A Metamodel for the Behavior Trees Modelling Technique

Cesar Gonzalez-Perez[a], Brian Henderson-Sellers[a] and Geoff Dromey[b]

[a] Department of Software Engineering
Faculty of Information Technology
University of Technology, Sydney
PO Box 123, Ultimo NSW 2007
Australia

cesargon@it.uts.edu.au · brian@it.uts.edu.au

[b] Software Quality Institute
School of Computing and Information Technology
Griffith University
Nathan Campus QLD 4111
Australia

G.Dromey@griffith.edu.au

Abstract

Behavior Trees have been used as a modelling technique to successfully develop a number of complex software systems. However, the only available formal characterisation of this technique makes heavy use of CSP, which is not very accessible for the object-oriented community where this technique could find valuable usage. This also makes Behavior Trees hard to learn, communicate, use and implement in software tools. This paper offers a comprehensive metamodel that formally describes the main areas of the Behavior Trees technique, thus improving its usability and overall value for the object-oriented community.

Keywords: Behavior Trees. Requirements Engineering. Metamodel.

1 Introduction

The technique of Behavior Trees[1] is based on the premise that software development should construct systems *out of* their requirements rather than satisfying their requirements. This idea, often called "genetic software engineering", together with the Behavior Trees technique itself, has been documented, explained and discussed in a number of papers [2, 3, 5, 6]. This paper is not an introduction to the Behavior Trees technique and therefore will not discuss its appropriateness or details. Rather, this paper will present the result of our research regarding the identification and description of a metamodel that formally represents most of the concepts and relationships used by Behavior Trees. This metamodel can be used to better understand, communicate, analyse, extend and implement the Behavior Trees technique, particularly for use in object-oriented developments, where it potentially has value.

The next section briefly introduces the Behavior Trees technique. Section 3 establishes the scope of the metamodel and describes the major guidelines that were taken into account during its design. Section 4 describes the entities in the metamodel, followed in Section 5 by a general discussion of the utility and application of the technique. Finally, Section 6 presents the conclusions.

2 The Behavior Trees Technique

The Behavior Trees technique is based on the assumptions that systems exhibit components that have attributes and realise and change states and, by doing so, they make other components realise and change states as well. From [2],

> *A Behavior Tree is a formal, tree-like graphical form that represents behaviour 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.*

Some clarifications are needed on the above definition. First of all, each node in a behavior tree corresponds to a component-state, i.e. a particular component in, waiting for or checking for a particular state. For example, a node in a behavior tree could represent a DOOR component in the state "Open", while another node could represent a "LIGHT" in the state "On" (Figure 1).

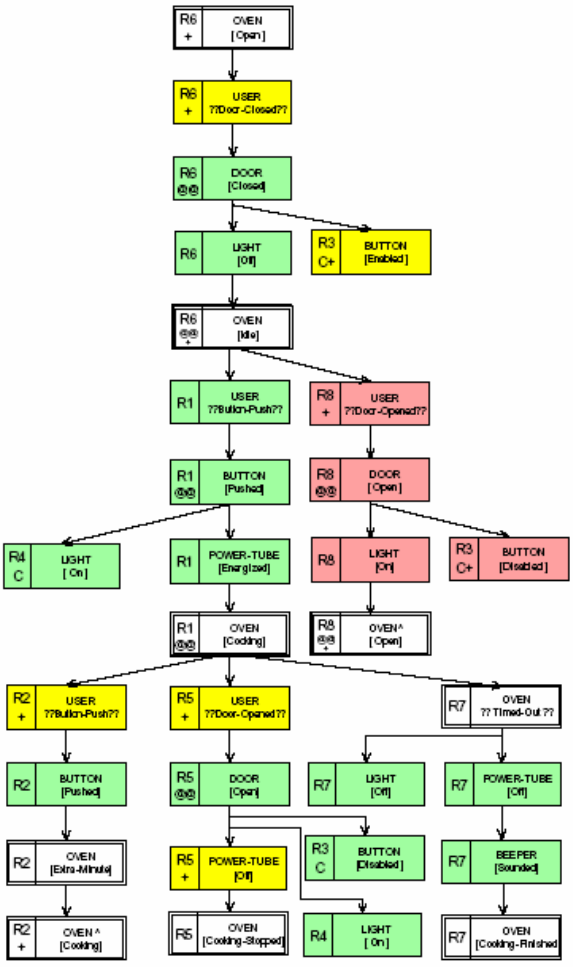

Figure 1. Example behavior tree. Each node references a given component (in all capitals inside the boxes) and a given state (in square brackets under the component's name). (Taken from [6])

[1] The American spelling "Behavior" will be used to refer to the trademarked "Behavior Trees" phrase.

Proceedings of the Third International Conference on Information Technology and Applications (ICITA'05)
0-7695-2316-1/05 $20.00 © 2005 IEEE

Secondly, transitions between nodes represent causal, logical and temporal relationships between component-states. For example, a transition from the open door node into the light on node would mean that the door being open causes the light to be turned on. The full context of this example is shown in Figure 1.

Using the Behavior Trees technique, individual functional requirements are represented as simple behavior trees, which are later combined, one by one, into a composite behavior tree. The final composite behavior tree represents the behaviour that will be exhibited by the final system. We must point out that different nodes in the same or different trees can represent exactly the same component-state; for example, regarding the microwave oven example used by [6], four nodes in the tree shown in Figure 1 are labelled as LIGHT [On], all of them meaning the same: the light in the oven is on. This apparent redundancy could be removed by deleting all LIGHT [On] nodes except one, and using multiple transitions into the remaining node. By doing this, the tree would not look like a tree anymore, but more like a graph, which is not regarded as good practice; behavior trees must be displayed as trees for simplicity. Despite this, their semantics do not correspond to a tree but to a graph. We must note that, from the definition at the beginning of this section, behavior trees are notational entities. We can thus further note that these tree-like notational entities actually depict graph-like conceptual constructs.

3 Metamodel Scope and Design Principles

Although the concepts underpinning the Behavior Trees technique are extremely simple, an enormous range of concerns can be modelled using this technique. To mention but a few, [1] lists state realisation, data input and output, boolean logic, flow of control, counting and component quantification, component qualification, threading, iterating, timing, data structures and relations. The examples and case studies in the literature make use of these to different degrees, some being more mature than others.

To identify the rules underpinning the Behavior Trees approach, we use metamodelling. A metamodel is simply a model of a model; in this instance we create a metamodel of any model than can be possibly created using Behavior Trees such that it captures all the rules and semantics underpinning this particular technique. Since we must delineate the scope of this metamodel, here we have decided that the metamodel should only cover those elements of Behavior Trees that are (a) mature enough to be significantly stable in their definitions and (b) central to the Behavior Trees technique. Thus, the metamodel presented here covers the following modelling concerns: state realisation, data input and output, boolean logic, flow of control and component quantification.

In addition, the metamodel has been designed around the idea that behavior trees are just (partial or total) depictions of an underlying model. Following this premise, the metamodel describes both the underlying model on which behavior trees can be built plus the behavior trees themselves.

4 Metamodel Elements

The classes in the metamodel are described below. The whole metamodel can be seen in Figure 2. In the following subsections, we define each element in the metamodel in turn, listing in tabular form all its attributes and relation-

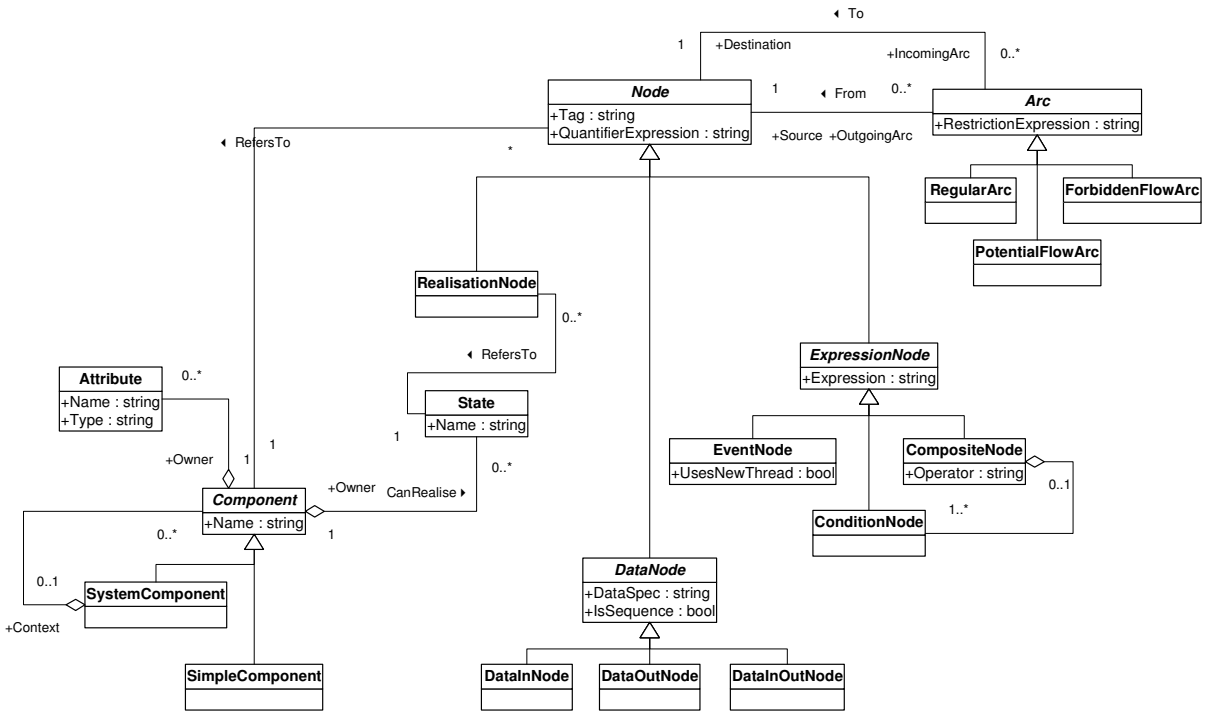

Figure 2. Metamodel for Behavior Trees. UML 1.5 is used to depict the metamodel. White diamonds mean "whole/part relationship", without any further implication about the secondary characteristics of the whole/part relationship.

ships.

4.1 Component

A component is *a formalised abstraction representing a set of entities of interest in the problem domain*. This concept is similar to that of class in the object-oriented paradigm. Components may correspond to real-world entities or mental constructs, and usually reveal themselves through the existence of component instances. For example, in an oven control system, Button may be a component and each of the existing buttons in the oven would be a component instance.

Component is an abstract class, specialised into SystemComponent and SimpleComponent. Its attributes and relationships are given in tabular format below.

Attributes

Name	Type	Description
Name	String	The component name.

Relationships

Name/Role	To Class	Description
Attributes	Attribute	A component may have attributes.
Context	System Component	A component may belong to a system component, which defines its context.
CanRealise	State	A component can realise different states.

4.2 Attribute

An attribute is *a scalar property of a component*. This concept is similar to that of attribute in the object-oriented paradigm. Attributes are often used to describe properties of components that take specific values in component instances. For example, "Colour" could be an attribute of component "CAR".

Attributes

Name	Type	Description
Name	String	The attribute name.
Type	String	The attribute data type: boolean, integer, string, etc.

Relationships

Name/Role	To Class	Description
Owner	Component	An attribute always belongs to a specific component.

4.3 SystemComponent

A system component is *a component representing the whole system being modelled*. System components allow the modeller to deal with the whole system as a component of a bigger system.

SystemComponent is specialised from Component.

Attributes

This class does not have any attributes of its own.

Relationships

Name/Role	To Class	Description
IsComposed Of	Component	A system component is composed of other components.

4.4 SimpleComponent

A simple component is *a component representing an atomic concept in the system being modelled*. Simple components are usually utilised to represent abstractions that are not further decomposable into smaller components.

SimpleComponent is specialised from Component and adds no further attributes or relationships.

4.5 State

A state is *a specific region in the space of possible configurations that a component may exhibit*. This concept is similar to that of state in conventional software engineering. Different states of a component reflect different behavioural settings. For example, states "Open" and "Closed" of a "DOOR" component (Figure 1) reflect different conditions of a door, which in turn dictate how a given door would behave. Different states of a component are mutually exclusive.

Attributes

Name	Type	Description
Name	String	The state name.

Relationships

Name/Role	To Class	Description
Owner	Component	A state always belongs to a component.

4.6 Node

A node is *a formalised abstraction representing a specific behavioural situation of a given component*. Nodes are used to represent a component realising a state, performing an action, waiting for an event to occur, or any other relevant situation. Notice that nodes are not graphical depictions but conceptual constructs. When drawing behavior trees on paper or on screen, the same node can appear as different icons in different places.

Node is an abstract class, specialised into RealisationNode, DataNode and ExpressionNode.

Attributes

Name	Type	Description
Tag	String	A textual identification of the node often utilised to trace the source information that justifies the existence of the node.
Quantifier Expression	String	Expression showing how many and which component in-stances of the associated component are ruled by the node.

Relationships

Name/Role	To Class	Description
RefersTo	Component	A node always refers to a component.

4.7 RealisationNode

A realisation node is *a node representing the fact that a component realises a given state*.

This class is specialised from Node.

Attributes

This class does not have any attributes of its own.

Relationships

Name/Role	To Class	Description
RefersTo	State	A realisation node always refers to a particular state of the associated component.

4.8 DataNode

A data node is *a node representing the fact that a component inputs and/or outputs data from/to other components.* Data nodes allow the modeller to define data flows.

DataNode is specialised from Node, and is an abstract class specialised into DataInNode, DataOutNode and DataInOutNode.

Attributes

Name	Type	Description
DataSpec	String	A specification of the data exchanged.
IsSequence	Boolean	Indicates whether or not the same data specification is exchanged repeatedly in a sequence.

Relationships

This class does not have any relationships of its own.

4.9 DataInNode

A data-in node is *a data node representing the fact that a component inputs data from another component*.

This class is specialised from DataNode and adds no further attributes or relationships.

4.10 DataOutNode

A data-out node is *a data node representing the fact that a component outputs data to another component*.

This class is specialised from DataNode and adds no further attributes or relationships.

4.11 DataInOutNode

A data-in/out node is *a data node representing the fact that a component inputs data from another component and, in turn, outputs the same data to a third component*.

This class is specialised from DataNode and adds no further attributes or relationships.

4.12 ExpressionNode

An expression node is *a node representing the evaluation of an expression plus an associated action on the result*. Expressions may contain references to components, attributes and states.

This class is specialised from Node, and is an abstract class specialised into ConditionNode, EventNode and CompositeNode.

Attributes

Name	Type	Description
Expression	String	The expression being evaluated and acted upon.

Relationships

This class does not have any relationships of its own.

4.13 ConditionNode

A condition node is *an expression node that passes control through its outgoing arcs if its expression evaluates to "true"*. Condition nodes allow modellers to define if-then logic.

This class is specialised from ExpressionNode.

Attributes

This class does not have any attributes of its own.

Relationships

Name/Role	To Class	Description
IsPartOf	Composite Node	A condition node may be part of a composite node.

4.14 EventNode

An event node is *an expression node that passes control through its outgoing arcs when its expression evaluates to "true"*.

This class is specialised from ExpressionNode.

Attributes

Name	Type	Description
UsesNew Thread	Boolean	Indicates whether a new thread of execution is started when the node's condition evaluates to "true".

Relationships

This class does not have any relationships of its own.

4.15 CompositeNode

A composite node is *an expression node that is defined as the logical combination of several condition nodes.* Different boolean operators such as "and", "or" and "xor" can be used.

This class is specialised from ExpressionNode.

Attributes

Name	Type	Description
Operator	String	Specifies the boolean operator to use in order to combine the conditions of the member condition nodes.

Relationships

Name/Role	To Class	Description
Member	Condition Node	A composite node is defined as a combination of condition nodes.

4.16 Arc

An arc is *a formalised abstraction representing a behavioral connection between two nodes*. Arcs are used to represent cause/effect, data flow and temporal dependencies between nodes.

This is an abstract class, specialised into RegularArc, PotentialFlowArc and ForbiddenFlowArc.

Attributes

Name	Type	Description
Restriction Expression	String	Expression that specifies a restriction on the flow of control.

Relationships

Name/Role	To Class	Description
From	Node	A flow arc always starts in a source node.
To	Node	A flow arc always ends in a destination node.

4.17 RegularArc

A regular arc is *an arc representing a simple and direct flow of control between two nodes*.

This class is specialised from Arc and adds no further attributes or relationships.

4.18 PotentialFlowArc

A potential flow arc is *an arc representing a potential flow of control between two nodes*.

This class is specialised from Arc and adds no further attributes or relationships.

4.19 ForbiddenFlowArc

A forbidden flow arc is *an arc representing a prohibition to transfer control*.

This class is specialised from Arc and adds no further attributes or relationships.

5 Discussion

Behavior Trees have been given a formal semantics using weakest preconditions [2] and CSP (Communicating Sequential Processes) [7]. These semantics are useful to support model-checking [4]. However they are not easily accessible to the broader object-oriented community. In that context, metamodels provide an alternative, more easily accessible representation for characterising, clarifying and documenting the semantics of modelling notations. The metamodel for Behavior Trees has highlighted and resulted in the clarification of several semantic issues. The metamodel is also important in assisting us to compare the expressive power of Behavior Trees relative to other notations. For example, Behavior Trees capture, as a subset of their capability, what is normally expressed in state diagrams, activity diagrams and data-flow diagrams. The metamodel makes this clear. It can also be used as a reference when considering extensions to the notation and tool development.

6 Conclusions

This paper has presented a metamodel for the Behavior Trees technique. This metamodel formally describes the core modelling areas of this technique (namely, state realisation, data input and output, boolean logic, flow of control and component quantification) and takes into account the separation between the model of a system (as a network of entities) and particular visual representation, i.e. a behavior tree. We believe that this metamodel will make Behavior Trees more accessible to the object-oriented community with regard to its adoption as modelling technique.

Acknowledgements

The authors wish to thank the Australian Research Council for financial support for this work. This is Contribution number 04/36 of the Centre for Object Technology Applications and Research.

References

1. Dromey, R.G., 2001. *Genetic Software Engineering - Simplifying Design Using Requirements Integration*. In *IEEE Working Conference on Complex and Dynamic Systems Architecture*. Brisbane (Australia), December 2001.
2. Dromey, R.G., 2003. *From Requirements to Design: Formalizing the Key Steps*. In *First International Conference on Software Engineering and Formal Methods*. Brisbane (Australia), 22-27 September 2003. IEEE Computer Society.
3. Glass, R.L., 2004. *Practical Programmer: Is this a Revolutionary Idea, or Not?* Communications of the ACM. **47**(11): p. 23-25.
4. Smith, C., K. Winter, L. Hayes, R.G. Dromey, P. Lindsay, and D. Carrington, 2004. *An Environment for Building a System Out of Its Requirements*. In *19th IEEE International Conference on Automated Software Engineering*. Linz (Austria), 20-24 September 2004.
5. Software Quality Institute at Griffith University, 2004. *Genetic Software Engineering* (website). Accessed on 10 December 2004. http://www.sqi.gu.edu.au/GSE/
6. Wen, L. and R.G. Dromey, 2004. *From Requirements Change to Design Change: A Formal Path*. In *Second International Conference on Software Engineering and Formal Methods*. Beijing, 28-30 September 2004. IEEE Computer Society.
7. Winter, K., 2004. *Formalising Behavior Trees with CSP*. In *International Conference on Integrated Formal Methods 2004*. Canterbury, Kent (UK), 4-7 April 2004. LNCS 2999. Springer-Verlag.