Execution of Natural Language Requirements using State Machines Synthesised from Behavior Trees

Soon-Kyeong Kim\textsuperscript{a,}*\textsuperscript{a}, Toby Myers\textsuperscript{b}, Marc-Florian Wendland\textsuperscript{c}, Peter A. Lindsay\textsuperscript{d,a}

\textsuperscript{a}Software Systems Research Group, Queensland Research Lab, National ICT Australia
\textsuperscript{b}Griffith University, Nathan, Qld 4111, Australia
\textsuperscript{c}Fraunhofer Institut FOKUS, Berlin, Germany
\textsuperscript{d}School of ITEE, The University of Queensland, Qld 4072, Australia

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 behavior described in natural language requirements. But state machines are more widely used in software development, and have a much broader range of tool support including simulation environments and animators. 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 improvement. 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, and the SHIRE simulation tool for execution.

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

*Principal corresponding author

Email addresses: soon@itee.uq.edu.au (Soon-Kyeong Kim), Toby.Myers@griffithuni.edu.au (Toby Myers), marc-florian.wendland@fokus.fraunhofer.de (Marc-Florian Wendland), p.lindsay@uq.edu.au (Peter A. Lindsay)
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) [12, 14] – 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 [7, 31] 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 BT models into lower-level, more detailed models 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 [33].

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 finish this list 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 [25] — 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 [17].

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 [32].

The paper is structured as follows. Section 2 describes the basic BT notation and illustrates it on the case study. Section 3 describes the BT-
to-SM transformation rules and illustrates them on small examples. The rules were implemented in the ATLAS Transformation Language (ATL) [1] using the Eclipse Modeling Framework (EMF) [15]. The section concludes with the full translation of the case study into an SM model. Section 4 illustrates how execution of the SM model reveals issues with formulation of requirements in the case study. The BT model is improved in a series of steps, using SM execution to debug the logic. SHIRE [20] was used to execute the SM models, together with a purpose-built visualiser. The section concludes with an improved set of requirements for the security alarm system. Finally Section 5 describes related work and Section 6 summarises our approach and discusses future work. Prior familiarity with UML state machines is assumed [29].

2. Introduction to Behavior Trees

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

2.1. Behavior Tree notation

The Behavior Tree notation is part of a whole methodology of system and software development developed by Geoff Dromey called Behavior Engineering [12]. 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-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

![Figure 1: Elements of a BT node](image)

![Figure 2: Summary of the core elements of the BT notation](image)
down branches according to the rules sketched out below: see [3] 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 won’t be cov-
ered 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[State1]’ indicates that Component1
  is in State1.

- Selection node ‘Component1?Condition1?’ is similar to an if statement:
  if Component1 satisfies Condition1 when control reaches this node, flow
  of control continues along the branch; otherwise it terminates. Typi-
  cally selection nodes appear immediately under an alternative branch-
  ing node; sometimes a ‘Component1?ELSE?’ is used, to cover the case
  where all the other selections fail.

- Guard node ‘Component1???Condition1???’ is similar to a wait state-
  ment: the Condition1 is continuously re-evaluated and flow of control
  is blocked at this node until (if ever) Condition1 becomes true.

- Internal output node ‘Component1<Message1>’ indicates that Com-
  ponent1 has broadcast Message1 on an internal channel, and internal
  input node ‘Component1<Message1>’ indicates that Component1 has
  received Message1 on some internal communication channel. Com-
  munication channel details are not modelled in BT notation: they can
  represent any kind of data flow. Communication is asynchronous. Out-
  put can be sent at any time, but flow of control is blocked at an input
  node until the corresponding output node has been reached, either im-
  mediately before the node if it is on the same branch, or in some other
  thread.

- External input/output is similar to internal I/O, except that external
  communications channels (/data interfaces) are involved.
The full BT notation also includes event nodes ‘Component??Event??’ but for this paper’s purposes these are indistinguishable from external input nodes so will be omitted.

BT operators (Fig. 2(h-k)) are used to indicate control flow beyond simple sequential execution and branching. In what follows, ‘origin node’ refers to the node within which the operator appears. The ‘target node’ is a node elsewhere in the tree which has the same component name and ‘behaviour’ as the origin node. Exactly which node is the target node depends on the operator involved, as explained below. For the BT model to be well-formed, the target node needs to be unambiguous; see [10] for the full static semantics of the BT notation. The meaning of the different operators is as follows:

- The reference operator ‘=>’ causes control to jump to the target node, which can appear anywhere in tree except at leaf node; reference can only be used at leaf nodes. In essence, reference acts as a macro, equivalent to creating in-place below the origin node a complete copy of the subtree from below the target node.

- The branch kill operator ‘– –’ terminates all threads that started at or below the target node. In this case the target node must be a sibling of one of the origin node’s predecessors further up the tree.

- The reversion operator ‘ˆ’ causes control to jump back to the target node, which must be one of the origin node’s predecessors further up the tree. Reversion causes termination of all threads that started at or below the target node.

- The synchronisation operator ‘==’ causes control flow to block until control reaches the target node. In fact, synchronisation can be applied to multiple threads. Both the origin node and the target nodes get labelled with ‘==’.

If flow of control reaches a leaf node that does not have an operator, the corresponding thread executes the node and terminates.

BTs are defined by a formal semantics defined in CSP\(\sigma\), an extension of CSP that can capture state based information [10].

2.2. Case Study: Security Alarm System and BT modelling

This section introduces the case study that is used as a running example in this paper – the Security Alarm System. Table 1 shows the seven natural-
language requirements that constitute the system functional specification in [32].

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>The security alarm has a detector that sends a trip signal when motion is detected.</td>
</tr>
<tr>
<td>R2</td>
<td>The security alarm is activated by pressing the Set button.</td>
</tr>
<tr>
<td>R3</td>
<td>The set button is illuminated when the security alarm is set.</td>
</tr>
<tr>
<td>R4</td>
<td>If a trip signal occurs while the security alarm is set, a high-pitched tone (alarm) is emitted.</td>
</tr>
<tr>
<td>R5</td>
<td>A three-digit code must be entered to turn off the alarm tone.</td>
</tr>
<tr>
<td>R6</td>
<td>Correct entry of the code deactivates the security alarm.</td>
</tr>
<tr>
<td>R7</td>
<td>If a mistake is made when entering the code, the user must press the Clear button before the code can be re-entered.</td>
</tr>
</tbody>
</table>

Table 1: Functional requirements for the Security Alarm System

A BT model developed by systematic formulation and integration of the seven requirements is shown in Figure 3. Nodes are labelled with the requirement(s) that gave rise to them, if any. For example, the first three nodes capture the behavior of activating the security alarm system. Pressing the Set button is modelled as an external input. This results in the system being activated. (The requirements also use the word ‘set’ but we stick with ‘activated’ for consistency in the model.) The rest of the BT model is fairly self-explanatory. External events such as detecting motion, entering a three-digit code, and pressing the Clear button are modelled as external inputs. Passing of the trip signal from detector to the control system is modelled using internal I/O. This BT model is used for validating and improving the requirements in Section 4 below.

3. Transforming Behavior Trees to UML State Machines

This section describes our transformation rules for mapping BTs to UML state machines, and the transformation environment used in our work.

3.1. Transformation Environment

Figure 4 shows the transformation environment used. The mapping from BTs to UML SMs is based on metamodels of the two languages and implemented using a model-to-model (M2M) transformation defined in the ATLAS...
Figure 3: Initial BT model of the Security Alarm System
Transformation Language (ATL) [1] within the Eclipse Modeling Framework (EMF). Figure 5 shows the BT metamodel used in the EMF-based TextBE tool [28]. The standard OMG metamodel was used for UML [29], with an EMF-based implementation [35]. SHIRE [20] was used to execute the generated SM models. Due to the complexity of some of transformation rules, not all of them were implemented fully in ATL. probably need to say a bit more about this last point.

Figure 4: Transformation environment

3.2. The Transformation Rules

The transformation maps a BT into a single monolithic state machine. Naming conventions and a variable associated with each component are used to preserve component behaviour in the SM. The transformation supports all the core elements of the BT notation shown in Figure 2. In what follows the transformation rules for each BT construct are explained and illustrated using fragments of BT models, mostly taken from the case study (Figure 3). Familiarity with SM concepts is assumed, such as entry actions, completion events, signals, and so on: see [29] for details.

3.2.1. State realisation nodes

A BT state realisation node maps to an SM state, although different instances of a node typically are given different names in the SM, for reasons explained below. To make the mapping easier to understand, a naming convention is used whereby the SM state’s name is composed of the BT component’s name and (BT) state. A UML attribute is introduced to track
the component’s state: see Figure 6 for an example. (Translation of the external input node is explained in the next section.) The attribute gets set to the appropriate value using an entry action in the SM. This attribute will be de-referenced in SM transitions corresponding to selection and guard nodes, for example.

Duplicate BT state realisation nodes are distinguished in the SM by appending a unique numeral to the name. This is necessary because BT models describe flow of control, not simply state changes: the position of the node in the tree describes the system behaviours involving the component, which can be very different. For example, in the BT fragment shown in Figure 6(a), the first and last nodes are identical but no further transition is possible from the last node.

3.2.2. Input/output nodes and parallel branching

Parallel BT branching translates to composite (parallel) substates in the SM model. BT input nodes translate to SM transitions with triggers, and BT output nodes translate to SM transitions with effects. The message in the BT node is translated to a signal event in both cases. (If the message is parameterised with a data value, then an effect is also added to the transi-
tion that assigns the value to an appropriate attribute.) In our SHIRE-based prototype, code `comms.raise("xxx")` is used to generate the signal event corresponding to the BT output node with message `xxx`. Dummy states are added to the SM model to allow for intervening transitions elsewhere in the BT model: see Figure 7 for an example.

Internal and external messages are not distinguished in our translation since the SM model is not concerned with system boundary issues. (In some treatments of BT semantics, internal transitions get priority over external transitions but such matters will not be treated here.)

3.2.3. Selection nodes, guard nodes and alternative branching

As noted above, BT selection nodes correspond to if statements, with termination if their condition is not met. To mimic this in the SM model one transition is introduced for each selection node that leads to (the translation of) the tree below the node and another transition is also added leading to a final state, corresponding to termination of the thread. The SM transitions are each labelled with a guard condition of its corresponding BT selection node. The guard condition of the additional transition leading to a final state is the combination of the negation of all the other guards of the selection nodes.

Typically BT selection nodes occur at the head of alternative branches,
sometimes using an "ELSE? node to capture the “otherwise” case (where none of the conditions in the other branches are true): see Figure 8(a) for an example. In this case, instead of being a final state, a transition is created for (the translation of) the tree below the ELSE node. Figure 8 shows an example.

A BT guard node is used to suspend behaviour until some parallel thread results in the node’s component realising the named state. To mimic this in the SM model we add a dummy state corresponding to suspended behaviour, and two transitions. The first transition targets (the translation of) the behaviour below the BT node and has an SM guard checking whether the component is in the desired state (see section 3.2.1 above for details of the UML attribute involved). The second transition loops back to the dummy state and has a guard which is the negation of the one just explained; this enables continuous re-evaluation of the component state.

3.2.4. Reversion and reference operators

Recall that the BT reversion operator causes control to flow back to the node with the same details above the current node and kills any threads that were created below that node. To mimic this in the SM model is relatively straightforward: We create a transition from the current SM state back to the appropriate state, which may be one or more levels outside the current state.
This transition results in the composite states corresponding to the other threads (if any) being killed, as desired. Figure 9 shows a simple example.

Recall that the BT reference operator acts simply like a jump in control flow. In many cases it thus simply gives rise to a transition from the current SM state to the appropriate target state. Figure 10 shows an example. The difficulty arises if composite states intervene since, unlike the reversion
operator, the reference operator does not kill other threads. In these cases we instead treat reference as a kind of ‘macro’ and create a complete copy of SM corresponding to the tree below the target node.

3.2.5. Branch kill and synchronisation operators

Translation of branch kill nodes is more tricky since it involves going back up the tree to the node where the to-be-killed thread branches off, and introducing some more structure into the SM model from this point onwards. An attribute ‘inState’ is added to the SM which keeps track of all threads that might need to be killed (as identified by kill nodes in the BT model). At each point in the SM model where such a thread gets started we insert a composite state composed of two substates. The first of the two sub-states corresponds to that part of the SM which results from translation of the corresponding BT subtree. The other sub-state handles killing of the thread by having a transition to a final state at the outer level, which terminates the first sub-state when the time comes. This second sub-state continuously checks to see if the thread has been added to the inState attribute. Finally, the branch kill node itself gets translated to a state with an entry action which adds the thread to inState. Figure 11 shows an example.

Translation of synchronisation nodes is also handled using the ‘inState’ attribute which keeps track of all instances of synchronisation nodes to check whether they have reached the synchronisation point. A dummy state is
inserted before (the translation of) the origin synchronisation node with two outgoing transitions. Target synchronisation nodes are translated to a dummy state with two outgoing transitions. The incoming transition to all the dummy states has an effect that removes any identifiers from a previous synchronisation. The incoming transition to the dummy states corresponding to the translation of the target nodes has an effect that adds a unique synchronisation identifier for that node to \textit{inState}. The outgoing transitions of the dummy state inserted for the origin node have a guard which checks whether all instances of the target nodes have been added to \textit{inState}. If it does, then the transformation of the synchronisation node is executed. Then another identifier is added to \textit{inState} that indicates the synchronisation has occurred. The outgoing transitions of the dummy state of all the target nodes have guards that continuously check for the synchronisation completion identifier (set by the origin node). When included in \textit{inState}, the flow of control continues. Figure 12 shows an example.

3.2.6. Atomic composition

The final BT construct to be explained is atomic composition. The translation goal is straightforward enough: we need to block all other SM transitions until the transitions for the atomic nodes have taken effect. This is achieved using a kind of semaphore mechanism similar to that used for synchronisation. An attribute is introduced for each atomic group. When control reaches such a group, the attribute gets set to True until (BT) control
flow leaves the group. Dummy states are inserted between transitions in all other sections of the SM model, with self-transitions when the attribute is True and progress to the next state when False. Figure 13 shows an example.

The SM model in Figure 14 was generated by applying the transformation rules to the BT model in Figure 3. This is used for validation in the following section.
4. Requirements analysis and improvement by SM execution

This section illustrates how execution of the SM model can be used to improve understanding and formulation of the case study requirements from Section 2.2. We start with the SM model in Fig. 14, which resulted from transformation of the BT model of the requirements in Fig. 3. At each step we use the SHIRE simulation environment to explore scenarios, to check that the system behaves as desired. (We extended SHIRE with graphical output, to better visualise simulation results.) When we encounter unexpected or undesired behaviour we modify the BT model to correct the behaviour and rerun the scenario until the desired behaviour is attained. We then rewrite the requirements to reflect our improved understanding.

The rest of this section describes four stepwise modifications of the requirements resulting from the above approach. Each step is described in four parts: validation goal (the expected behaviour of the system); simulation results (one or more scenarios where execution either failed or gave unexpected behaviour); correction (modification of the BT model and successful rerun of the scenario); and rewording of the offending requirements.
4.1. Button illumination

*Validation goal:* SET_BUTTON should be illuminated when the security alarm is activated, so users can see that the system is active.

*Simulation results:* After the user press-button event it was possible to activate the system without the button actually getting illuminated. Figure 15(b) shows one such scenario. (In fact closer inspection of the SM model in Fig. 14 reveals that many SM transitions can occur in the second sub-state of the composite state without the button becoming illuminated. But SHIRE only executes the left/top-most sub-state.)

*Correction:* Fig. 15(a) shows the offending part of the BT model. The Set button should be illuminated at the same time as the rest of the system is activated, rather than as a parallel behaviour. Fig. 15(c) shows the reworked part of the BT model, in which the component state changes are composed atomically. The scenario which demonstrated unexpected behaviour, whereby the system is activated before the button illuminated, now fails as expected. Fig. 15(d) shows a successful scenario, in which the light illuminates and the system gets activated.

*Reworded requirement:* The following revision of R3 reflects our improved understanding: “The Set button is illuminated whenever the security alarm is activated.”

4.2. Deactivating the system

*Validation goal:* Closer inspection of the SM model reveals that if the button illuminates then it remains illuminated thereafter. But the desired behaviour is that the button should cease to be illuminated when the system is deactivated.

*Simulation results:* We ran a scenario in which the system got deactivated by the user entering a correct code. But at that point the system deadlocked, with the button still illuminated. Here we treat the deactivation issue first and then return to the button illumination issue in a separate experiment below.
Figure 15: Experiment 1: Button illumination behaviour
Correction: Fig. 16(b) shows the last steps of the run that deadlocked and Fig. 16(a) shows the corresponding part of the BT model. There should be a reversion back to the deactivated state from the final BT node, so the system can keep operating. Fig. 16(c) shows the corrected BT model and Fig. 16(d) confirms the deadlock is broken.

Reworded requirement: R6 should be clarified to say “Correct entry of the code deactivates the security alarm and returns the system to its initial state.”

Figure 16: Experiment 2: System deactivation behaviour

(a) Initial BT model (excerpt)

(b) Example simulator run

(c) Improved BT model (excerpt)

(d) New simulator run
4.3. De-illuminating the button

Validation goal: The button should de-illuminate once the system is deactivated, so the user can see that the system is no longer active.

Simulation results: In the previous experiment, the light remained illuminated after the system was deactivated (see Fig. 16(d)).

Correction: The button needs to get de-illuminated when the system gets deactivated. The best way to achieve this is to insert a node before the reversion node, using atomic composition. Fig. 17(a) shows the result. The run in Fig. 17(b) confirms the correction.

Reworded requirement: Add a clarifying requirement: “R8. The Set button is de-illuminated whenever the security alarm is deactivated.”

4.4. Deactivating the system (II)

Validation goal: The user should be able to deactivate the system without needing to trip the alarm.
Simulation results: Executing the current SM model it is apparent that the only way to reach the system deactivated state is to first detect motion (cf. Fig. 18(b)). Yet clearly deactivation needs to be possible independent of whether motion is detected or not.

Correction: We fixed this issue by inserting a new alternative branch under the SECURITY_ALARM[Activated] node, with a reference to the code-entry subtree. Fig. 18(c) shows the revised part of the BT model. Fig. 18(d) confirms the user can now deactivate the system by entering a code without motion being detected.

Reworded requirement: R5 should be clarified to say “A three-digit code must be entered to deactivate the system.” and a new requirement added: “R9. To turn off the alarm tone, the user must deactivate the system by entering a correct code.”

Figure 18: Experiment 4: System deactivation behaviour revisited
4.5. Improved requirements and models

Fig. 19 shows the BT model with the corrections made. Fig. 20 shows the resulting SM model. The simulation result in Fig. 21 shows two successful scenarios of the system behavior where the user can deactivate the system either with motion being detected (transitions 4 to 10) or by entering correct code (transitions 13 to 16).

The final set of requirements is given in Table 2 below. Three requirements (R3, R5 and R6) are reworded and two clarifying requirements (R8 and R9) are newly added. ‘Set’ has been changed to ‘activated’ for consistency.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>The security alarm has a detector that sends a trip signal when motion is detected.</td>
</tr>
<tr>
<td>R2</td>
<td>The security alarm is activated by pressing the Set button.</td>
</tr>
<tr>
<td>R3</td>
<td>The Set button is illuminated whenever the security alarm is activated.</td>
</tr>
<tr>
<td>R4</td>
<td>If a trip signal occurs while the security alarm is activated, a high-pitched tone (alarm) is emitted.</td>
</tr>
<tr>
<td>R5</td>
<td>A three-digit code must be entered to deactivate the system.</td>
</tr>
<tr>
<td>R6</td>
<td>Correct entry of the code deactivates the security alarm and returns the system to its initial state.</td>
</tr>
<tr>
<td>R7</td>
<td>If a mistake is made when entering the code, the user must press the Clear button before the code can be re-entered.</td>
</tr>
<tr>
<td>R8</td>
<td>The Set button is de-illuminated whenever the security alarm is deactivated.</td>
</tr>
<tr>
<td>R9</td>
<td>To turn off the alarm tone, the user must deactivate the system by entering a correct code.</td>
</tr>
</tbody>
</table>

Table 2: Reworded requirements for the Security Alarm System

5. Related Work

We are not aware of any work related to transforming entire BTs to SMs. Previous work has briefly outlined an approach to map BT state realisations and reversion to SMs, though it does not address other parts of the language [5]. They use the BT notation to visualise part of a SM in a tree-structure.
Figure 19: Revised BT model of the Security Alarm System
In this section, we discuss other work that considers mappings BT or SM to other modelling techniques or vice-versa for different reasons.

Regarding BT, several authors present a mapping of BT to a formal modelling technique mainly to check BT models using analysis tools available for the formal technique. For example, Grunske et al [18] define a transformation from BTs to the input language of the SAL model checking tool using graph grammars. Linear Temporal Logic (LTL) is used for formalising the properties to be checked. The model-checking capability is exploited for Failure Modes and Effects Analysis (FMEA), for example to check that sufficient functionality is incorporated into system design to ensure that single component failures do not lead to critical system failures [30]. [9] extends the BT notation to capture timing requirements more formally, with a translation to the UPPAAL model checker in order to check timing properties. [8] extends the BT notation and semantics with probability and timing for specifying some system performance aspects, and translates to the PRISM stochastic model checker.

Regarding the UML SM notation, several approaches generate a SM from scenario-based models: for example, from UML Sequence Diagrams [37], from Message Sequence Charts [21, 34] and from Live Sequence Charts [19, 27]. The motivation for this work is similar to ours in supporting the transition from requirements to early analysis and design models, and supporting automatic simulation and validation of scenarios by executing the SMs. As pointed out in [36], however, relationships between scenarios and/or use cases

---

**Figure 20: SM Model from the revised BT Model in Fig. [19]**

---
Figure 21: Example simulation run for the SM model for Fig. 20
are not defined explicitly, which makes it difficult to generate a complete SM that is executable. Whittle and Jayaraman [36] 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. BTs on the other hand integrate use cases and scenarios into a single model.

Several works simulate SMs using a model checker such as ASM [6], SPIN [23, 24, 22], SMV [16], and Petri-nets [4]. One reason that the model-checking of SMs is popular is that they can be modelled quite naturally using Abstract State Machines, which in turn are convenient to model check. Work in this area can be applied to simulate the transformed SM from a BT model.

There also exist many simulation tools that claim to support the UML SM including the tool SHIRE used in our work. Michelle et al [11] present syntactic and semantic differences between the UML SM and its variants (e.g. Classical and Rhapsody statecharts), and discuss their implications in supporting tools.

6. Conclusion and Future Work

In summary, the main contribution of this paper is to provide a detailed transformation of models in the BT notation to UML state machines (SMs). The BT notation was developed to improve the modelling and analysis of system requirements expressed in natural language, staying as close as possible to the original natural-language description, to better preserve the intent of the original requirements [13]. We demonstrated how transformation of BT models to SMs and use of SM execution tools enables validation and improvement of requirements. This idea can be taken further, to provide a rigorous path from natural language requirements specifications to Model Driven Engineering. The transformation also provides a formalisation of the semantics of dynamic behaviour represented by BT notation, alternative to the CSP$\sigma$ formalisation given in [10].

The BE methodology also supports the systematic development of detailed designs from requirements BTs [12]. Our transformation works equally well for Requirements BTs and Design BTs: the point at which one switches from the BE methodology to Model Driven Engineering in UML is the developer’s decision. BTs provide a coherent and complete framework for expressing scenarios and use cases which is currently lacking from UML.
The approach was illustrated on the well-known security alarm system example [32], using the TextBE implementation of BTs and the SHIRE implementation of state machines within the Eclipse Modeling Framework. We were able to use the SHIRE model to animate behaviour of the security alarm system and validate the requirements and the BT model. We have applied our work to several other case studies including a complex system, Automatic Train Station [2] and have validated the system via simulation. In future work we plan to exploit SM tool capabilities to extend the range of software analysis and development techniques that can operate on requirements specifications, such as automated generation of test cases and automated analysis by model checking. We also plan to apply our work to industrial applications such as Air Traffic Management System.

Acknowledgements

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centres of Excellence program. This research was also supported under Australian Research Council Linkage Projects funding scheme (project number LP0989363).

References


ASMs are used to define the dynamic semantics of UML state machines, which integrate statecharts with the UML object model.


