The Journal of Systems and Software 85 (2012) 2652–2664

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Execution of natural language requirements using State Machines synthesised from Behavior Trees

Soon-Kyeong Kim[a,*], Toby Myers[b], Marc-Florian Wendland[c], Peter A. Lindsay[d,a]

[a] Software Systems Research Group, Queensland Research Lab, National ICT, Australia
[b] Griffith University, Nathan, QLD 4111, Australia
[c] Fraunhofer Institut FOKUS, Berlin, Germany
[d] School of ITEE, The University of Queensland, QLD 4072, Australia

ARTICLE INFO

Article history:
Received 18 November 2011
Received in revised form 5 April 2012
Accepted 5 June 2012
Available online 19 June 2012

Keywords:
Requirements
Requirements Validation
Behavior Trees
Behavior Engineering
MDE
Model Transformation
UML State Machine

ABSTRACT

This paper defines a transformation from Behavior Tree models to UML state machines. Behavior Trees are a graphical modelling notation for capturing and formalising dynamic system behaviour described in natural language requirements. But state machines are more widely used in software development, and are required for use with many tools, such as test case generators. Combining the two approaches provides a formal path from natural language requirements to an executable model of the system. This in turn facilitates requirements validation and transition to model-driven software development methods. The approach is demonstrated by defining a mapping from Behavior Trees to UML state machines using the ATLAS Transformation Language (ATL) in the Eclipse Modeling Framework. A security-alarm system case study is used to illustrate the use of Behavior Trees and execution to debug requirements.

1. Introduction

Behavior Trees are a notation for capturing natural-language requirements in a graphical format in a manner which stays close to the structure and terminology of the original requirements, but with a formal semantics. Geoff Dromey described a method – which he called Behavior Engineering (BE) (Dromey, 2003, 2006) – for developing a Behavior Tree (BT) model from requirements systematically, in such a way that issues with consistency and completeness are revealed and resolved as the tree is built. The resulting tree expresses all the scenarios and use cases that are implied by the requirements in a single coherent model.

Evidence from industry use (Boston, 2008; Powell, 2007) has demonstrated that requirements quality can be significantly improved using this approach, and that the resulting models are much easier for non-experts to understand. This in turn leads to improved requirements understanding early in the system and software development process. The BE method supports transformation of high-level BTs into more detailed ones in which design decisions are embodied, and there are tools for generating code from sufficiently detailed models, but these aspects of the method are not as widely used. At some point it becomes preferable to switch to more traditional development notations and methods, such as UML and MDE (Schmidt, 2006).

In this paper we define a transformation from basic BT models to UML state machine (SM) models. The full BT notation supports a rich variety of relations, capturing non-functional aspects (the what, why, when, where and how of requirements) as extra annotations to nodes in the Behavior Tree. The stripped-down basic BT notation captures functionality and behaviour, such as control and data flow (Lindsay, 2010) — the “logic” of the requirements. Ensuring the consistency and completeness of this logic can be one of the hardest things for a system developer to get right, and one of the most expensive things to fix if it is wrong (Glass, 2004).

We contend that BT modelling combined with formal transformation to SM models is a highly effective means for going from natural-language system requirements to Model Driven Engineering (MDE) in UML. Conversely, the ability to execute SM models enables dynamic aspects of BT models to debugged, leading to improved system specifications. We illustrate the approach on a well known case study – the security alarm system from Prowell et al. (1999).

* Corresponding author.
E-mail addresses: Soon-Kyeong.Kim@nicta.com.au (S.-K. Kim), Toby.Myers@griffithuni.edu.au (T. Myers), marc-florian.wendland@fokus.fraunhofer.de (M.-F. Wendland), p.lindsay@uq.edu.au (P.A. Lindsay).

0164-1212/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2012.06.013

The paper is structured as follows: Section 2 describes related work, including potential applications of the translation. Section 3 describes the basic BT notation and illustrates it on the case study. Section 4 describes the BT-to-SM transformation rules and illustrates them on a set of small examples that capture generic BT structures. Section 5 illustrates how execution of the SM model reveals issues with formulation of requirements in the case study as an application of the transformation. Section 6 summarises our approach and discusses future work. Prior familiarity with UML state machines is assumed (OMG, 2011a).

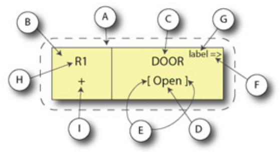
2. Related work

Other papers have presented translational semantics for the BT notation, but ours is the first full translation to state machines. (Grunske et al., 2008) define a graph grammar-based transformation from BT models to Action Systems, as used by the SAL model checking tool. The SAL translation has been used to provide support for Failure Modes and Effects Analysis (Grunske et al., 2011) and Cut Set Analysis (Lindsay et al., 2011) using the BT method. Colvin et al. (2008) extend the BT notation with timing requirements, with a translation to the UPPAAL model checker. They also extend the BT notation with probability and timing, with a translation to the PRISM stochastic model checker in Colvin et al. (2007). Colvin and Hayes (2010) provide a full formal static semantics for the BT notation in an extension of the CSP notation.

Other approaches have been developed for developing SM models from requirements-capture notations, including UML Sequence Diagrams (Whittle and Schumann, 2000), Message Sequence Charts (Kruger, 2000; Uchitel et al., 2003) and Live Sequence Charts (Harel et al., 2005; Meng et al., 2011). As noted in Whittle and Jayaraman (2006), however relationships between scenarios and/or use cases are not defined explicitly, which makes it difficult to generate a complete executable SM. Whittle and Jayaraman (2006) address this problem by explicitly defining relationships between scenarios and use cases using activity diagrams and interaction diagrams, and then generate a SM from these diagrams. By contrast BTs integrate use cases and scenarios into a single model.

Animators and code generators have been developed for BTs, but they are generally still embryonic. By contrast, many simulation tools support UML state machines, to a greater or lesser extent, with varying interpretations of SM semantics. We use the SHIRE tool here (Höfig et al., 2011), with an extension to enable visualisation of execution sequences. Crane and Dingel (2007) present syntactic and semantic differences between the UML SM and its variants (e.g. classical and Rhapsody statecharts) and discuss the implications for supporting tools. OMG has released a new specification notation fUML (OMG, 2011b) within the UML family, that describes a fundamental subset for an executable representation of UML. However, fUML is based solely on activities and as yet does not support the state concept which is vital in mapping BTs to UML, and there is no mature execution tool for fUML as yet.

There are many approaches to model-based testing based on UML state machines. These approaches include generation and specification of test cases (Chevalley and Thévenod-Fosse, 2001; Kim et al., 1999). BT models can be developed directly from high-level functional specifications of systems, so are potentially a strong basis for derivation of system-level test cases, via state machines and SM-based testing tools. The latter include commercial tools (Conformiq, 2011; TestCast MBT, 2011; MBTsuite, 2011) and non-commercial tools (Alin et al., 2010; SpecExplorer, 2011). This topic is the subject on ongoing research.



(A) Behavior Tree node; (B) tag; (C) component name; (D) behaviour note; (E) behaviour type; (F) operator; (G) label; (H) traceability link; (I) traceability status

Fig. 1. Elements of a BT node.

3. Introduction to Behavior Trees

This section describes the BT notation and illustrates it on the security alarm system case study.

3.1. Behavior Tree notation

The Behavior Tree notation is part of a whole methodology of systems and software engineering originally developed by Geoff Dromey called Behavior Engineering (Dromey, 2003). Behavior Trees capture the dynamic behaviour of a system of components in a graphical form. The nodes in the tree describe how components change state in response to flow of data and/or control in multiple parallel threads. Behavior Tree models are developed directly from natural-language system functional requirements by a stepwise process of first translating the behaviour expressed by individual requirements into a partial tree and then integrating the fragments together to form a complete tree. Nodes in the tree are tagged with identifiers of the requirements that gave rise to them, for traceability.

Fig. 1 displays the full contents of a BT node. Each node (A) is associated with a component (C) and has a 'behaviour' (D and E), which is described in more detail below. It can also have an operator (F) and/or label (G), which describe flow of control. The tag (B) has two parts: a link (H) which traces the node back to the requirements that gave rise to it; and a traceability status (I), which the modeller uses to indicate how well the requirement captures the behaviour (not used in this paper).

The different ways nodes can be connected in a Behavior Tree are shown in Fig. 2, together with the different node types and operators. Control flows down branches according to the rules sketched out below: see Behavior Tree Group (2007) for more details. Control flow forks into separate threads when a parallel branching node is reached. For alternative branching, only one thread gets executed, chosen nondeterministically. Although Fig. 2(l–m) show just two branches below a branching node, more than two are also allowed. For the purposes of this paper a fully interleaved control-flow semantics applies. (Sometimes internal actions are given priority over external I/O, but that will not be covered here.) The exception is when nodes are joined by atomic composition (cf. Fig. 2(o)): when flow reaches an atomic grouping, other threads block until all of the nodes in the group have been executed.

The meaning of the different node types is as follows:

- State realisation node `Component1[`*State*1`]` indicates that `Component1` is in *State*1.
- Selection node `Component1?`*Condition*1 ? is similar to an *if* statement: if `Component1` satisfies *Condition*1 when control reaches this node, flow of control continues along the branch; otherwise it terminates. Typically selection nodes appear immediately under an alternative branching node; sometimes a `Component1?`*ELSE*`?` is used, to cover the case where all the other selections fail.
- Guard node `Component1???`*Condition*1`???` is similar to a *wait* statement: the *Condition*1 is continuously re-evaluated and flow