listing the specific function. Following Refine & Define the FR and NFR are the results. FR explicitly records the functions hold by the system; NFR demands what constraints should be met when certain function executes. It is self-evident NFR depends on FR, and it is the FR design rather than NFR that makes the system runnable, hence FR design is the key, and also FR design is the main job for SA; as for NFR, it is mainly delegated to subsequent design which chooses among alternative solution as response to NFR. So next in Figure 1, are there constructive methods existing for transforming the FR into SA? If the answer is yes, it will be barrier free to go down and make subsequent design with UML. Actually, the transformation between FR and SA usually resorts to the architecture's experience instead of a systematic constructive method. Fortunately, in recent years some representative achievements based on BT are reported on this aspect. They employed the BT to realize that transformation. However, their final artifact merely presents Component & states, the other two main elements (Trunk Data, Interfaces) that, make up of the SA are not contained.

BT-based approach is an alternative approach that is matching to "Abstract & Extract" in Figure 1. The approach provides an intuitive manner to construct a design out of a set of FR [10]. One at a time, an individual requirement is transformed and expressed formally in a BT; after an integration of all those BTs, a final artifact called Design Behavior Tree (DBT) should be the result that is matching to SA in Figure 1. As the BT-based approach is actually describing how a system is transferred between its stats, it is clear and simple to get from DBT how many components are composed of in the system and how many abstract/concrete states there are in the system. We could observe the control-flow clearly in DBT, but in DBT we cannot get the knowledge of data-flow and what function is invoked for state transfer. That is somewhat deficient to guide into the next design stage; so the achieved DBT without trunk data and interfaces is incapable to be titled as a complete SA. Details analysis about this is going to be explored in Section 3.

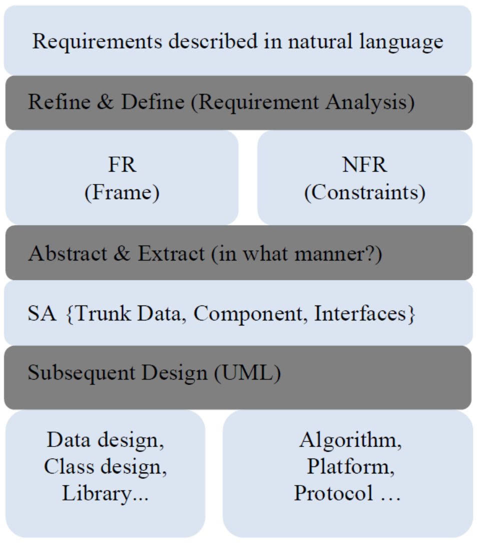

Requirements described in natural language
Refine & Define (Requirement Analysis)
FR
(Frame)
NFR
(Constraints)
Abstract & Extract (in what manner?)
SA {Trunk Data, Component, Interfaces}
Subsequent Design (UML)
Data design,
Class design,
Library...
Algorithm,
Platform,
Protocol ...

Figure 1. The responsibility of SA in an engineering process

3. A Proposed BT Expansion for the Transformation from FR to SA.

3.1. Analysis for the deficiency of existing BT-based approach. Using BT to translate FR should pass through five steps: 1) Translate each individual FR into individual BT; 2) Combine all BTs into a DBT; 3) Check if some components lack precondition that prevents them from integrating into the DBT, or check if the component could respond to the events properly; 4) Distinguish all components; 5) Identify all states that belong to the component. We use an example of oven simulation to prove that after these 5 steps the final artifact attainted is not a qualified SA.

Below, Figure 2, is a set of requirements for a microwave oven. Our goal is to translate them to form an SA. According to BT principles all requirements can be transformed into BT. We choose three of them, R3, R7, R6, as representative to show how to make the transformation.

```
R1. a single control button available for the user of the oven. If oven is idle with
    the door closed and you push the button, oven will start cooking, for one minute.
R2. each time the button is pushed when oven is cooking, one extra minute is added.
R3. pushing button will be ineffective when door is open.
R4. when oven is cooking or door is open, the light in the oven will be on.
R5. opening the door stops cooking.
R6. closing the door turns off the light. This is the normal idle state.
R7. if oven times-out the light and power-tube are turned off and then a beeper emits sound.
```

Figure 2. A set of requirements for microwave oven

R3 stands for an NFR, also known as Constraint. R3 can be translated into Figure 3 as two BTs. In Figure 3, according to the requirement specified by natural language it is evident this requirement is a constraint instead of a function. As there are two components, DOOR & BUTTON, in R3 context, their relationships can be expressed as a control flow between their states. The semantic for Figure 3 is like this: if DOOR realizes the state [open], the control will be passed to BUTTON and BUTTON will set its state [disabled]; likewise, if DOOR is in state [closed], then BUTTON sets its state [enabled]. In Figure 3, “C” denotes that the conversion from DOOR to BUTTON is a constraint; “+” indicates the binding component and its state is implicitly implied and approved by the requirement description.

Another requirement R7 can be translated as BT into Figure 4. That BT can clearly express the behaviors described by R7. In Figure 4 “??Timed-Out??” denotes the semantic of "When"; while the rectangle with double line border indicates it is a system-state which is a start point to expand subsequent behaviors. So the behavior semantic BT expressed in Figure 4 is like this: OVEN realizes its state [cooking] now and when "Timed-Out" events happens, LIGHT & POWER-TUBE will both realize their states [off]; after that, BEEPER realizes the state [sounded]; at last, OVEN reaches a system-state [cooking-finished]. This BT matches R7 closely and reasonably.

How to integrate all BTs to become an overall structure called DBT when the transformation like Figure 3 and Figure 4 is finished? The manner is simple and clear. See Figure 5. The combination happens when two BTs have the same component with the same state. It can be observed that BTs for R3 and R6 have a matched state which provides a point of integration ("@" denotes this semantic). It is evident to notice the integration result in Figure 5, which is already integrated. In this manner, a DBT could be obtained. After iteratively handling the other BTs and further integrating them into one DBT a

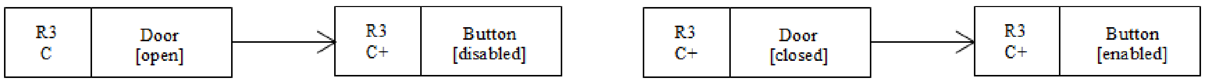


FIGURE 3. Two BTs corresponding to R3

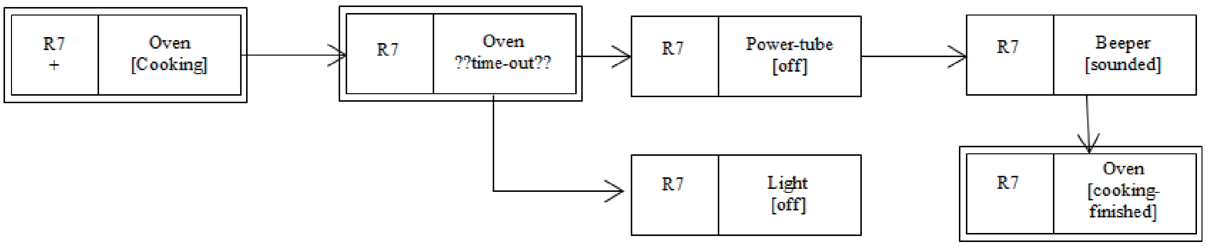


FIGURE 4. BT for R7

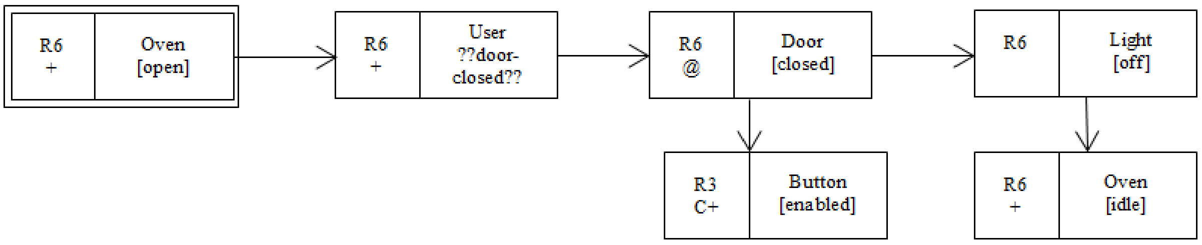


FIGURE 5. A BT integration between R3 and R6

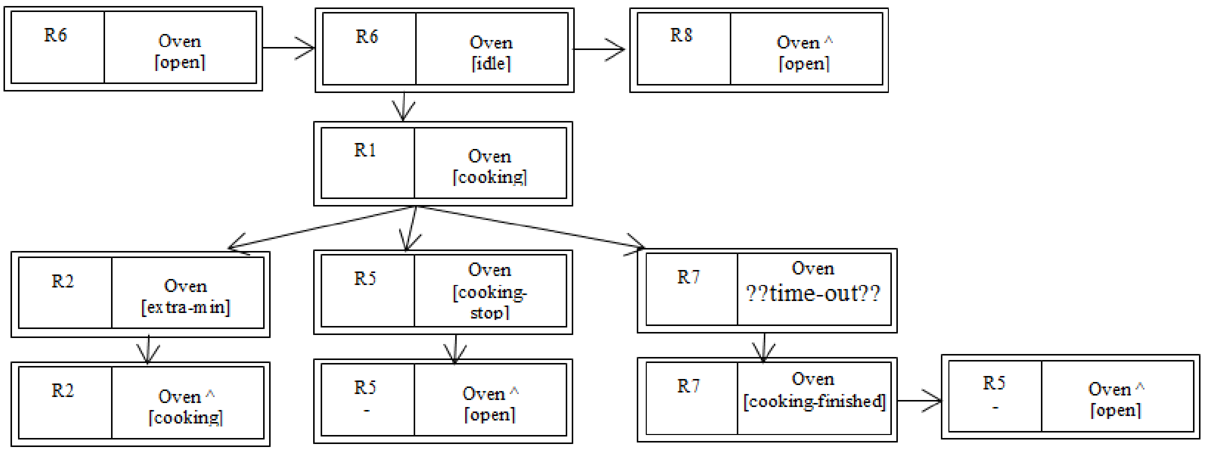


FIGURE 6. Final DBT artifact achieved by existing BT-based approach

final artifact that reflects all OVEN states could be attained in Figure 6. In Figure 6 we can get all states belonging to the component OVEN. Symbol of ^ indicates the state could be backward to the preceding one, which usually means the original requirement depiction is incomplete (we could define R8 additionally as a supplement) or missing ("–" denotes this in semantic).

Figure 6 is the final artifact we achieved using existing BT-based manner. In the figure we can realize intuitively how many system states there are, but we cannot get the knowledge for proceeding to the next design. 1) How does it transfer the control flow among those states at what condition. It is well known the state diagram is little useful if it does not provide the information for condition. So does this one. 2) It is unclear to invoke what function to handle what data. State transfer demands and depends on data. Since Figure 6 focuses on system state transfer, it is inevitable that it should provide trunk data to be supportive. However, Figure 6 does not show them.

Actually, Figure 6 is neither a pure state diagram nor a pure flow diagram. It is rather a mixture of them. As you can see in Figure 6 the ??time-out?? is viewed actually as an event instead of a state and it is different from all the other nodes in semantic. In our opinion it is inappropriate to bring a ??time-out?? into a diagram with main concerning of states.

Based on the above analysis existing BT-based method is good at translating functional requirements into a DBT from which we can master the framework of system-level state transfer. However, this is not sufficient to get software architecture: lack of trunk data and the condition laid on them.

3.2. **BT expansion proposal for getting a complete SA.** Since the existing BT-based approach lacks the necessary elements to finish an SA, we expand it to form a BT-based proposal that can complement this deficit. As depicted in Figure 1, the SA contains three elements: components & states, interfaces, and trunk data; our proposal is composed of three steps as a counter-part.

1) Get system components & states.

2) Obtain events for preparing the data and interfaces.

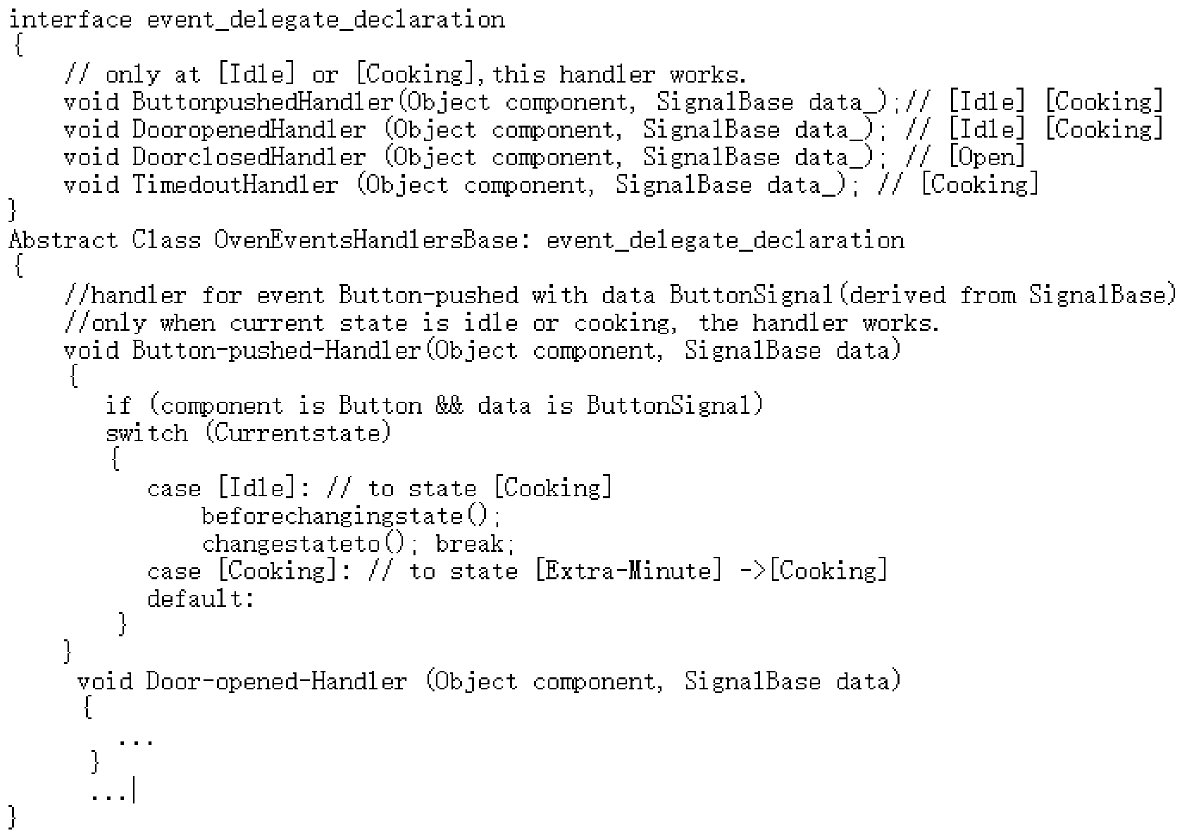
3) Design interfaces through delegation principle.

Existing BT-based approach is capable to get components & states, so step 1 here will not show any unnecessary details. We start from step 2 to continue. Still we use Figure 2 as an example. When each requirement listed in Figure 2 is transformed to BT and all BTs are integrated into DBT, we could conveniently observe those events located in the middle of adjacent system-states. For instance, event "Timed-Out" in Figure 6 is the key to triggering the control flow from [Cooking] state of OVEN to the following state [cooking-finished]. The events are important for state transfer; therefore organizing all events from DBT into a table like Table 1 is the main goal of step 2. In Table 1 we could clearly get to know what events and data need to be handled for state transfer. For instance, the handler for Button-pushed event will use Button-signal data to make the transfer from state [Idle] to state [Cooking], or from state [Cooking] to state [Extra-Minute] ([Extra-Minute] naturally transfers itself to [Cooking] without any conditions. "→" denotes the unconditional transfer). After we check to ensure that all events are taken into consideration and all states are reachable, step 2 is finished; Table 1 achieved in this step will be a start point for step 3.

In step 3 we refer to Table 1 to form the interface signature, see Figure 7. In Figure 7 the *event_delegate_declaration* Interface declares the system state transition and establishes the relationship between components and data (the types for component and data, *Object* and *SignalBase*, are all abstract). In this way, a framework built on components, EventHandler, and data would be settled down. For better describing the sense of the interface we have been provided an abstract base class, *OvenEventHandlerBase* as

Table 1. Relationships of events, trunk data, and pre/post states

Events	Data	From State	To State
Button-pushed	Button-signal	[Idle]	[Cooking]
		[Cooking]	[Extra-Minute] → [Cooking]
Door-opened	Door-signal	[Idle]	[Open]
		[Cooking]	[Cooking-Stopped] → [Open]
Timed-out	Time-signal	[Cooking]	[Cooking-Finished] → [Idle]
Door-closed	Door-signal	[Open]	[Idle]



```
interface event_delegate_declaration
 {
    // only at [Idle] or [Cooking],this handler works.
    void ButtonpushedHandler(Object component, SignalBase data_);// [Idle] [Cooking]
    void DooropenedHandler (Object component, SignalBase data_); // [Idle] [Cooking]
    void DoorclosedHandler (Object component, SignalBase data_); // [Open]
    void TimedoutHandler (Object component, SignalBase data_); // [Cooking]
}
Abstract Class OvenEventsHandlersBase: event_delegate_declaration
 {
    //handler for event Button-pushed with data ButtonSignal(derived from SignalBase)
    //only when current state is idle or cooking, the handler works.
    void Button-pushed-Handler(Object component, SignalBase data)
    {
       if (component is Button && data is ButtonSignal)
       switch (Currentstate)
        {
          case [Idle]: // to state [Cooking]
              beforechangingstate();
              changestateto(); break;
          case [Cooking]: // to state [Extra-Minute] ->[Cooking]
          default:
        }
    }
     void Door-opened-Handler (Object component, SignalBase data)
      {
        ...
      }
      ...|
}
```

Figure 7. Interface declaration and abstract implementation for it

a foundation to support the interface signatures. *event_delegate_declaration* and *Oven-EventHandlerBase* are consistent translation along with Figure 6 and Table 1. They are not trivial but necessary to be designed for guiding the subsequent classes design. After all, a code frame is worth more than a single interface declaration, and the information provided by code frame is worth more than something documented in natural language. We believe that so far the elements for building an SA after above three steps are quantitatively sufficient: components & states (Figure 6), events & data (Table 1), interface frame (Figure 7). That means the microwave oven system comes to the end of SA design. The next is the detailed design and implementation, which, though are not our focus, is still expanded a little in Section 4.1 for further demonstrating the artifacts achieved so far are effective.

4. Simulation and Comparison.

4.1. **Microwave oven simulation.** In Figure 8, component classes such as Buttonsim, Doorsim, Light, and Power are derived from *event_delegate_declaration* Interface. In a like manner concrete data classes such as TimeSignal, ButtonSignal, and DoorSignal are

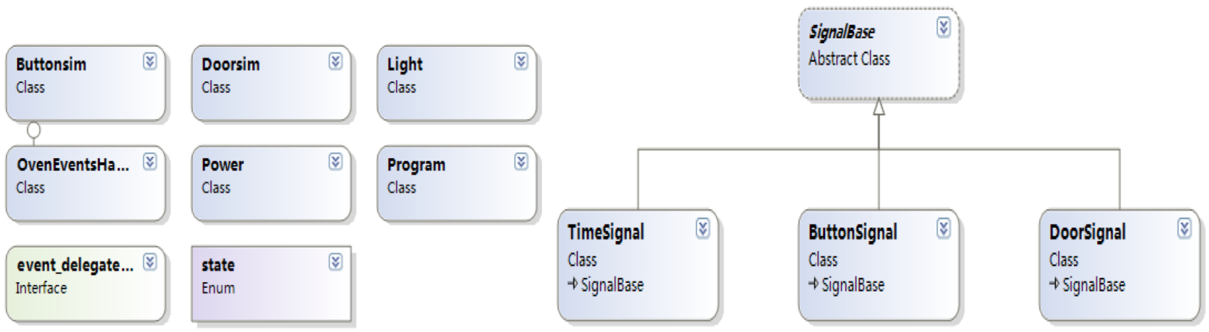
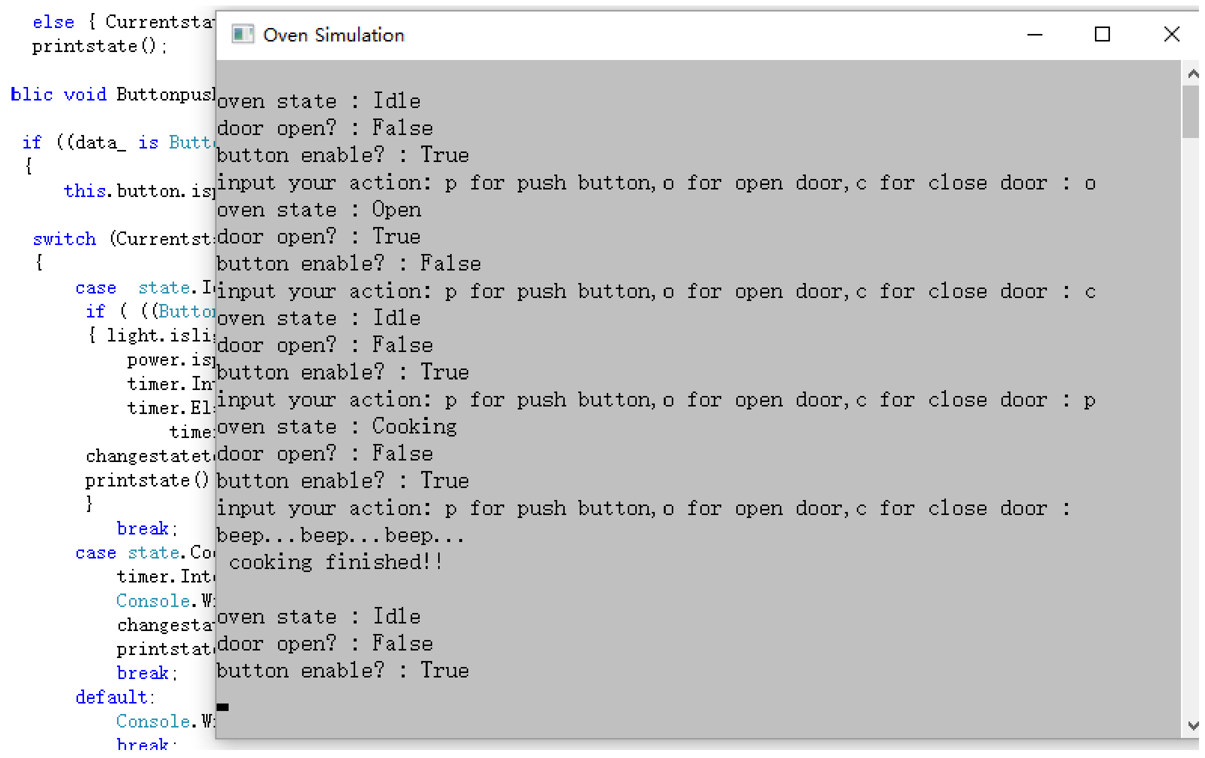


FIGURE 8. A detailed design with respect to the SA obtained through our proposal



```
  else { Currentsta
  printstate();

blic void Buttonpusl

 if ((data_ is Butt
 {
     this.button.is

  switch (Currentst
  {
      case  state.I
       if ( ((Butto
       { light.isli
           power.is
           timer.In
           timer.El
               time
       changestatet
       printstate()
       }
          break;
      case state.Co
          timer.Int
          Console.W
          changesta
          printstat
          break;
      default:
          Console.W
          break;
```

```
Oven Simulation

oven state : Idle
door open? : False
button enable? : True
input your action: p for push button,o for open door,c for close door : o
oven state : Open
door open? : True
button enable? : False
input your action: p for push button,o for open door,c for close door : c
oven state : Idle
door open? : False
button enable? : True
input your action: p for push button,o for open door,c for close door : p
oven state : Cooking
door open? : False
button enable? : True
input your action: p for push button,o for open door,c for close door :
beep...beep...beep...
 cooking finished!!

oven state : Idle
door open? : False
button enable? : True
```

FIGURE 9. Microwave oven simulation result

defined. Those classes and data are all ready for supporting the realization of *OvenEventHandlerBase*. Once the *OvenEventHandlerBase* links well with all the other classes definition, an abstract OvenSystem model is well finished. If you want to run the model, declare a derived class from *OvenEventHandlerBase*, and then make an instance of it.

We have been fully coded to build an abstract model, and the simulation runs successfully and the results are correct as we expected. The simulation result is shown in Figure 9. The initial state of oven is Idle, door is not open, and button is enabled. We input 'o' to open the door at first; then 'c' to close door after putting food in; finally, 'p' as pushing button for cooking. After a while, oven makes sound "beep" to show it is finished; meanwhile, the oven in state Idle again, door is not open, and button is enabled. No matter in what sequence the 'o', 'c', and 'p' are triggered. The results are all reflecting the reality.

4.2. **Comparison with existing work.** The proposal consists of 3 steps and the goal is to get an SA which, on one hand, expresses FR formally and without information missing; on the other hand, it holds sufficient information for subsequent detailed design. The SA attained at last can classify components & states and event handlers & data through interfaces definition. With all these elements it is sufficient to support a qualified SA and subsequent design. In short, the SA can explicitly express what data can be handled by what event handlers on what state. Existing BT-based methods do not strengthen events and data, nor does the interface frame, they just define the component and states. Though component and states are important for building SA, they are just a static diagram which does not indicate what behaviors could trigger the states to change. In other words, we are not aware of what function signature is invoked to drive the state to change – this is just the flaw of existing BT method. Only components and states are not sufficient for SA, because that cannot contribute to the inner design of components. It also means the connection between SA and component design is interrupted. Just like Robert Martin said "a class diagram without function signature is little useful". The SA has the same reason, too.

As our proposal is originated from BT-based approach and it operates for state change, it is inevitable to compare ours with State Transition Diagram (STD) of UML. There are 5 major contrasts between the two methods. 1) STD is a network rather than a tree. In STD it is not convenient to observe which state path matches which FR; however, in BT it is clear to trace the corresponding FR. 2) In BT tree we clearly know what we do until we get to the next state even though there are some state nodes in certain tree path; in contrast, STD uses the sub-states to describe the process and that will make the STD a mass. 3) In our approach each stage is explicit. We get three artifacts through 3 steps of our proposal; while in STD components, state, events, and data are mixed up. 4) Ours aims to provide an SA, at component level, while STD is more suitable to model an object's states. They are not at the same granularity level. 5) Ours provides explicit interface frame definition; in contrast, STD is seldom used to define the interface. About more comparisons with other methods please refer to [11-13] and the web site: http://www.beworld.org/BE/.

5. **Conclusion.** SA is the first artifact in the design domain, and it can guide the subsequent detailed design. However, how to translate FR into an SA is a matter that is not addressed. To date, building SA is almost empirical and there is not a systematic and constructive way for building formally the SA. The existing works that concern this issue are associated with BT that is effectively used for getting components and states. The existing BT-based approaches are effective in getting an SA, but still there exist deficits and that result in the SA obtained is not complete, lacking events-handler & data and interfaces frame definition, which are important elements for subsequent design. After fully analyzing the existing BT-approach we proposed a method for getting a complete SA, which consistently reflects the FR as well as provides sufficient formal information for guiding subsequent design. The simulation result proves our proposal is feasible and effective. Finally, we make an analysis about the proposal and a comparison with related work, STD in particular, to indicate that our expansion for BT-based proposal is able to get a complete SA. To formalize our proposal in order to develop a tool for supporting SA extraction from requirements is our future work.

REFERENCES

[1] L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[2] L. Wen, D. Tuffley and R. G. Dromey, Formalizing the transition from requirements change to design change using an evolutionary traceability model, *Innovations in Systems and Software Engineering*, vol.10, no.3, pp.181-202, 2014.

[3] T. Myers and R. G. Dromey, From requirements to embedded software – Formalizing the key steps, *Australasian Conference on Software Engineering (ASWEC 2009)*, pp.23-33, 2009.

[4] R. G. Dromey, Formalizing the transition from requirements to design, *Mathematical Frameworks for Component Software – Models for Analysis and Synthesis*, 2006.

[5] A. Kushal, M. A. H. Newton, L. Wen and A. Sattar, Formalisation of the integration of behavior trees, *Proc. of the 29th IEEE/ACM International Conference on Automated Software Engineering*, Vasteras, Sweden, 2014.

[6] H. Yao and Y. Ma, An exploration for the software architecture description language of WRIGHT, *ICIC Express Letters*, vol.8, no.12, pp.3481-3487, 2014.

[7] H. Yao, Y. Jiang and W. Shen, A revised orthogonal software architecture: COA, *ICIC Express Letters*, vol.7, no.8, pp.2255-2261, 2013.

[8] R. C. Martin and M. Martin, *Agile Principles*, Patterns, and Practices in C#, Prentice Hall, 2008.

[9] L. E. Lin, From requirements to architectural design – Using goals and scenarios, *International Conference on Software Engineering*, 2003.

[10] *Behavior Engineering World*, http://www.beworld.org/BE/.

[11] A. Ciancone, A. Filieri and R. Mirandola, Testing operational transformations in model-driven engineering, *Innovations in Systems & Software Engineering*, vol.10, no.1, pp.19-32, 2014.

[12] S. K. Kim, T. Myers, M. F. Wendland et al., Execution of natural language requirements using state machines synthesized from behavior trees, *Journal of Systems & Software*, vol.85, no.11, pp.2652-2664, 2012.

[13] P. A. Lindsay, P. A. Strooper, S. Kromodimoelio and M. Almorsy, Automation of test case generation from behavior tree requirements models, *Australasian Conference on Software Engineering (ASWEC 2015)*, Adelaide, Australia, 2015.