An Environment for Building a System out of its Requirements

C. Smith, K. Winter, I. Hayes, G. Dromey, P. Lindsay, D. Carrington
ARC Centre for Complex Systems, University of Queensland 4072, Australia
camerons@itee.uq.edu.au

Abstract

A toolset for system design and analysis is described. The tool allows individual translated functional requirements to be entered graphically as Behavior Trees. Once integrated these Behavior Trees form a problem domain representation of the design. This representation is automatically mapped to CSP to enable model checking with FDR. A number of consistency checks on the design can be performed. Examples are used to illustrate the results produced by the toolset.

1. Introduction

The design environment is an integrated set of tools that support system design. Genetic design [1] is based on the idea that individual functional requirements represent fragments of behaviour, while a design that satisfies a set of functional requirements represents integrated behaviour. This perspective admits the prospect of constructing a design out of its requirements. A formal representation for individual functional requirements, called Behavior Trees (BTs), makes this possible. Behavior Trees, derived by rigorous translation from individual functional requirements stated in natural language, may be composed, one at a time, to create an integrated Design Behavior Tree (DBT). We can then transition from this problem domain representation directly and systematically to a solution domain representation of the component architecture of the system and the behaviour designs of the individual components that make up the system - both are emergent properties of a DBT. It is also possible to conduct a number of formal verification checks on a design behaviour tree.

We are building tool support for this whole approach to design. The environment currently consists of two tools:

- A Behavior Tree Environment (BTE) that allows the user to create and edit graphic representations of BTs.
- A system (BC) that uses the formal semantics of BTs to map the graphical representation to CSP [2] and to a separate representation that supports consistency checking. We can check the design for a variety of problem-independent defects.

Both tools have been developed to operate independently using XML as a common interface. They also operate as a combined environment. Future plans include the development of an Agent-Based Prolog translation to support system simulation.

2. Visual Design with BTE

BTE allows the user to create a Behavior Tree using simple actions performed on nodes and the local and global component libraries (e.g. double click to add a child node, right click and select to edit a node). BTE also provides a basic level of automation to increase the ease of use and restrict the tree to the defined form [1] (e.g. there is no child link without a child node) thus increasing the productivity of the developer and the quality of the design produced.

Following are some of the ways in which user input is partially automated or restricted:

- Node details are able to be selected from a pop-up menu.
- A local component library dynamically captures component/state information. A global component library contains saved component/state details.
- A simplified architecture of the system can be projected from a Behavior Tree.
- Individual or combinations of components can be projected from the tree and viewed separately.
- Pure BT files (not containing node placement details) can be imported and exported.

3. Model Checking using BC

The system representation produced by integrating separate functional requirements to form an integrated DBT is a problem domain representation defined using the components, states, events and conditions specified by the user. This unique representation allows formal tasks like model and consistency checking. Model checking of the system can be performed directly on the requirements of the system, the place where inconsistency and incompleteness defects first occur. This provides an advantage over model checking a solution-domain design in two ways. First, the cause of the problem is addressed directly rather than via a consequence of it. Second, any changes made to the requirements of the system can be instantly available in the design.

The toolset provides automated means for capturing functional requirements, identifying incompleteness and inconsistency in the requirements and in some cases providing part solutions for identified issues. BC can take as input an XML file describing the structure and details of the Behavior Tree it is to analyse or can operate directly on the model underlying the graphical representation of the tree (used in BTE). BC currently has two main uses:

- Provide a translation from Behavior Tree Notation to Communicating Sequential Processes (CSP) for model checking on deadlock and livelock conditions.
- Use information directly from the tree to generate consistency information and run checks based on this information.

3.1. CSP for FDR

Using a translation process based on the one described in [2], CSP can be generated for a Behavior Tree and model checked using a tool such as FDR. The following shows the CSP generated for the left hand fragment of the microwave oven example (see **Idle 1** in Figure 1) (also [1]). These CSP processes are run in parallel and synchronise on common events.

```
OvenX = X → ovenOpen → OvenOpen
UserX = ovenOpen → userDoorClosed → UserDoorClosed
DoorX = userDoorClosed → doorClosed → DoorClosed
LightX = doorClosed → lightOff → LightOff
ButtonX = doorClosed → buttonEnabled → ButtonEnabled
OvenOpen = lightOff → buttonEnabled → ovenIdle
           → OvenIdle
```

3.2. BT Consistency Checks

Building a system out of concrete components that interact instead of a linked group of objects that exhibit

behaviour provides a significant advantage when checking the system to ensure consistent operation. In Figure 1 an example is shown where two sections of behaviour from the system set the Oven to idle. These two sections establish preconditions in the form of the state each component must be in for the Oven to become idle.

Figure 1. BT for Microwave Oven

Idle 1		Idle 2	
Behaviour establishes			
Door	Closed	Powertube	Off
Light	Off	Light	Off
Button	Enabled		
Established by previous System State			
Powertube	Off	Door	Closed
		Button	Pushed

Two different states are established for the Button component. This shows an inconsistency when defining Oven by its constituent component/state combinations. This inconsistency is caused by missing behaviour required to enable the Button when the Oven times out (Idle 2). The Behavior Tree structure allows other context free consistency checks to be performed.

References

- [1] R. Dromey. From requirements to design: Formalizing the key steps. In A. Cerone and P. Lindsay, editors, *Int. Conf. on Software Engineering and Formal Methods (SEFM 2003)*, pages 2 – 11. IEEE Computer Society, 2003.
- [2] K. Winter. Formalising behaviour trees with CSP. In E. Boiten, J. Derrick, and G. Smith, editors, *Proc. of Int. Conf. on Integrated Formal Methods (IFM'04)*, volume 2999 of *LNCS*, pages 148–167. Springer-Verlag, 2004.