Rule-Based Behaviour Engineering: Integrated, Intuitive Formal Rule Modelling

Lin Wah Chan, René Hexel, Lian Wen

*School of Information and Communication Technology,
Griffith University, Nathan Campus,
Brisbane, Australia*

Email: linwahchan@gmail.com, r.hexel@griffith.edu.au, l.wen@griffith.edu.au

Abstract—Requirement engineering is a difficult task which has a critical impact on software quality. Errors related to requirements are considered the most expensive types of software errors. They are the major cause of project delays and cost overruns. Software developers need to cooperate with multiple stakeholders with different backgrounds and concerns. The developers need to investigate an unfamiliar problem space and make the transition from the informal problem space to the formal solution space. The requirement engineering process should use systematic methods which are constructive, incremental, and rigorous. The methods also need to be easy to use and understand so that they can be used for communication among different stakeholders. Is it possible to invent a human intuitive modelling methodology which systematically translates the informal requirements into a formally defined model?

Behaviour Engineering has arguably solved many problems. However, the size and low level of the final Behavior Tree makes it hard to match with the original requirements. Here, we propose a new requirement modelling approach called Rule-Based Behaviour Engineering. We separate two concerns, rules and procedural behaviours, right at the beginning of the requirement modelling process. We combine the Behavior Tree notation for procedural behaviour modelling with a non-monotonic logic called Clausal Defeasible Logic for rule modelling. In a systematic way, the target model is constructed incrementally in four well-defined steps. Both the representations of rules and procedural flows are humanly readable and intuitive. The result is an effective mechanism for formally modelling requirements, detecting requirement defects, and providing a set of tools for communication among stakeholders.

Keywords—Requirement Engineering; Requirement Modelling; Model-driven engineering; Modelling language; Rule-Based Behaviour Engineering; Rule-Based Behavior Tree; Non-monotonic reasoning; Defeasible Logic; Clausal Defeasible Logic; Behavior Tree; Behaviour Engineering

I. INTRODUCTION

It is widely accepted that requirements engineering (RE) is an extremely difficult task in the software engineering field [1]. It is difficult because (a) software developers are usually unfamiliar with the problem domain, and (b) requirements are usually presented in natural language in an informal, incomplete, ambiguous and often defective way. Furthermore, software developers have to cooperate with stakeholders from different backgrounds with different and

sometimes conflicting concerns. Last but not least, developers have to overcome the gap between informal requirements and a formal representation of these requirements. This means they have to create a complete, structured set of consistent and precise representations from incomplete and unstructured collections of sometimes inconsistent or incorrect statements. It also means they must make hidden, implicit needs and assumptions explicit.

Errors related to requirements have recurrently been recognised as the most expensive, persistent, and frequent types of software errors. They are the major cause of project delivery delays, budget overruns, and failure to meet expectations [2].

Considering the impact and difficulties as a consequence of the above, systematic methods should be used to minimise the errors that may occur in the RE process. The methods should be constructive, have effective guidance in model building, be incremental for early analysis, and rigorous enough to produce accurate models, but easy to use for communication among different stakeholders [2].

Behaviour Engineering (BE) as an arguably revolutionary approach [3] has been proven to be efficient to handle those problems [4], especially to identify the requirement defects [5]. However, a major drawback of the traditional BE approach is that when each logic operation is modelled with the Behavior Tree (BT) notation, the final Behavior Tree will be very low-level and can be very large. Consequently, it will be difficult to present and to follow.

In this paper, we propose *Rule-Based Behaviour Engineering* (RBBE), a new requirement modelling approach to supersede the traditional BE approach it is based on. Right from the beginning of the requirement modelling process, we radically separate two major concerns, the procedural behaviours and the rules that govern it. In a systematic, constructive and incremental way we transform informal requirements, which are written in natural language, into a single, complete *Rule-Based Behavior Tree* (RBBT) system, a high level model that can be used as a basis for further refinement and implementation. By separating the concerns, the formal model is able to more closely track the original set of requirements (expressed in informal, human language).

Table I: Common behaviour types in Behavior Tree

[State]	<i>State Realisation</i> : Change the state or the value of an attribute of a component.
?Condition?	<i>Condition checking</i> : If true, flow continues to its child nodes, else flow terminates.
??Event??	<i>Event</i> : Wait until the event occurs.
???Condition???	<i>Guard</i> : If true, flow continues to its child nodes, else wait until the condition comes true.
>>Message<<	<i>Input</i> : Wait until the message arrives.
^	<i>Reversion</i> : Flow reverts back to the closest parent node with the same component name, same behaviour and same behaviour type.

This way, our method inherits all the advantages from the traditional BE approach including maintaining the traceability, detecting ambiguities, requirement defects, etc. Furthermore, subsystems and components are identified early in the process, vastly simplifying the modelling and integration of complex systems. Most importantly, as the new approach separates the procedural behaviour and logic rules, the models are smaller and easier to understand and verify.

The rest of the paper is organised as follows. Section II gives an overview of relevant background research. Section III describes our proposed methodology and the design process used for RBBE. Section IV exemplifies the RBBE process through a real Software Engineering case. The paper is concluded by a discussion in section V.

II. BACKGROUND

A. Behaviour Engineering and Behavior Trees

The BT notation is a graphical language which formally models the functional behaviour of a software system [6]. The BE method systematically and incrementally constructs the whole system [7]. The incremental way of constructing the BT helps control the complexity of the requirements. The process of translation and integration reveals requirement defects such as incompleteness, inconsistency, ambiguity, inaccuracy and redundancy [8]. These types of defects are usually difficult to uncover because of the limitation of human short-term memory.

The BE method is component-based [6]. Nodes of component behaviour are linked sequentially or concurrently to make a formal BT, which specifies state changes in components, how data is passed between components and how threads interact [6]. The BT notation is easy to learn because it only has several node types. The basic types of node are listed in Table I. Together they can handle component states, logic, events, control-flow, data-flow and threads [6]. BT has a formal semantics [9] and can be model checked [10]. It has been used to detect requirement defects [11], perform software change impact analysis [6], and co-model with Modelica [12].

One of the aims of the BE method design is to promote the shared understanding of requirements among stakeholders of a system [8]. Non-technical stakeholders usually have difficulty understanding formal method representation of requirements. Compared with other popular system design methods, including State Transition Diagrams, Functional Flow Block Diagrams, Object Oriented Design, IDEF0, UML and SysML, BTs come closest to natural language specification in terms of what they can express [13].

Traditional BTs models logic with procedural flows using nodes and edges. When the logic becomes more complex, the branches are not easy to follow. Our idea of replacing the logic in BTs with a declarative human intuitive language raises the level of abstraction regarding logic representation [14]. It eliminates errors that might occur during the modelling of logic with BT nodes. This lets developers focus on “What” instead of “How”. In addition, the rise in abstraction level enables the models to stay at a high level, making them easier to understand for different stakeholders.

B. Non-Monotonic Reasoning

Non-monotonic reasoning works in a way similar to common sense reasoning [15], acquisition of new knowledge can polish or correct the old understanding of a concept [16]. This separates them from classical monotonic logic in which learning a new piece of knowledge cannot reduce the set of old understanding [15].

Non-monotonic reasoning can handle, in a human readable way, complex reasoning tasks, which cannot be handled by monotonic logic. These tasks include reasoning by default, abductive reasoning, and belief revision [15].

Isaac Asimov’s famous Three Laws of Robotics are a good example to show how humans define rules.

- 1) A robot may not injure a human being or, through inaction, allow a human being to come to harm.
- 2) A robot must obey any orders given to it by human beings, except where such orders would conflict with the First Law.
- 3) A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

They define a set of rules so that some have a higher priority over the others. Incremental refinement of a rule set is possible in non-monotonic logic [16].

C. Clausal Defeasible Logic

The family of non-monotonic logics called Defeasible Logics has the advantage of being designed to be implementable [17][18]. Defeasible Logics have been used in a wide variety of areas, such as an expert system for learning and planning [18], and in a robotic dog which plays soccer [19] [20]. Defeasible Logics have been advocated for applications including modelling regulations and business rules

[21], agent negotiations [22], the semantic web [23][24], modelling agents [25], modelling intentions [26], modelling dynamic resource allocation [27], modelling contracts [28], and legal reasoning [29]. The wide variety of applications indicates that Defeasible Logics are important for knowledge modelling and reasoning.

A recent addition to this family is Clausal Defeasible Logic (CDL) [30]. CDL reasons with three types of rules and a priority relation. These rules are strict rules, defeasible rules, and warning rules [30].

Defeasible rules are used to represent a plausible situation. They are written in the form $A \Rightarrow l$. If all the evidence against l has been nullified then we can deduce l . For example, Birds usually fly will be written as $bird(x) \Rightarrow fly(x)$ [30].

The priority relation, $>$, are used on the set of defeasible rules to represent preferences among them. The priority relation must not be cyclic [30]. Billington (2008) gives an example of how priority relation works.

$r1 : bird(x) \Rightarrow fly(x)$	Birds usually fly.
$r2 : quail(x) \Rightarrow \neg fly(x)$	Quails usually do not fly.

$r2$ is more specific than $r1$. Thus we specify $r2 > r1$, meaning that $r2$ is preferred over $r1$.

CDL has six proof algorithms. The monotonic μ algorithm is used only for strict rules. CDL restricted to μ is practically classical propositional logic. The β algorithm is for ambiguity blocking. Algorithm ρ and π are for ambiguity propagating. The last two algorithms ι and ε are co-algorithms for ρ and π [30].

A recursive proof function performs the task of formula proving. Initially it requires two inputs, namely the proof algorithm to be used and the formula to be proved. A third input called background is required to detect loops during the recursive calling of the function itself [30].

The proof function outputs three possible proof-values, +1, 0, -1. The +1 means that the formula was proved in a finite number of steps. The 0 means that the proof got into a loop, which did not end after a finite number of steps. The -1 means that the formula was disproved [30].

D. Separation of Rules From Procedural Flows

Separation of rules from procedural behaviour is not a new idea. Rule engines such as “JESS” [31] and “Bossam” [32] have been developed for this purpose. They are used mainly as an inference engine for expert systems. Prolog is a popular logic programming language. However, it is too low level and unnatural for different stakeholders to comprehend. RuleML is a markup language which is closer to natural language. It is for sharing rules on the Word Wide Web.

Billington et al. proposed to use non-monotonic with state diagrams for RE [16]. They successfully demonstrate that the separation of rules from the procedural behaviours are

feasible and implementable. Plausible logic is used for state transitions in their work [16].

We believe that the separation of the two main concerns certainly will benefit the requirement modelling process. However, following the original BE method to model logic with BT nodes in a procedural way first and then convert the BT nodes into CDL theories is very inefficient and does not benefit the process. This leads us to invent a new requirement modelling approach which separates the two concerns at the beginning of the modelling process. This paper goes beyond previous works by providing a systematic way to go from informal requirements to a formal model while keeping the two concerns separated.

III. DESIGNING RULE-BASED BEHAVIOUR TREES

A. Overview

The RBBE methodology we propose here is a formal approach to requirement modelling. It provides a systematic and rigorous way for software developers to transform informal natural language requirements into a human intuitive formal model. At its core is the separation of two major, but fundamentally different concerns: procedural behaviour and the rules that govern that behaviour. We argue here that this is typically how requirements are specified originally and therefore this separation needs to occur right at the beginning of the requirements engineering process. While we believe that any common-sense non-monotonic logic can be used to represent rules, we use CDL to represent rules in conjunction with graphical BT notation to model state changes and other procedural behaviours. The feasibility of the implementation and integration of CDL into BTs has already been proven [14]. The integration of the two expressive and user-friendly languages enhances the communication between developers and other stakeholders. By keeping the formal model at a high level close to the original requirements, our approach detects defects in requirements at the early stages of the development life cycle, and paves a fast and smooth path from informal to formal representation of requirements.

A key premise of the RBBE method is that behaviour is always associated with components. Every action performed by a component is governed by a set of rules. These rules are conditions based on certain states, attribute values of components, and/or external or internal events. When the conditions are satisfied, the component take predefined actions, either changing the state of a component or changing the value of an attribute in a component.

The RBBE method includes four steps.

- 1) Translate each requirement into several rows in a table called the *Precondition-Component-Action Table* (PCAT).
- 2) Create CDL rule sets based on the information in the PCAT.

- 3) Construct RBBTs based on the remaining information in the table.
- 4) Integrate all RBBTs according to their interfacing requirements.

B. Illustrating RBBE with a Classical Example

To help illustrate the method, let us look at a classical example which has been repeatedly used by the software engineering community, the so-called “One-minute microwave oven” [5][6][33][34]. The requirements for the microwave oven are as follows.

- 1) There is a single control button available for the use of the oven. If the oven is closed and you push the button, the oven will start cooking (that is, energise the power-tube) for one minute.
- 2) If the button is pushed while the oven is cooking, it will cause the oven to cook for an extra minute.
- 3) Pushing the button when the door is open has no effect.
- 4) Whenever the oven is cooking or the door is open, the light in the oven will be on.
- 5) Opening the door stops the cooking.
- 6) 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.
- 7) 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.

The following paragraphs explain the four steps in detail using the microwave oven requirements as an example.

1) *Creating the PCAT:* This step is to factor out two concerns, rules and procedural behaviours. To do this we convert the requirement specification into a PCAT. Table II shows the complete PCAT for the microwave oven. The PCAT contains four columns. The first column is the requirement number for traceability. The second column is the precondition column which includes two parts. The first part contains preconditions under which a component acts. The *Precondition Axiom* is one of the two axioms in the BE method. It states that every constructive, implementable, individual functional requirement has associated with it a precondition that needs to be satisfied in order for the behaviour in the functional requirement to be applicable [8]. Preconditions are essentially states of components and/or attribute values of components connected by logical operators. The second part contains event(s) that trigger the action. The third column contains names of components that are related to either actions or post-conditions. The last column is the action column. It also has two parts, action and post-condition.

The double vertical lines in the middle of the PCAT indicates that the component names on the right of them are related to the last column only. They are unrelated to the precondition column. Although we do not separate

Table II: The Precondition-Component-Action Table (PCAT) for the One-Minute Microwave Oven

Req.	Precondition Trigger	Component Component	Action Post-Condition
1	Door is closed and Oven is idle	Timer	Set time interval to 1 minute
	Button is pushed	Timer	Running
1	Door is closed and Timer is running	Oven	Cooking
1	Oven is cooking	Power Tube	On
2	Oven is cooking	Timer	Add 1 minute to time interval
	Button is pushed		
4	Door is open or Oven is cooking	Light	On
5	Door is open	Oven	Idle
6	Door is closed and Oven is idle	Light	Off
7	Timer timeout	Timer	Idle
7	Door is closed	Beeper	Beep
	Timer timeout	Beeper	Idle
7	Timer is idle	Oven	Idle
7	Oven is idle	Power Tube	Off
7	Door is closed and Oven is idle	Light	Off

components from the conditions, we need to name the component when specifying each precondition and trigger so that the component names form part of the conditions. For example, the precondition “Door is closed” contains the component name `Door`. We do not extract component names from conditions because preconditions are sometimes complex with many logical operators and brackets. It would be difficult to present the logical relationship in an intuitive way if we separate the component names out.

Since the definition is clear for each box in the PCAT, the production of a PCAT is not much harder than a fill-in-the-blank exercise for each requirement in its original representation. Thus, the PCAT is an intermediate working document towards the formation of RBBTs. Depending on how the requirements are interpreted, there can be more than one revision of a PCAT (e.g. after resolving ambiguities or defects). The clear definition of every box in the PCAT

Table III: The CDL theory for the Light control

Req.	CDL rule	Explanation
	$L0 : \{\} \Rightarrow \neg \text{LightOn.}$	Usually the light is not on.
4,6	$L1 : \text{DoorOpen} \Rightarrow \text{LightOn. } L1 > L0.$	If the door is open, the light is on. This rule overrides rule $L0$.
4,6,7	$L2 : \text{OvenCooking} \Rightarrow \text{LightOn. } L2 > L0.$	If oven is cooking, the light is on. This rule overrides rule $L0$.

Table IV: The CDL theory for the Oven control

Req.	CDL rule	Explanation
	$L0 : \{\} \Rightarrow \neg \text{OvenCooking.}$	Usually the oven is not cooking.
1,5,7	$L1 : \text{DoorClosed and TimerRunning} \Rightarrow \text{OvenCooking. } L1 > L0.$	If the door is closed and the timer is running, the oven is cooking. This rule overrides rule $L0$.

Table V: The CDL theory for the Power Tube control

Req.	CDL rule	Explanation
	$L0 : \{\} \Rightarrow \neg \text{PowerTubeOn.}$	Usually the power tube is not on.
1,7	$L1 : \text{OvenCooking} \Rightarrow \text{PowerTubeOn. } L1 > L0.$	If the oven is cooking, the power tube is on. This rule overrides rule $L0$.

forces the developer to interpret the requirements rigorously. It is a useful tool for discussion between developers and other stakeholders for the clarification of the requirements as the PCAT is written with the terms used by the users in natural language. You can keep improving the PCAT until it is ready for the next step. However, the focus in this step is not to arrive at an optimal design, but to clarify the requirements.

After creating the PCAT, we can group the table entries by components. By doing so, many requirement defects, such as redundancy, inconsistency and ambiguity, can be uncovered.

2) *A rule set for each component action:* The purpose of this step is to model the logical behaviours separately with CDL. A rule set allows rules in it to be prioritised. It is also called a theory in CDL. We shall use the two terms interchangeably. All CDL theories for the microwave oven example are shown in Table III through Table VIII. Human readable rule sets can be understood by average stakeholders without much training. It is another great tool for promoting shared understanding among stakeholders.

From a PCAT we can group all preconditions for a particular pair of component action with little effort. These grouped entries show the collection of conditions that decide whether the action should be taken. From them we can easily create a CDL theory for each pair of component action. In terms of logical evaluation, events and component states are

Table VI: The CDL theory for the Beeper control

Req.	CDL rule	Explanation
	$L0 : \{\} \Rightarrow \neg \text{Beep.}$	Usually the beeper is silent.
7	$L1 : \text{DoorClosed and TimerTimeoutEV} \Rightarrow \text{Beep. } L1 > L0.$	If the door is closed and the timer timeout, the beeper beeps. This rule overrides rule $L0$.

Table VII: The CDL theory for the Timer set control

Req.	CDL rule	Explanation
	$L0 : \{\} \Rightarrow \neg \text{TimerSet.}$	Usually the timer is not set.
1	$L1 : \text{DoorClosed and OvenIdle and ButtonPushedEV} \Rightarrow \text{TimerSet. } L1 > L0.$	If the door is closed and the oven is idle and the button is pushed, the timer is set. This rule overrides rule $L0$.

Table VIII: The CDL theory for the Timer add control

Req.	CDL rule	Explanation
	$L0 : \{\} \Rightarrow \neg \text{TimerAdd.}$	Usually the time of timer is not added by 60 seconds.
2	$L1 : \text{OvenCooking and ButtonPushedEV} \Rightarrow \text{TimerAdd. } L1 > L0.$	If the oven is cooking and the button is pushed, the time of timer is added by 60 seconds. This rule overrides rule $L0$.

treated equally in CDL rules. We append “EV” to trigger names to distinguish them. During the creation of the CDL theory, conflicting requirements regarding when an action should be taken can clearly be identified. Sometimes the conflicts can be resolved by prioritising them in the CDL theory.

Theoretically, a rule set should be created for each state transition within a component. For example, if a component has three states, we should have six rule sets. In practice, some transitions can share a same rule set. For example, the rule set in Table III can be shared by both directions of the transitions between the on and off state of the light.

In the RBBE method, we wrap a rule set with a component called “oracle”, which is modelled with the BT notation. Each oracle has a standard sub component called “CDL_Wrapper” that calls an external CDL engine to perform the reasoning based on the CDL theory. The basics of oracles can be found in our last paper [14]. An oracle takes as inputs some component states, component attribute values, and unprocessed events. After evaluating the inputs according to the set of rules, if the condition of the action is satisfied, the oracle will send an event to its parent component to trigger the action.

3) *Construct RBBTs based on the PCAT:* The purpose of this step is to construct RBBTs for all components. The PCAT can be searched to collect a full list of component states and behaviours. RBBTs are then formed based on this

Figure 1: The RBBT for Light Control

information. RBBTs are created using the original BT notation. These rule-free components are easier to understand and maintain than the original BTs because of their simple organisation resulted from hiding the logics in oracles. In addition, this step provides a good opportunity to check whether the state cycle of a component is complete. Figure 1, 2, 3 and 4 are examples of RBBTs.

Two types of behaviours can be found in the PCAT. The first one is sensor behaviours which indicate what events or conditions that the components need to detect from the environment or other components. We can find them in the precondition column. The second one is actuator behaviours, which show what the components need to do if certain preconditions are satisfied. They can be found in the last two columns.

The light control shown in Figure 1 is an actuator. Its behaviour is governed by oracles such as `OracleLightOn` and `OracleLightOff`. The conditions specified in the CDL theory (Table III) inform the light control to either turn on or off the light. The oven control is very similar to the light control. It has two states, `Idle` and `Cooking`. The transitions are governed by the CDL theory in Table IV. Power tube control is another similar actuator. The transitions between the two states, `On` and `Off`, are governed by the CDL theory in Table V. Figure 4 shows yet another actuator called “Beeper”. Its behaviour is triggered by the CDL theory in Table VI. After beeping, the state of `Beeper` changes to `Idle` automatically thus no oracle is needed.

The `Door` is a sensor component. It does not rely on oracles to inform its state transitions. Instead, it relies on external events to trigger the state transitions. As a result, the RBBT of `Door` (Figure 3) is different from that of `Light`, event nodes such as `??Open??` and `??Close??` will replace the event nodes for the oracles. The `Button` is another sensor component, it only has one state `Idle` and

Figure 2: RBBT for Timer Control

one external event `??Pushed??`.

The `Timer` component has both sensor and actuator behaviours. These behaviours are connected in a logical way in the RBBT (Figure 2). From the `Timer` RBBT you see how we connect an action and a post condition (the 3rd and 4th node), which are triggered by the same set of preconditions. You also see how an action, which does not change component state, is modelled (the last 2 nodes at the bottom right). Finally, you see how an external event and an oracle event coexist in the same component (the 2 nodes under the `[Running]` state). The `Timer` is a special component because its `time` attribute decreases according to a real clock and the `TimeOut` event will be fired when the `time` is down to zero. Other components do not have the autonomous behaviours. The CDL theories that the `OracleTimerSet` and `OracleTimerAdd` contain are shown in Table VII and Table VIII respectively. There is no oracle governing the transition from `Running` to `Idle`. It is because the preconditions of the transition have already been fulfilled by the preceding node, timer timeouts.

4) *Integration of the RBBTs:* By now the RBBTs of all components have been constructed. The purpose of this step is to integrate these self-sustaining RBBTs based on their input requirements. The inputs for a component are as follows.

- *Component states and/or attribute values from other components which are required by the component to perform actions:* For example, in the case of microwave oven, if we add a new requirement that a screen displays

Figure 3: RBBT for Door

the remaining cooking time, the `Screen` component will need the remaining time from the `Timer` component in order to perform its task.

- *Oracle related inputs:* Each oracle has an agent constantly running to detect incoming events and relevant changes in their environment.
 - *Events (Triggers) required by oracles:* In the RBBE design, when it comes to querying an oracle, two or more events might have already happened but not yet processed. We have a buffer in each oracle for the storage of unprocessed events so that the events can take part in the reasoning process simultaneously. The buffer is cleared after the reasoning process.
 - In some cases, an event might require feedback from the acting component to indicate the completion of the task. Both synchronised and asynchronised communications are considered for event passing.
 - *Component states and attribute values required by oracles:* Oracles need to retrieve the current state or attribute values from other components for reasoning purposes.

The communication between the components is achieved through an approach called *whiteboard* which is proposed in the paper written by Billington, et al [16]. All components share information by writing messages to the *whiteboard*. In this step, all interfacing requirements are examined to ensure they can be fulfilled. Requirement issues such as incompleteness, ambiguity and inconsistency will be uncovered. The requirements are specified at a high level without the need to care about how the events, component states, and other values are passed between components. The integrated RBBT system is a complete and unambiguous specification of the system.

Figure 4: RBBT for Beeper

C. Traceability in RBBE

Traceability is crucial for finding out the original intention of system behaviour and for change management. In the RBBE methodology, direct and clear traceability of the original requirements is maintained throughout the whole process. The relevant requirement numbers are preserved in the PCAT, the CDL theories and all BTs. The PCAT might not be important after the RBBTs are built. However, the CDL theories and BTs are not throwaway models. They form a solid foundation for automatic system generation. These models should evolve together with the actual system, thus providing complete traceability for the software system.

IV. AMBULATORY INFUSION PUMP

To illustrate some of the properties of RBBE in more detail, we use a real case as an example, the software requirements for an ambulatory infusion pump (AIP), a medical device used in drug therapy for patients [35]. Its main task is to pump drugs into a patient's blood circulation system (the requirements are shown in Table IX). Beside the basic functions which allow the pump to work properly, there are safety and security functions which protect patients from being harmed and the device from being abused or mis-operated.

A. Separating events from preconditions in the PCAT

The clear separation of events from preconditions in the PCAT forces the developer to think carefully what the requirement actually means. Natural language requirements tend to ignore the difference between events and states. For example, in R4, the requirement is “When the battery is low, the system sends three beeps.” What it actually means is that in the event when the battery state changes from `Normal` to `Low`, the system sends three beeps. If interpreted otherwise, the system will keep sending three beeps while the battery state is low. Our interpretation is confirmed by R5, in which the system only sends one beep when the battery is low.

Table IX: Ambulatory Infusion Pump Requirements

No.	Requirement
R1	The system is turned on when the batteries are put in and is turned off when the batteries are out.
R2	To start the pump, when in stopped state, start-stop button is held down until it beeps three times and (... ..) is displayed on the screen.
R3	To stop the pump, when in running state, start-stop button is held down until it beeps three times and (... ..) is displayed on the screen.
R4	When the battery is low, the system sends three beeps and displays battery low message on the screen.
R5	Every time the system pumps 1 ml of drug when the battery is low it sends a single beep alarm.
R6	After a set time pump activates to pump 1ml of drug through the line.
R7	When the volume reaches 5ml the system does three beeps and displays volume low message every 1ml as it counts down to empty.
R8	When there is no drug left, the pump enters stopped mode and the system sounds a continuous beeping alarm.
R9	When the line is closed/blocked or kinked the system does a constant alarm beeping if it is in the running mode.
R10	The security mode of the pump can be changed as follows: 1) the pump must be in stopped mode, 2) the current security mode is displayed by pressing the lock button, 3) the up and down arrow buttons are used to select the desired security mode (patient, clinic or program), 4) the enter is pressed to save the selected mode, 5) once the enter button is pressed 000 appears on the screen, 6) the up and down arrow buttons are used to select the correct password, 7) the enter button is used to select the displayed password and as a result displayed security mode is also selected as the current security mode.
R11	The pumps volume can be reset to a preset value as follows: 1) The pump must be in stopped mode, 2) the pump can be in any of the security modes, 3) the next is used to display volume on the screen, 4) pressing the enter button resets the volume to a preset value.
R12	The pumps upper limits of the pumps infusion rate can be set as follows: 1) The pump must be in stopped mode, 2) the security mode on the pump must be set to program mode, 3) the next button is used to display Infusion Rate Upper Limit on the screen, 4) the up and down arrow buttons are used to select the desired infusion rate, 5) pressing the enter button sets infusion rate upper limit to the displayed limit on the screen.
R13	The pumps infusion rate can be set as follows: 1) The pump must be in stopped mode, 2) The security mode of the pump must be set to clinic mode, 3) The next button is used to display Infusion Rate on the screen, 4) The up and down arrow button are used to select the desired infusion rate, 5) Pressing the enter button sets the infusion rate to the infusion rate displayed on the screen.
R14	The amount of drug given can be cleared as follows: 1) The pump must be in stopped mode, 2) The security mode of the pump must be set to clinic mode, 3) The next button is used to display given on the screen, 4) Pressing the enter button clears the given amount to zero.
R15	Whenever air is detected in the line, by the air detector sensor, the pump is stopped and the beeper a continuous beep.
R16	The main screen displays the pump status (running/stopped), battery status (normal/low) and drug volume.
R17	If no key is pressed for 2 minutes then the screen is reset to the main screen.

Table X: The CDL rules for the Beeper

Req.	CDL rule	Explanation
	$L0 : \{ \} \Rightarrow \neg \text{beep}.$	Usually the beeper is silent.
2,3	$L1 : \text{ssBeepsDotsStartedEV} \Rightarrow 3\text{beeps. } L1 > L3.$	If the start-stop super component called “ssBeeps-Dots” is started, beeps three times.
4	$L2 : \text{batteryTurnLowEV} \Rightarrow 3\text{beeps. } L2 > L3.$	If the battery become low, beeps three times.
5	$L3 : \text{1mlJustPumpedEV and batteryLow beep. } L3 > L0.$	If 1ml of drug has just been pumped and the battery is low, beep once.
7	$L4 : \text{volumeReach5mlEV} \Rightarrow 3\text{beeps. } L4 > L3.$	If drug volume lowered to 5ml, beeps three times.
8	$L5 : \text{noDrugLeft beepForever. } L5 > L1, L2, L4.$	If no drug is left, beeps continuously.
9	$L6 : (\text{lineClosed or lineBlocked or lineKinked}) \text{ and pumpRunning beepForever. } L6 > L1, L2, L4.$	If line is closed or blocked or kinked when pump is running, beeps continuously.
15	$L7 : \text{airInLine beepForever. } L7 > L1, L2, L4.$	If air is detected in the line, beeps continuously.

B. Explicit requirements of a post condition in the PCAT

Post conditions are often omitted or assumed in natural language requirements. The explicit requirement of a post condition in the PCAT forces the developer to think carefully about the aftermath of an action. For example, in R2, the requirement “To start the pump, when in stopped state, start-stop button is held down until it beeps three times and (... ..) is displayed on the screen” does not specify the state of the pump after the start action. We derive from it and other requirements that the pump state will be *Running* after the start action. What is more, the requirement does not tell what the screen should display after the action. Thanks to the post condition field, this question become an item in the to-answer list of the developer. We finally find the answer when we see R16 and R17, which tell us that there is a main screen for normal display. Therefore, we can safely conclude that the post condition of the screen for R2 is back to the main screen.

C. Component centric CDL Rules

Usually the actions that a component should take are specified across a number of scattered statements. For example, the behaviour of the beeper in the AIP requirement is described in R2, R3, R4, R5, R7, R8, R9, and R15. Moreover, there are three types of behaviour for the beeper, i.e. single beep, three beeps, and continuous beep. The beeper can only do one type of behaviour at any single time. The way our methodology creates a rule set forces the developer to resolve all conflicting requirements relating to the action taken by a component. In the beeper example,

we find that these different types of actions can be put into a same rule set and be prioritised. See Table X for the resulted CDL rule set. Obviously, the continuous beep has a higher urgency than the three beeps. The three beeps have a higher urgency than the single beep. This rule set can then be shared by several oracles to inform the transition of the states in the beeper component.

V. DISCUSSION AND CONCLUSIONS

A. Comparison with non-monotonic logic in state diagrams

The RBBTs are somewhat analogous to the use of plausible logic in state diagrams [16]. The approach of separating behaviour from rules that govern it has been shown to be extremely powerful, allowing failure modes and effect analysis (FMEA) [36] and formal verification of properties [37]. While we believe that interchangeable semantics between BTs and state machines makes many of these properties applicable to RBBT, our approach goes further by providing a systematic methodology to transform requirements in natural language into their model. Furthermore, the inherent tree form of BTs ensures RBBTs to be planar and more readable than state diagrams, which tend to become difficult to visualise and follow for more complex behaviours.

B. Comparison with Behavior Engineering

The original BE method effectively transforms requirements in natural language to a formal model. The RBBE methodology does the same in a more intuitive, human-readable way. Our method adopts the original BT notation and retains from the original BE method some important beneficial properties such as traceability, incremental construction, stepwise processes, consistency, completeness, defects detection, component-based, and scalability. With the integration of CDL into BTs, the engineering processes depart dramatically from the original method. In the original method, requirements are translated into Requirements Behavior Trees (RBT) first. The RBTs are then integrated into an Integrated Behavior Tree (IBT) by matching the precondition of a RBT with an existing node in the IBT. This forces the developer to connect low level behaviours when their understanding of the problem domain is limited.

In our approach, instead of translating requirements into RBTs, we translate them into a PCAT which can be searched, sorted and grouped. We avoid the use of an IBT which is difficult to comprehend due to its large size. We project RBBTs directly from the PCAT. We integrate the RBBTs with clearly defined interfaces. Rules are represented in a intuitive and declarative style, focussing on the “What” instead of “How” and making the formal specification resemble more closely the original requirements.

C. Conclusion

In this paper we proposed a systematic requirements modelling methodology, guided by a high-level, stepwise

process of separating behaviour from the rules that govern it. The systematic and constructive modelling process creates an easy to understand, formal model that closely matches the original requirements and helps uncover ambiguities and defects. Further refinement and development towards the final software system only happens after such errors are rectified. The resulting model provides a single integrated, formal representation of the system. The same form of representation is used for both requirements and implementation. As a result, the model will be growing with the software system and forming an important part of the documentation as an operational specification. With the right tools selected for the representation of logic and procedural flows separately, the RBBTs are easy to understand and maintain. The PCAT, CDL rule sets and RBBTs can all be used for communication between software developers and different stakeholders. The practical tools facilitating communication and the early detection of requirement defects help shorten the time needed to formalise the requirements. The proposed methodology also provides direct and clear traceability between the final model and the original requirements.

REFERENCES

- [1] S. R. Faulk, *Software Requirements Engineering*, 2nd ed., R. H. Thayer and M. Dorfman, Eds. IEEE Computer Society Press, 2000.
- [2] A. van Lamsweerde, “Requirements engineering: from craft to discipline,” in *Proc. of the 16th ACM SIGSOFT Intl Symposium on Foundations of software engineering*. New York: ACM, 2008, pp. 238–249.
- [3] R. Glass, “Practical programmer: Is this a revolutionary idea, or not?” *Communications of the ACM*, vol. 47(11), pp. 23–25, 2004.
- [4] G. R. Dromey, “Climbing over the no silver bullet? brick wall,” *IEEE Software*, vol. March/April, pp. 96–98, 2006.
- [5] R. G. Dromey and D. Powell, “Early requirements defect detection,” *TickIT Journal*, vol. 4Q05, pp. 3–13, 2005.
- [6] L. Wen and R. G. Dromey, “From requirements change to design change: A formal path,” in *Intl Conf. on integrated Formal Methods*, 2004, pp. 148–167.
- [7] R. Dromey, “From requirements to design: formalizing the key steps,” in *Software Engineering and Formal Methods, 2003. Proc. First Intl Conf. on*, sept. 2003, pp. 2 – 11.
- [8] R. G. Dromey, “Formalizing the transition from requirements to design,” in *Mathematical Frameworks for Component Software - Models for Analysis and Synthesis, World Scientific Series on Component-Based Development*, J. He and Z. Liu, Eds. River Edge, NJ, USA: World Scientific Publishing Co. Inc., 2006, pp. 156–18.
- [9] R. Colvin and I. J. Hayes, “A semantics for behavior trees,” *ACCS Technical Report*, April 2007.

- [10] K. Winter, "Formalising behavior trees with csp," in *Proc. of the 2nd IEEE Intl Conf. on Software Engineering and Formal Methods*, 2004, pp. 104–113.
- [11] D. Powell, "Requirements evaluation using behavior trees: Findings from industry," in *Australian Conf. on Software Engineering*, 2007.
- [12] T. Myers, P. Fritzson, and R. G. Dromey, "Seamlessly integrating software & hardware modelling for large-scale systems," in *Proc. of the 2nd Intl Workshop on Equation-Based Object-Oriented Languages and Tools*, Paphos, Cyprus, 2008, pp. 5–15.
- [13] P. A. Lindsay, "Behavior trees: from systems engineering to software engineering," in *8th IEEE Intl Conf. on Software Engineering and Formal Methods*, Pisa, Italy, 2010, pp. 21–30.
- [14] L. W. Chan, R. Hexel, and L. Wen, "Integrating non-monotonic reasoning into high level component-based modelling using behavior trees," in *SoMeT*, 2012, pp. 21–40.
- [15] G. Antoniou, *Nonmonotonic Reasoning*. MIT Press, Cambridge, Mass., 1997.
- [16] D. Billington, V. Estivill-Castro, R. Hexel, and A. Rock, "Non-monotonic reasoning for requirements engineering state diagrams driven by plausible logic," in *5th Intl Conf. on Evaluation of Novel Approaches to Software Engineering*, 2010, pp. 68–77.
- [17] D. Billington and A. Rock, "Propositional plausible logic: Introduction and implementation," *Studia Logica*, vol. 67, pp. 243–269, 2001.
- [18] D. Nute, "Defeasible logic," in *Web Knowledge Management and Decision Support*, ser. LNCS, O. Bartenstein, U. Geske, M. Hannebauer, and O. Yoshie, Eds. Springer Berlin / Heidelberg, 2003, vol. 2543, pp. 151–169.
- [19] D. Billington, V. Estivill-Castro, R. Hexel, and A. Rock, "Non-monotonic reasoning for localisation in robocup," in *Proc. of the 2005 Australasian Conf. on Robotics and Automation*, 2005.
- [20] D. Billington, V. Estivill-Castro, R. Hexel, and A. Rock, "Using temporal consistency to improve robot localisation," in *RoboCup 2006: Robot Soccer World Cup X*, ser. LNCS, G. Lakemeyer, E. Sklar, D. Sorrenti, and T. Takahashi, Eds. Springer Berlin / Heidelberg, 2007, vol. 4434, pp. 232–244.
- [21] G. Antoniou, D. Billington, and M. Maher, "On the analysis of regulations using defeasible rules," in *Proc. of the 32nd Annual Hawaii Intl Conf. on System Sciences (HICSS)*. Maui, HI, USA: IEEE Press, 1999, p. 7.
- [22] G. Governatori, M. Dumas, A. H. M. ter Hofstede, and P. Oaks, "A formal approach to protocols and strategies for (legal) negotiation," in *Proc. of the 8th intl conf. on Artificial intelligence and law*. New York: ACM Press, 2001, pp. 168–177.
- [23] G. Antoniou, "Nonmonotonic rule systems on top of ontology layers," in *The Semantic Web ISWC 2002*, ser. LNCS, I. Horrocks and J. Hendler, Eds. Springer Berlin / Heidelberg, 2002, vol. 2342, pp. 394–398.
- [24] N. Bassiliades, G. Antoniou, and I. Vlahavas, "Dr-device: A defeasible logic system for the semantic web," in *Principles and Practice of Semantic Web Reasoning*, ser. LNCS, H. Ohlbach and S. Schaffert, Eds. Springer Berlin / Heidelberg, 2004, vol. 3208, pp. 134–148.
- [25] G. Governatori and A. Rotolo, "Defeasible logic: Agency, intention and obligation," in *Deontic Logic in Computer Science*, ser. LNCS, A. Lomuscio and D. Nute, Eds. Springer Berlin / Heidelberg, 2004, vol. 3065, pp. 114–128.
- [26] G. Governatori and V. Padmanabhan, "A defeasible logic of policy-based intention," in *AI 2003: Advances in Artificial Intelligence*, ser. LNCS, T. Gedeon and L. Fung, Eds. Springer Berlin / Heidelberg, 2003, vol. 2903, pp. 414–426.
- [27] G. Governatori, A. Rotolo, and S. Sadiq, "A model of dynamic resource allocation in workflow systems," in *Proc. of the 15th Australasian database conf.*, vol. 27. Darlinghurst, Australia: Australian Computer Society, Inc., 2004.
- [28] G. Governatori, "Representing business contracts in ruleml," *Intl Journal of Cooperative Information Systems*, vol. 14(2-3), pp. 181–216, 2005.
- [29] G. Governatori, A. Rotolo, and G. Sartor, "Temporalised normative positions in defeasible logic," in *Proc. of the 10th intl conf. on Artificial intelligence and law*. NY: ACM Press, 2005, pp. 25–34.
- [30] D. Billington, "Propositional clausal defeasible logic," *LNCS*, vol. 5293, pp. 34–47, 2008, springer.
- [31] E. F. Hill, *Jess in Action: Java Rule-Based Systems*. Manning Publications Co. Greenwich, CT, USA, 2003.
- [32] M. Jang and J.-C. Sohn, "Bossam: An extended rule engine for owl inferencing," in *Rules and Rule Markup Languages for the Semantic Web*, ser. LNCS, G. Antoniou and H. Boley, Eds. Springer Berlin, Heidelberg, 2004, vol. 3323, pp. 128–138.
- [33] T. Myers and R. G. Dromey, "From requirements to embedded software - formalising the key steps," in *Proc. 20th Australian Software Engineering Conf. ASWEC*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 23–33.
- [34] S. Shlaer and S. J. Mellor, *Object lifecycles : modeling the world in states*. Yourdon Press, Englewood Cliffs, N.J., 1992.
- [35] S. Zafar and R. G. Dromey, "Integrating safety and security requirements into design of an embedded system," in *Asia-Pacific Software Engineering Conf.*, 2005, pp. 629–636.
- [36] V. Estivill-Castro, R. Hexel, and D. Rosenblueth, "Failure modes and effect analysis (FMEA) and model-checking of software for embedded systems by sequential scheduling of vectors of logic-labelled finite-state machines," in *Proc. 7th Intl IET System Safety Conf.*, Edinburgh, UK, October 2012.
- [37] V. Estivill-Castro, R. Hexel, and D. Rosenblueth, "Efficient modelling of embedded software systems and their formal verification," in *Proc. 19th Asia-Pacific Software Engineering Conf. (APSEC)*, Hong Kong, Dec. 2012.