From Requirements to Embedded Software - Formalising the Key Steps

Toby Myers and R. Geoff Dromey
School of Information and Computing Technology, Griffith University
Nathan, Brisbane, Qld, 4111, AUSTRALIA
Toby.Myers@student.griffith.edu.au, G.Dromey@griffith.edu.au

Abstract

Failure of a design to satisfy a system’s requirements can result in schedule and cost overruns. When using current approaches, ensuring requirements are satisfied is often delayed until late in the development process during a cycle of testing and debugging. This paper introduces a more rigorous approach to design using Behavior Engineering, which has previously been applied primarily to requirements analysis and specification development. To support design with Behavior Engineering we introduce the embedded Behavior Runtime Environment, a virtual machine created to execute a Behavior Engineering design on an embedded system. The result is a model-driven development approach that can create embedded system software that satisfies its requirements, as a result of applying the development process.

1. Introduction

Software development for embedded systems is often a complex task. Complexity arises from the need to create software at a low-level of abstraction that also interacts closely with hardware. The environment in which embedded systems are deployed also requires strong emphasis to be placed on the dependability and real-time constraints of the software. Furthermore, as embedded systems become ubiquitous they are being used to perform tasks of ever-increasing complexity. This is evident in the 2008 Embedded Market Survey, where respondents listed their three biggest concerns during embedded systems development as meeting schedules; the debugging process; and increased lines and complexity of code. Of the same respondents, 59% also indicated that they were not using a modelling technique in their current embedded projects [1].

One potential reason for the poor adoption of modelling techniques is the gaps that exist when transforming a specification described in natural language requirements to a design using current modelling techniques. Traditionally, the gaps in the path from requirements through to a design are overcome by intuition and iteration. However for embedded systems, these gaps are more evident due to the need to design on a lower-level of abstraction. This often results in embedded systems that do not satisfy their requirements, necessitating extra time to be spent on testing and debugging.

This paper describes an approach to bridge these gaps by using Behavior Engineering (BE) to progress from a system’s requirements to software for embedded systems. BE is an integrated approach to system’s development that supports the engineering of large-scale, dependable, software intensive systems at both the systems and software engineering level. BE works by creating a design out of its requirements, as opposed to using the requirements as a guide to assess the suitability of a design or implementation. It advocates a correct-by-construction approach that can avoid the potential schedule overruns and dependability problems inherent in construct-by-correction approaches.

The focus of BE has previously primarily been on requirements analysis and the location and correction of defects in specifications. When using BE to develop designs for embedded systems, it was found that the BE approach to design required refinement. This is because previous uses of BE muddled the creation of a specification with the task of adding real design decisions beyond the specification. This paper addresses this issue by separating the task of specification and design into separate stages, and clarifying types of design decisions that can be applied to refine a BE specification into a design. In order for the created BE design to be executable, we developed the embedded Behavior Runtime Environment, a virtual machine with a small footprint to make it suited for embedded systems.

The remainder of the paper is laid out as follows. Section 2 provides an introduction to BE. Section 3 describes the embedded Behavior Runtime Environment that executes BE models. Section 4 shows an example embedded system built using BE. Section 5 discusses related research for developing software for embedded systems.

2. An Introduction to Behavior Engineering

BE has recently become recognized as a useful approach to the requirements analysis of large-scale industry projects, as it can detect defects at a rate approximately two to three times higher than conventional techniques [2]. In order to manage these large-scale systems, BE has tool support developed in the Eclipse Modeling Framework [3]. BE has also been applied to early-stage co-modelling of software and
The BMP consists of four phases with associated tasks: formalisation by translating requirements; a fitness-for-purpose test by integrating requirements; specification by identifying and correcting defects; and design by refining the specification using design decisions. The newly refined design phase of the BMP is now separated from specification and results in an executable BE model that integrates several components to form a system. The refinement of the specification into a design is achieved by examining the boundaries of the system described by the BE model. As BE models are component-based, the system has two boundaries: the System-Environment boundary and the System-Component boundary. The specification is refined by determining the location of these two boundaries. Placing the System-Environment boundary determines which components in the specification exist in the environment outside of the sensing capabilities of the system. Placing the System-Component boundary consists of determining the balance of adding complexity either to the system or the components, and managing the association of components with behaviors using the principals of responsibility-driven design.

The BML is a graphical, formal language consisting of three tree-based views: Behavior Trees (BTs), Composition Trees (CTs) and Structure Trees\(^1\). A BT is a formal, tree-like graphical form that represents behavior of individual entities or networks of entities which realize or change states, make decisions, respond-to/cause events, make and break relations, and interact by exchanging information and/or passing control [5].

A BT node consists primarily of a component that is exhibiting some behavior, with information providing traceability to the requirements on the left of the node as is shown in Figure 2. Traceability to the original requirements is maintained using a node coloring convention where green nodes represent original functionality, yellow nodes (+ status) represent implied functionality, and red nodes (- status) represent missing functionality. Additionally, when nodes are deleted from the BT they are colored grey ——

\(1\) Due to space restrictions Structure Trees will not be discussed.

\(2\) A BT has a structure of a tree, but is implicitly a graph as node operators form semantic links that break the tree structure.
The formal semantics of BTs are described in the Behavior Tree Process Algebra (BTPA) language [6]. BTPA supports simulation, formal verification by model-checking and is a foundation for BT execution. BTs can describe multiple threads of behavior operating in parallel, with coordination achieved by using either message-passing (via events), shared variable blocking or synchronisation. BTs also maintain a clear separation of computation and integration [7], in which the interacting behavior of components is captured separately from the computation which they encapsulate. A summary of the BT notation is shown in Figure 3.

A CT is a tree of components arranged into a compositional hierarchy using structural and functional aggregation or specialisation relations. CTs contain the complete system vocabulary. This vocabulary is consistent with the vocabulary used in BTs as they both originate from the same natural language requirements. Each component in the CT contains the complete set of states, attributes, events and relations for which the component is responsible. CTs are an important tool in resolving defects, such as aliases, which are not easily visible in BTs.

3. The Behavior Runtime Environment

The Embedded Behavior Runtime Environment (eBRE) is a small footprint real-time operating system that executes BE models. A BE model is executed by the eBRE after the application of transformations and code generation into C source code. The transformations for the eBRE are written using the ATLAS Transformation Language (ATL) [8], and are applied to the BT in order to simplify the execution
of a BE model. The transformations remove more complex notation such as node operators, and replace them with a special edge or a group of nodes that fulfill the same purpose. For example, when transforming a node with a reversion operator, the destination node is located and linked to the new special edge. This removes the need to search for the destination node that the node with the reversion operator is linking to each time it is executed. Code generation for the eBRE is written in the Java Emitter Template (JET) language and is used to convert the BT and the CT into C header files which are executed by the eBRE.

The eBRE version of the CT is called a Deployment Composition Tree (DpyCT), as it contains details about how to deploy the BE model. This deployment information describes the starting context of the system and the starting state of one or more components. The starting context normally consists of a single BT node at which execution begins, but more complex contexts where several threads are active are possible. Including the starting states of components in the DpyCT avoids the need to record these initial states at the beginning of the Design Behavior Tree. This makes deployment more flexible as components often have to be initialised to different starting states for different starting contexts. When the BE model executes, the DpyCT becomes the Dynamic Composition Tree (DynCT), and stores the current context of the system and each component’s current state. This allows the eBRE to check the state of a component without having to directly interrogate the component’s process.

At the core of the eBRE is a process control model [9], similar to those used in Operating Systems. When executing a BE model, each BT node can be considered a thread which executes the computation related to the node’s behavior. Similarly, the component associated with the BT node can be considered a process, composed of multiple BT node threads. Each BT node within the process control model has a state of either New, Ready, Blocked or Running based upon the BT semantics and the current context of the system. BT nodes take one of two paths through the process control model depending on the behavior type of the BT node (as described in Figure 3), and some special cases. The Ready Path consists of the transitions, New⇒Ready⇒Running, and the Blocked Path consists of the transitions, New⇒Blocked⇒Ready⇒Running. State Realisations, Selections, and Output Events generally take the Ready Path, whereas Guards and Input Events take the Blocked Path.

The behavior type of the BT node and its parent edge also determines how it behaves when in the process control model. A BT node with an atomic parent edge is added to the model with a Ready state rather than a New state and marked to be the next node to enter the Running state. Another case is when a BT node with an alternate branching parent edge transitions from Ready⇒Running or Blocked⇒Ready. When this occurs, any sibling nodes in the process control model are removed.

By default, after a BT node in the Running state has been executed, its children are added to the process control model in the New state and the executed BT node in the Running state is removed from the process control model. As an exception to this, Selections only add their children to the process control model if their behavior evaluates to true. If a Guard evaluates to true it is unblocked, transitioning from the Blocked state to the Ready state. Input Events wait to be unblocked by messages added to the eBRE, either from the environment (for external input) or from an Internal Output Event (for internal input). When Running, Output Events send messages to the eBRE, which are either sent to the environment (for external outputs) or added to the eBRE (for internal outputs).

Executing a State Realisation BT node accesses the thread associated with the behavior of the component. For example, when a BT node LIGHT [On] is executed, it consists of executing the associated thread On encapsulated in the Light process. As the thread represents an encapsulated computation, any BT nodes that follow sequentially from this node in the tree must then wait for this thread to be completed. The only case when a thread may break the encapsulation of its component process is to setup an interrupt to occur. When an interrupt occurs, its execution is considered outside of the eBRE, and may only interact with components by sending an external message to the eBRE.

The footprint of the eBRE without any BT information or user code is 2.6 kB. On an AVR atmega128 with a 16MHz frequency, the eBRE can theoretically perform in excess of 100,000 cycles of the scheduler per second. Of course, in practice this figure is limited by the structure of the BT being executed and the execution time of the component code. In order to create this small footprint, the eBRE has a number of limitations. The eBRE does not block and restore running threads, and only uses a simple First In First Out (FIFO) scheduler to determine the running thread from those ready to be run. The expression parser of the eBRE, currently only matches against the current state of the component. We have developed a more complex expression parser which can parse numerical expressions, set expressions, and manage messages consisting of expressions but this functionality was not considered necessary for low-level implementations.

4. Case Study: The One-Minute Microwave

The system used for this case study is based upon the One-minute Microwaver which was used to describe state-transition diagrams of the Shlaer-Mellor method [10]. This system has been described in various subsets of UML [11], [12]. It also has been used previously to discuss in detail how to perform requirements analysis with BE [5].
The One-minute Microwaver is a simplified microwave oven which cooks in increments of one minute, as determined by the number of times a button is pushed. The system has the restriction that when the door is open, the oven is unable to cook due to the potentially harmful effects. The requirements for the microwave oven are shown in Table 1. Figure 4 shows a mock-up of the One-minute Microwaver, complete with hardware components (For safety purposes, the power-tube was simulated with a red light) and an embedded controller running the deployed BE model.

The first step in creating a BE model is to translate each of the natural language requirements one at a time into a Requirement Behavior Tree (RBT). The aim when forming an RBT is to capture the original intent of the system's analyst whilst removing any ambiguity from the natural language requirement. During this process the Requirement Composition Tree (RCT) is used to collect the information about each component that has been translated, thereby providing a system overview. Figure 5 shows the result of translating the first requirement of the microwave oven into the resulting RBT and RCT.

During the second stage of the BMP, the RBTs are integrated together to form an Integrated Behavior Tree (IBT). The third stage of the BMP consists of refining the IBT to form a Model Behavior Tree (MBT). The MBT of the One-Minute Microwaver can be seen in Figure 6 with green nodes representing the original functionality described in the requirements. The MBT shows implied functionality with yellow nodes with a + traceability status and functionality missing from the requirements with red colored nodes with a - traceability status. When nodes are deleted from a BT they are colored grey with a - - traceability status and nodes representing design changes are colored white with a + - traceability status.

An example refinement of the IBT to form the MBT is extracting the interactions of the User from the controlling thread of behavior to highlight that the user may interact with the Door and the Button at any time regardless of the current state of the microwave oven. The MBT also models that the User is unable to place food in the oven or remove food from the oven if the Door is closed.

As the MBT is a specification, defining details such as the user interactions are important in order to describe what the system must do. In order to create a design, we must create a DBT that describes how the system will satisfy the specification defined by the MBT.

A DBT naturally emerges from an MBT by clearly defining the two boundaries that exist in a BE design. The two boundaries are the System-Environment boundary and the System-Component boundary. Figure 7 shows the two boundaries as they relate to the microwave oven.

When modelling with BE, it is normal for the MBT to model a portion of the environment in which the system will exist, so that interactions between the system and the environment may be specified. During the design stage, the MBT and the Model Composition Tree (MCT) must be examined to determine the most suitable location for the System-Environment boundary. The System-Environment boundary defines any interactions that take place between the environment and the BE system. These interactions may be coming from a user or an external system, communicating through an interface.

In the case of the microwave oven, it is clear the User, the Door and the Button are not components in the Oven System by the following reasoning. The system does not control these components i.e. the system cannot open the Door or push the Button. The system also does not receive events directly from these components. i.e. the Door does not tell the system it has been opened. In a design, the opening and

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>There is a single control button available for the use of the oven. If the oven door is closed and you push the button, the oven will start cooking (that is, energize the power-tube) for one minute.</td>
</tr>
<tr>
<td>R2</td>
<td>If the button is pushed while the oven is cooking, it will cause the oven to cook for an extra minute.</td>
</tr>
<tr>
<td>R3</td>
<td>Pushing the button when the door is open has no effect.</td>
</tr>
<tr>
<td>R4</td>
<td>Whenever the oven is cooking or the door is open, the light in the oven will be on.</td>
</tr>
<tr>
<td>R5</td>
<td>Opening the door stops the cooking.</td>
</tr>
<tr>
<td>R6</td>
<td>Closing the door turns off the light. This is the normal idle state, prior to cooking when the user has placed food in the oven.</td>
</tr>
<tr>
<td>R7</td>
<td>If the oven times out, the light and the power-tube are turned off and then a beeper emits a warning beep to indicate that the cooking has finished.</td>
</tr>
</tbody>
</table>

Table 1. Requirements of the Microwave Oven

Figure 4. The BE One-Minute Microwaver
If the oven door is closed ... 

... and you push the button, ... 

... (that is, energise the power-tube) ... 

... the oven will start cooking ... for one minute.

(a) Requirement Behavior Tree of first requirement; (b) Graphical view of Composition Tree after translation of first requirement; (c) Details from Composition Tree.

Figure 5. Initial Translation of Requirement #1 of the Microwave Oven

closing of the Door and the pushing of the Button is handled by physical components which relays the information to the microwave oven system using interrupts. These interrupts then interact with the eBRE using messaging.

The System-Component boundary determines the balance between adding complexity to the BT or to the component implementations. It also incorporates responsibility-driven design to determine if the current behaviors are most suited to the components which they belong to. For example, in the specification of the microwave oven, the oven cooks for one minute, cooks for an extra minute and times out. According to the principles of responsibility-driven design this behavior is best assigned to a timer component. The time out event should also be based upon a timer interrupt which the eBRE receives.

The DBT that results after this process is shown in Figure 8. The BE Model has been implemented on an AVR atmega128 operating at 16MHz. The total size of the project, complete with manually created component code, was 5.5 kilobytes. Approximately 20% of the total code was manually coded and another 20% was the automatically generated representation of the BE model. The remaining 60% is accounted to the eBRE.

Even for this small case study, use of the eBRE greatly simplified the task of manual coding. The component code that had to be manually defined had a clear and concise purpose, and was separated from any interacting behavior. Interrupt code only ever consisted of reconfiguring the interrupt to reoccur and adding a message to the eBRE using external input. The more complex behavior involving interacting components was dealt with at a higher-level of abstraction, in the DBT. This greatly simplified testing and debugging, which proceeded in two stages. Firstly, the DBT was tested to ensure it functioned as expected using a PC version of the eBRE. Debugging was aided by views of the DynCT and the system’s context (as represented by the state of the BT nodes in the scheduler). Then manually defined component code was added to the deployed system, and confirmed to be functioning correctly. This essentially involved a unit-based approach which allowed the system to be built incrementally, and also enhanced the potential for reuse of component code in future projects.

5. Related Research

Though not technically a software modelling technique, virtual platforms allow the software for embedded systems to be developed and tested as the hardware is still being designed. Examples of virtual platforms include VaST’s Virtual Prototypes [13] and Virtutech’s Simics full system
simulation [14]. Tools using virtual platforms support numerous software modelling techniques including MATLAB, UML, SystemC and programming languages like C or C++. Embedded systems that have a close coupling with sensors and actuators often use a hardware-in-the-loop approach. Examples of modelling techniques that support a hardware-in-the-loop approach, include Labview graphical design environment [15] and MATLAB [16].

Labview uses a dataflow programming language called G, which consists of multiple functional blocks wired together. Labview models are deployed as embedded systems, by being compiled into executable code which executes on the Labview runtime engine. Producing code for embedded systems with MATLAB requires the Real-Time Workshop add-on which converts Functional Block descriptions of Simulink and the MATLAB textual language into ANSI C code [16].

Software for embedded systems is particularly suited to graphical modelling techniques. Graphically modelling an embedded application decomposes it into smaller, independent computations which can then be described in a lower-level language. The more complex interacting behavior can then be generated from the graphical model, rather than manually coded. The UML is often applied for this purpose, though due to ambiguous semantics, more formalised subsets of the UML language must be used [17].

Bridgepoint provides a tool to implement executable UML (xtUML) models, which consists of a mixture Class Diagrams, State Chart Diagrams and computation described in an Action language [18], [12]. Telelogic provides the Rhapsody tool, a complete UML and SysML modelling tool that is also suited for embedded systems development [19]. StateWorks take a simpler approach, using finite state machines which can be executed on a target embedded system using a run-time system [11].

The application of components to embedded systems has also been investigated in the literature. The COMDES framework can be used to develop embedded systems using prefabricated executable components [20]. The COMDES framework is founded on system engineering principles, using diagrams based upon functional blocks supported by state machines for complex behavior.

The problem with all of these approaches is that the modeler is often allowed too much creativity when developing a software model from the requirements. UML-based approaches such as Rhapsody try to address this by providing requirements capture using a mixture of use cases, sequence diagrams, statechart diagrams and activity diagrams [21]. Unfortunately, the information captured in each of these diagrams is often hard to coordinate, and progressing from the requirements to a design is non-trivial. Other approaches do not provide any requirements capture, creating informally specified models using a construct-by-correction approach. Behavior Engineering seeks to bridge the gaps in the transition from requirements to a design, and create a clear path from requirements to embedded system software consisting of a sequence of explicit, formally defined steps.

6. Conclusion

This paper addresses the issue of how to create software for an embedded system that satisfies the requirements for which it is intended. It advocates a correct-by-construction model-driven development approach, in which software for embedded systems is systematically built out of the requirements. The approach uses BE, and is illustrated with a case study involving a microwave oven. Previously, the main focus of BE has been on performing requirements analysis in order to produce a specification. This ensures that models created with BE satisfy their requirements. This paper builds on this to describe how BE is also suited to, and can be used for design and deployment of embedded software systems.

In addition to the primary benefit of satisfying requirements, the case study elicited some additional benefits of using BE for developing embedded systems. The holistic view of a BE specification, which includes the environment, makes it easier to determine the boundaries of the system when creating a design. Capturing more complex interacting behavior in the BE model removes the need to describe it in languages with a lower level of abstraction such as C or assembly language. Software developed with BE is also easier to develop, test, update, maintain and reuse as it has been partitioned into small, independent pieces of behavior encapsulated in a software component.

References


Figure 6. The Model Behavior Tree of the Microwave Oven
char seconds = 0;
char timer (char behavior, char behaviorType){

One Minute
if (behavior == TIMER_ONE_MINUTE){
    seconds = 5;
    // Enable Timer0 Overflow
    ETIMSK |= _BV(TOIE3);
    return 1;
}

Extra Minute
}else if(behavior == TIMER_ADDITIONAL_MINUTE){
    seconds+=5;
    return 1;
}

return 0;
}

ISR(TIMER3_OVF_vect)
{
    TCNT3 = 3035;
    if (seconds > 0){
        seconds--;
    }else{
        addExternalMessage(TIMED_OUT);
        // Disable Timer3 Overflow
        ETIMSK &~ _BV(TOIE3);
        TCNT3 = 3035;
    }
}

---

Figure 7. BE Boundaries of the One Minute Microwaver
Figure 8. The Design Behavior Tree of the Microwave Oven