

# Integrating Requirements: The Behavior Tree Philosophy

Kirsten Winter      Ian J. Hayes  
School of ITEE,  
The University of Queensland,  
Brisbane, Qld 4072, Australia  
email: kirsten, ianh@itee.uq.edu.au

Robert Colvin  
The Queensland Brain Institute,  
The University of Queensland,  
Brisbane, Qld 4072, Australia  
email: robert@itee.uq.edu.au

*This paper is dedicated to the memory of our friend and colleague  
Professor R. Geoff Dromey (1946 – 2009).*

**Abstract**—Behavior Trees were invented by Geoff Dromey as a graphical modelling notation. Their design was driven by the desire to ease the task of capturing functional system requirements and to bridge the gap between an informal language description and a formal model. Vital to Dromey’s intention is the idea of incrementally building the model *out of its building blocks, the functional requirements*. This is done by graphically representing each requirement as its own Behavior Tree and incrementally merging the trees to form a more complete model of the system.

In this paper we investigate the essence of this constructive approach to creating a model in general notation-independent terms and discuss its advantages and disadvantages. The result can be seen as a framework of rules and provides us with a semantic underpinning of requirements integration. Integration points are identified by examining the (implicit or explicit) preconditions of each requirement. We use Behavior Trees as an example of how this framework can be put into practise.

**Keywords**—Requirements, modelling, analysis, integration, Behavior Tree

## I. REQUIREMENTS

In engineering, a set of requirements is understood as what is needed as a system’s behaviour from the customer’s perspective. To develop a specification from requirements different activities can be distinguished [1], [2]:

- elicitation (gathering, understanding, reviewing, and articulating the needs of the customer),
- analysis (checking for consistency and completeness),
- specification (building a model) and verification (checking the model against requirements), and
- validation (making sure the specification reflects the intention of the customer).

In the context of this paper we are not concerned with issues of elicitation of requirements. We assume requirements are given as natural language descriptions and focus on *modelling* of the requirements (i.e., building a specification), *analysis* and *verification* of the resulting model.

©©2010 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

SEFM 2010, 13-18 September 2010 Pisa, Italy 978-1-4673-0269-2/12/\$31.00

© 2012 IEEE

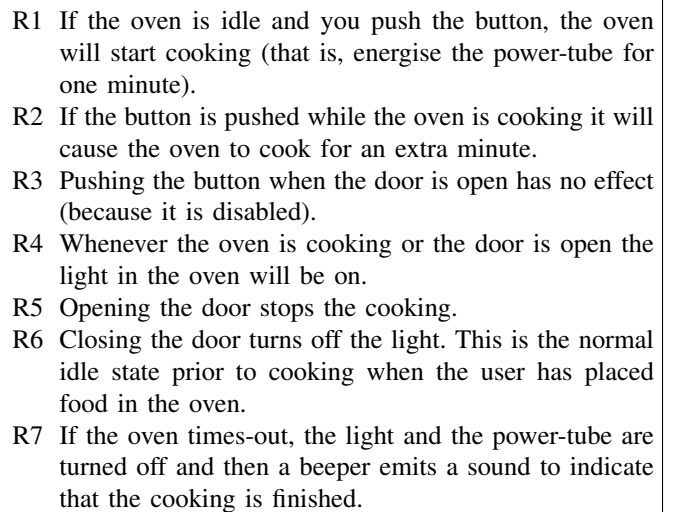
- 
- R1 If the oven is idle and you push the button, the oven will start cooking (that is, energise the power-tube for one minute).
  - R2 If the button is pushed while the oven is cooking it will cause the oven to cook for an extra minute.
  - R3 Pushing the button when the door is open has no effect (because it is disabled).
  - R4 Whenever the oven is cooking or the door is open the light in the oven will be on.
  - R5 Opening the door stops the cooking.
  - R6 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.
  - R7 If the oven times-out, the light and the power-tube are turned off and then a beeper emits a sound to indicate that the cooking is finished.

Figure 1. The requirements of a Microwave Oven

The requirements are usually given as a set of individual statements:

$$R = \{r_1, \dots, r_n\}.$$

As an example, we reproduce the requirements for the Microwave Oven as published in [3] in Figure 1.

The set  $R$  serves as an input for developing a model  $\mathcal{M}$  of the system. *Modelling* can be informal, i.e., using a notation for which the meaning of its syntactic elements is not formally defined. This can serve well for the purpose of documentation and clarification. However, creating a formal model (using a formal notation) has the benefit that the analysis of the model (and its properties) as well as the development of an implementation from the model can be supported by tools and formal techniques.

The model needs to be *verified* against the requirements to ensure their intention is preserved. Verification can be done on an informal basis, if  $\mathcal{M}$  and  $R$  are not given formally. If  $\mathcal{M}$  is a formal model and the requirements are formalised (individually) then validation can be pursued by verifying

that  $\mathcal{M}$  satisfies  $R$ :

$$\mathcal{M} \models R.$$

The model  $\mathcal{M}$  can also serve as a basis for the analysis of  $R$ , such as checking consistency and completeness. If changes to the requirements are made (e.g., due to the results of the analysis) the model has to be adjusted to accommodate these and once again the new model has to be validated and analysed.

In the following we are aiming at a reasoning approach that will hold for informal as well as formal models, for some notion of satisfiability. In the formal approach satisfiability can be checked on the basis of formal techniques and potentially using tool support.

We structure the rest of the paper as follows: Section II summarises aspects of requirements modelling which then leads on to a constructive approach for modelling by building the model out of the requirements. Integrating the requirements is discussed in Section III. In Section IV we introduce Behavior Trees as an example notation that supports the integration of requirements and explain the integration steps by means of an example in Section V. In Section VI we provide the basis for a formal framework for integration as a set of abstract rewriting rules and show how these can be interpreted on the example of Behavior Trees. Section VII concludes our work with a discussion.

## II. REQUIREMENTS MODELLING

Models can be created in various ways. The standard approach is to posit a model  $\mathcal{M}$  based on the understanding and interpretation of (a sub-set of) the requirements. This often goes hand-in-hand with a partitioning of the requirements based on the components that are identified as parts of the system. Subsequently, the modeller has to show or prove that  $\mathcal{M}$  meets the individual functional requirements  $r_i$ . This is summarised as the “posit and prove” approach.

The advantage of this approach is that  $\mathcal{M}$  might (depending on the capability of the modeller) have a certain elegance and neatness. It might contain suitable abstractions that support the subsequent analysis and it often already includes sensible design decisions.  $\mathcal{M}$  is most likely already in the shape of a model that can be used for the further development of the system.

The disadvantages of this approach, however, can be summarised by the following points.

- 1) Checking  $\mathcal{M} \models r_i$  for each  $i$  is expensive for a large set  $R$ . In a formal approach to verification this requires one to formalise each  $r_i$  and to provide a proof that each  $r_i$  is satisfied. In an informal approach validating  $\mathcal{M}$  can be approximated by testing, which is known to be incomplete and time consuming, thus expensive as well.
- 2) *Traceability* of individual requirements within  $\mathcal{M}$  may be lost for two reasons: firstly, abstractions built into

$\mathcal{M}$  cause the loss of the original detail and secondly, the design often combines behaviours based on components rather than based on the client-focussed functionality which usually defines the structure of the requirements.

- 3) Due to the lack of traceability it may be difficult to cope with *requirements change*. If  $r_i$  changes to  $r'_i$ , it is unclear how this affects the model  $\mathcal{M}$ .
- 4) For the customer, who is ultimately involved in the validation of the model, interpreting and understanding the model  $\mathcal{M}$  becomes more challenging the further the structure of  $\mathcal{M}$  is removed from the structure of the requirements. Moreover, the use of a specialised formal notation that is not derived from the customers domain may be a barrier to understanding.

These disadvantages may be addressed by taking a *constructive* approach to building  $\mathcal{M}$  instead of the “posit and prove” approach, that is, by *constructing  $\mathcal{M}$  out of  $R$*  as advocated by Geoff Dromey [3], [4]. The advantage of design and abstraction may be lost in this construction; however, as we will discuss in Section III-B, the best of both worlds may be possible. We have taken the approach of interpreting requirements as specifying *behaviour*, as distinct from specifying a condition, such as a predicate. The predicate-based approach has been considered as the basis of constructing models by others [5], [6], [7], [8], [9], but results in models that lack the advantage of visual control flow as exemplified by graphical languages and process algebras.

The individual requirements collected in  $R$  typically describe (desired) behaviour. For a model that captures each  $r_i$  as a predicate it suffices to build the conjunction of all  $r_i$  to express that *all* requirements are satisfied by the model (see e.g., the work in [10] and [6]). However, capturing requirements that describe a sequence of steps as predicates is not a trivial task and often leads to a model that is not easily understood by the customer. Alternatively, we might attempt to construct a trace-based model out of the set of individual requirements  $r_i$ . For the sake of presentation, assume the behaviour requested by  $r_i$  is given as a *trace* of actions. A naive approach might use parallel composition to represent the conjunction of all traces  $r_i$ :

$$\mathcal{M} = r_1 \parallel r_2 \parallel \dots \parallel r_n.$$

For nontrivial  $r_i$  this is clearly not suitable, as requirements are interdependent. This interdependency may be accounted for by giving each requirement  $r_i$  a precondition,  $p(r_i)$ , which is the condition required to ensure  $r_i$  behaves as expected. We abbreviate  $p(r_i)$  as  $p_i$ .<sup>1</sup> If  $p_i$  has not been established the execution of  $r_i$  would lead to a behaviour that would not match the intention of requirements in  $R$ .

<sup>1</sup>The term *precondition* is used here in the specific context of Behavior Trees and comes with a particular interpretation that might not coincide with interpretations given to the term in other contexts.

The precondition can be understood as a condition over the system's state that has to be established by (the execution of) a behaviour before another behaviour is enabled. Preconditions are often implied or only partially stated by the requirements because each requirement is not stand-alone but has to be understood in the context of the complete set of requirements. For instance, the behaviour of requirement R5 in Figure 1 has the implicit precondition that the door is closed and the oven is cooking.

To convey our concept we use the notation  $p_i \rightsquigarrow r_i$  to mean that when the behaviour that establishes the condition  $p_i$  has occurred, then  $r_i$  may occur, i.e.,  $p_i$  enables  $r_i$ . Let us furthermore assume the preconditions are made explicit in  $R$ .

$$R = \{p_1 \rightsquigarrow r_1, p_2 \rightsquigarrow r_2, \dots, p_n \rightsquigarrow r_n\}$$

This gives the constructed model

$$\mathcal{M} = (p_1 \rightsquigarrow r_1) \parallel (p_2 \rightsquigarrow r_2) \parallel \dots \parallel (p_n \rightsquigarrow r_n).$$

Assuming it is possible to define  $\rightsquigarrow$  and track it appropriately, this model addresses points 1, 2 and 3, but it does have some fundamental problems:

- It is difficult to comprehend the overall system behaviour, as the flow of control is not represented by the structure of  $\mathcal{M}$  but is hidden in the parallel construction.
- There is no notion of choice between behaviours, i.e., in a state in which both conditions  $p_i$  and  $p_j$  hold both behaviours  $r_i$  and  $r_j$  can occur (in parallel), but the intended meaning may be for one or the other but not both behaviours to occur.

Clearly, constructing a reasonable model from the requirements is a nontrivial task.

### III. INTEGRATING REQUIREMENTS

Let us examine *integrating the requirements*, in order to gain a model that makes control flow more explicit: instead of only using the parallel operator to combine requirements we also describe rules for incorporating sequential and parallel composition, nondeterministic choice between alternatives, and iteration. These rules are an attempt at a simple framework and are incomplete. We will refine them in Section VI.

*Sequential Composition:* If behaviour  $r_i$  establishes the precondition of another behaviour  $r_j$ , denoted as  $r_i \rightsquigarrow p_j$ , then one possible way to integrate the requirements,  $p_i \rightsquigarrow r_i$  and  $p_j \rightsquigarrow r_j$ , is as a single sequential behaviour,  $p_i \rightsquigarrow (r_i; r_j)$ .

Rule (1) describes this step and it reads as follows. The hypothesis above the line specifies the necessary condition to apply the rule, i.e.,  $p_i \rightsquigarrow r_i$  establishes  $p_j$ . The conclusion below the line shows the transformation relation  $\rightarrow$ . On the left hand side of  $\rightarrow$  we can find the original term, the tuple

$(p_i \rightsquigarrow r_i, p_j \rightsquigarrow r_j)$ , which is transformed into the single term shown on the right hand side of  $\rightarrow$ .

$$\frac{p_i \rightsquigarrow r_i \rightsquigarrow p_j}{(p_i \rightsquigarrow r_i, p_j \rightsquigarrow r_j) \rightarrow (p_i \rightsquigarrow (r_i; r_j))} \quad (1)$$

Figure 2 depicts this rule graphically: the behaviours  $r_i$  and  $r_j$  are shown as bars, paraphrasing that their execution proceeds from left to right. On the left side of the behaviour the preconditions required for execution are given, e.g.,  $p_i$ , and to the right the condition that is established after executing the behaviour is shown. Linking the two bars into one long bar symbolises the sequential composition of the behaviours.

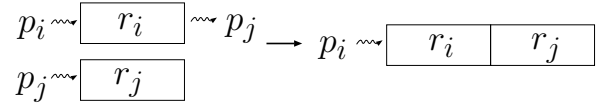


Figure 2. Rule (1) for sequential composition graphically

*Branching Composition:* If both requirements  $r_i$  and  $r_j$  have the same precondition, i.e.,  $p_i = p_j$ , then both behaviours can occur when this precondition has been established. In this case the modeller must use their domain knowledge to decide if parallel ( $\parallel$ ) or alternative ( $\oplus$ ) behaviour models the requirements appropriately. Rule (2) captures both cases, letting the symbol  $\oplus$  stand for either  $\parallel$  or  $\oplus$ .

$$\frac{p_i = p_j}{(p_i \rightsquigarrow r_i, p_j \rightsquigarrow r_j) \rightarrow (p_i \rightsquigarrow (r_i \oplus r_j))} \quad (2)$$

This rule is depicted in Figure 3. Two requirements with the same precondition (on the left) can be integrated into a construct in which the common precondition is followed by both behaviours combined in the tree-like fashion of either parallel or choice (on the right).

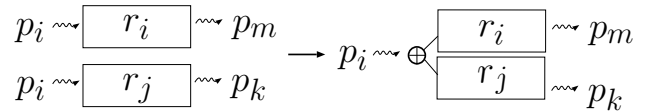


Figure 3. Rule (2) for branching composition graphically

*Iteration:* If a requirement  $r$  with precondition  $p$ , establishes its own precondition, i.e.,  $r$  establishes  $p$ , then we extend the behaviour with iteration such that the flow of control loops back to the beginning once behaviour  $r$  has terminated. The following rule captures the introduction of loops. The notation  $\mu x.r;x$  denotes a recursive term in which  $x$  is replaced by  $\mu x.r;x$  when the term is unfolded into behaviour.

$$\frac{p \rightsquigarrow r \rightsquigarrow p}{p \rightsquigarrow r \rightarrow (p \rightsquigarrow (\mu x.r;x))} \quad (3)$$

Graphically, this rule can be depicted as in Figure 4. A requirement that establishes its own precondition (on the left), leads to iterated behaviour (depicted using a looping back arrow on the right).

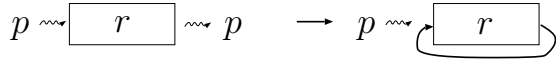


Figure 4. Rule (3) for iteration graphically

The rules we have presented in this section are simple cases of more general rules that are presented in Section VI.

#### A. Degrees of freedom in integration

In general for any requirement  $p_i \rightsquigarrow r_i$  there might be many places with which it may be integrated, since the precondition might be established more than once through other requirements. This might imply ambiguities of the requirements (see also Section III-B). Some requirements may need to be integrated multiple times. Moreover, the order in which the integration takes place ultimately determines the possibilities of integration. The modeller must also decide whether to interpret the operator  $\oplus$  as parallel or alternative composition.

All these choices define the degree of freedom in the modelling phase, and with this the *modelling space*. They will ultimately impact on the form of the full model. They depend on the modeller’s understanding and interpretation of the given requirements and may have to be discussed with the customers to ensure that the correct intention is captured.

The modelling process in the constructive integration approach is therefore human-centered because many decisions have to be made by the modeller in consultation with the customer. However, the process is more constrained than the “posit and prove” approach, and tool support can potentially be provided to rule out integrations that while syntactically valid (the precondition is established) lead to inconsistencies in the resulting model.

#### B. Full integration

The process of integration continues until a single structure remains which constitutes the full model. If a requirement is not able to be integrated then this indicates either that it is incomplete or inconsistent with the other requirements in that its precondition is never satisfied. We refer to this as a *defect* in  $R$  that must be corrected by consultation with the client. The integration process might also reveal ambiguity in the requirements  $R$  which needs to be resolved with the client in order to create a model that captures their intention. Detection of defects and ambiguities in the requirements are valuable feedback for the client.

Assume that all requirements in  $R$  have been integrated (possibly after fixing defects and resolving ambiguities in the requirements). The resulting model “satisfies” the requirements in the sense that the behaviour of each occurs

explicitly as part of the model. Checking  $\mathcal{M} \models r_i$  is reduced to verifying that  $r_i$  has been successfully integrated and we have thus addressed the problem of validating a model (point 1 in Section II). Traceability (point 2) can be achieved straightforwardly; since each requirement  $r_i$  maintains its integrity within the model, it may be syntactically tagged with the corresponding label in the client’s original requirements document. Handling requirements change (point 3) is similarly supported due to the localised nature of requirements in the constructed model. The readability of the model by the client (point 4) is improved as the structure of the requirements is preserved in the structure of the model.

The main disadvantage, however, is the converse of the advantage of standard modelling approaches. The model is not abstract. It reflects the structure of the requirements document which is most likely not the desired structure of the system. Hence the initial model that is constructed from the requirements has to be transformed to incorporate design.

For example, we might be interested in a model with a component-based structure that highlights the behaviour of each individual component and the interaction between components. This requires restructuring the initial model into a model consisting of parallel behaviours each of which specifies the behaviour of one component only. Consequently, we have to introduce communication mechanisms between components, in order to maintain the correct sequencing of steps. We argue that these refactoring steps can be undertaken in a structured fashion and (in the case of a formal model) can be potentially supported by tools and formal techniques, once an initial model has been created and validated. Most importantly, defects in the requirements have been addressed and ambiguities resolved beforehand.

## IV. BEHAVIOR TREES

In Section II we presented rules for a constructive approach of building a system model out of its requirements. The approach of *Behavior Engineering*, as it was termed by Geoff Dromey, was created out of this core idea. Underlying this philosophy is a graphical notation that takes the shape of trees and thus accommodates the proposed integration steps. In this section we describe the notation of Behavior Trees and in particular show how the *syntax-directed* integration approach of Geoff Dromey [3] relates to the abstract integration we outlined in the previous section.

#### A. Syntax of Behavior Trees

The notation of Behavior Trees resembles some aspects of flowcharts. A flowchart is a type of diagram that can be used to represent an algorithm or a process. It depicts the steps of the algorithm as nodes of various kinds, and their order of their execution by connecting arrows. Behavior Trees have in common with flowcharts the concept of nodes (of various types) to symbolise behaviour or conditions, and connecting arrows to denote control flow.

Behavior Trees aim at the specification of a system's behaviour. Beyond simple sequential and alternative control flow Behavior Trees allow the user to model concurrent behaviour. The concept of conditioned alternative is extended to a more general nondeterministic choice (guarded or unguarded). Moreover, each single behaviour, as specified in a node, is dedicated to one particular component. That is, in a Behavior Tree the modeller can piece together behaviours from various components in one tree of actions, thus taking a global perspective on the system's behaviour. These features enable the modeller to capture sets of (individual) requirements and integrate them into a model of the system as a whole.

Figure 5 shows the syntax for basic node types. (In this paper we present the core of the notation only; for a full description we refer the reader to [11]). Each node carries

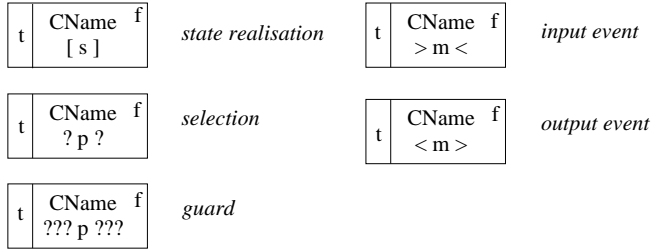


Figure 5. Basic node types

the component name CName to indicate which component in the system is subject of the action or condition.

- 1) *State realisation* denotes an update on the state of component CName to a new state  $s$ .
- 2) *Selection* specifies a condition under which the control flow continues; if condition  $p$  is not satisfied the execution terminates at this point (i.e., we have non-blocking behaviour).
- 3) *Guard* models the concept of a blocking wait. If  $p$  holds the flow of control continues, otherwise the behaviour is blocked and the system waits until  $p$  holds.
- 4) *Input and output events* model communication between components where the component in the output event node is the *sender* of message  $m$  and the component in the input event node is the *reader* of  $m$  (multicast communication with several readers is possible).

We say that two nodes are *matching* if they are identically labelled, i.e., if they refer to the same component, are of the same type, and specify the same new state/condition/message.

Nodes also carry a *tag*,  $\tau$ , which specifies the (set of) requirements label(s) to indicate from which requirement this node was derived to provide traceability of individual requirements in the model.

A set of flags,  $f$  in a node, allows the modeller to manipulate the control flow. For instance, the *reversion* flag, marked by '^', leads to a looping behaviour back to the closest matching ancestor node and all behaviour started after the matching ancestor node is terminated. The reversion flag can only be set for leaf nodes. Another example is the *thread kill* flag, marked by '--'. It has the effect of killing the thread that starts with the matching node. The *synchronisation* flag, marked by '=', enforces the control flow to wait until all other synchronisation points (matching nodes that also have the synchronisation flag set) are reached.

The control flow is syntactically specified by the arrows between the nodes. A single arrow that links two nodes denotes sequential composition. Nodes with two or more outgoing arrows indicate a branching flow of control. Branching can either denote alternative or parallel composition of behaviour. Figure 6 shows the syntax of the two concepts.

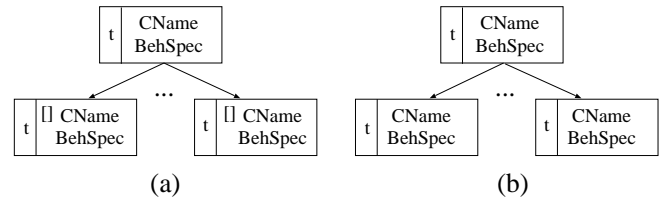


Figure 6. Branching control flow: (a) alternative; (b) parallel

Note that the leading nodes of alternative behaviours have to be either all selections (i.e., of the form  $?p?$ ) or none of them selections. Semantically, it provides us with a nondeterministic choice, and it can be additionally guarded.

The Behavior Tree notation has been formalised [12] by translating the notation into a variant of CSP,  $CSP_\sigma$  [13]. The process algebra  $CSP_\sigma$  allows interprocess communication through synchronisation, as well as mutable state (to model the components of the system). The meaning of a Behavior Tree may be interpreted via its operational semantics.

## B. Modelling with Behavior Trees

We provide some examples from the microwave oven as given in Figure 1 to demonstrate how the syntax of Behavior

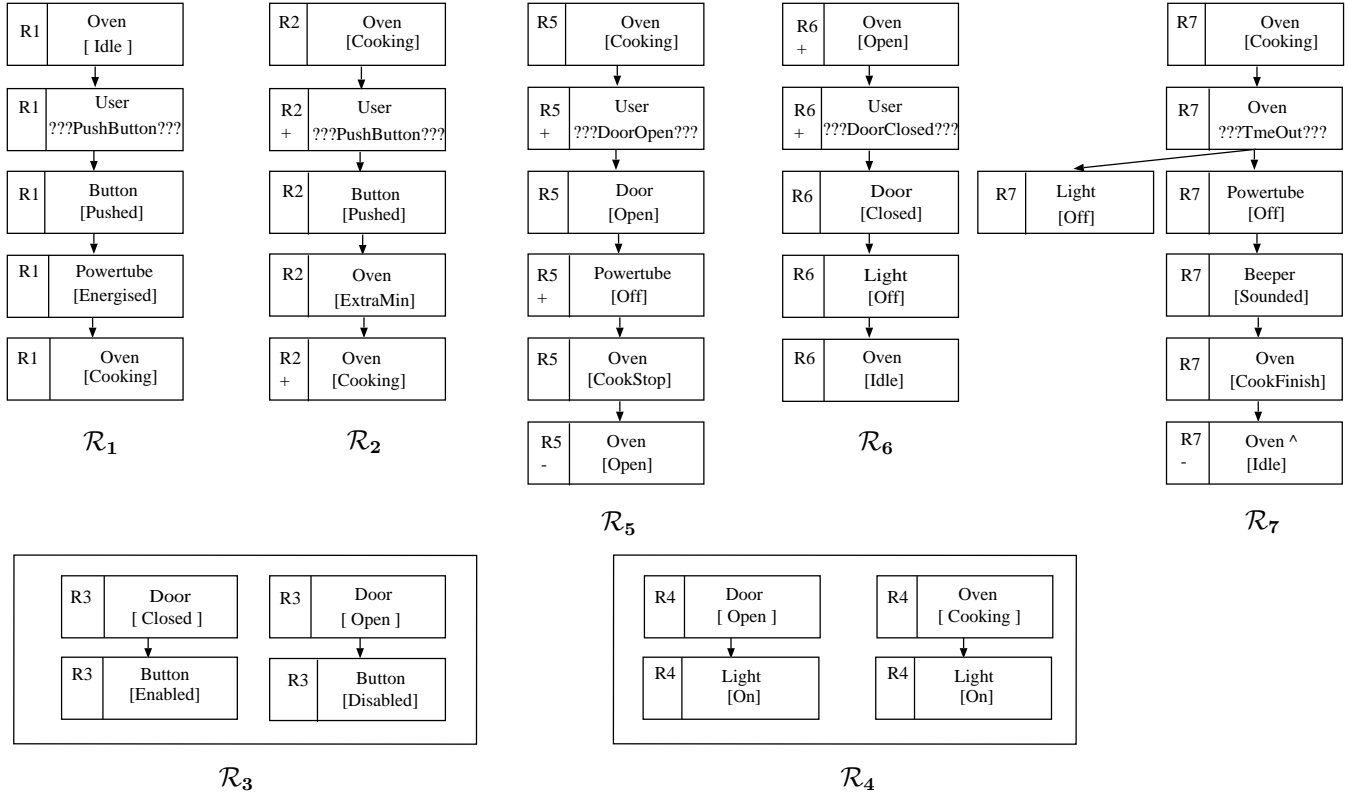


Figure 8. Individual Behavior Trees for each of the given requirements R1 - R7

Trees is utilised to capture individual requirements.

Consider requirement  $R_1$ :

*If the oven is idle and you push the button, the oven will start cooking (that is, energise the power-tube for one minute).*

To model  $R_1$  we create a Behavior Tree as shown in Figure 7. The first node models the condition that the `Oven` component has to be in the state `Idle`. The second node models the guard that is satisfied when and if component `User` has pushed the button – until the guard is satisfied the control flow cannot proceed past this point. As a consequence a sequence of further

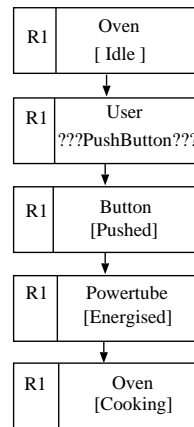


Figure 7. Behavior Tree for  $R_1$

state changes in other components occurs, namely in the `Button`, the `Powertube`, and in the `Oven` which starts cooking. We refer to a Behavior Tree that originates from a single requirement as an *individual requirement tree*.

The Behavior Tree in Figure 7 (as suggested by Geoff Dromey [3]) is close to a literal translation of the requirement (apart from adding the state change in component `Button` which was not explicit in the description). A notable decision was made however, to model the first node as a state realisation, where in fact a selection may have

sufficed, that is, the root node could possibly have been a test on whether the oven is idle. The reason for this choice is explained in the next section. Figure 8 shows how the other requirements can be captured by individual requirement trees (again these are taken from Geoff Dromey’s work [3]).

## V. INTEGRATING BEHAVIOR TREES

The Behavior Tree approach to integrating requirements (represented as Behavior Trees) is to look for common nodes through simple syntactic matching. As examples we demonstrate how to integrate some of the individual requirement trees from Section IV-B. In the following we denote a Behavior Tree that captures requirement  $R_i$  by  $\mathcal{R}_i$ .

Figure 9 shows the integration of  $\mathcal{R}_2$  and  $\mathcal{R}_5$ . Both sequences have the same the root node, `Oven [Cooking]`, which is the matching node, and we merge one of the Behavior Trees (say  $\mathcal{R}_5$ ) into the other ( $\mathcal{R}_2$ ) at this point. The root node is common to both  $\mathcal{R}_2$  and  $\mathcal{R}_5$  and hence is tagged with both  $R_2$  and  $R_5$  in the merged tree. This results in a branching construct which can be either alternative or parallel flow of control. In this case the modeller has to make a design decision which of those captures the intention of the requirements. In our case we want both behaviours to be a possibility. That is, if cooking for an extra minute ( $\mathcal{R}_2$ ) is requested it should not disable the functionality of opening the door to interrupt the cooking. Hence, we model the behaviours as being concurrent. The intermediate

result is the Behavior Tree that we denote as  $\mathcal{R}_{25}$ . (Note that further analysis will reveal safety violations due to race conditions between the two threads since any interleaving of the behaviour is possible. The model will have to be modified to solve this problem.)

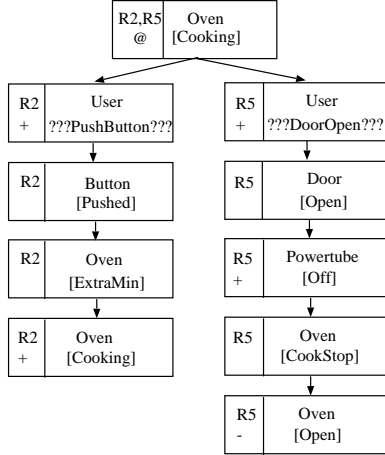


Figure 9. Integration of  $\mathcal{R}_2$  and  $\mathcal{R}_5$  into  $\mathcal{R}_{25}$

The individual requirement tree  $\mathcal{R}_6$  can be integrated with  $\mathcal{R}_1$  because  $\mathcal{R}_6$  establishes the situation where the `Oven` is `Idle`, which matches the root node of  $\mathcal{R}_1$ . This exact syntactic matching on nodes is why the root node of  $\mathcal{R}_1$  was modelled as a state realisation and not a selection (see Section IV-B). In the latter case, the matching would not occur and the trees would remain unintegrated. This would lead to a re-examination of the interpretation of  $\mathcal{R}_1$  and changing its root node to a state realisation.

The second behaviour of  $\mathcal{R}_4$  can be integrated with  $\mathcal{R}_1$  at the matching node `Oven [Cooking]`. However, we might decide that it is more suitable for the microwave oven if the light comes on as soon as the button is pushed, and consequently choose `Button [Pushed]` in  $\mathcal{R}_1$  as the point of integration.

The leaf node in requirements tree  $\mathcal{R}_2$  matches the root node of  $\mathcal{R}_2$ . That is, the behaviour of  $\mathcal{R}_2$  establishes its own precondition. This indicates the possibility for looping behaviour. A reversion flag ‘^’ added to  $\mathcal{R}_2$ ’s leaf node captures the iteration.

Figure 12 shows Geoff Dromey’s fully integrated Behavior Tree (from [3]). What can be seen in this example is that the concept of matching nodes provides support to locate possible points of integration. The modeller can then still deviate from the exact points or modify the Behavior Trees to achieve an integration that reflects his/her interpretation of the requirements. The potential ambiguities of integration stem from the requirements and are identified by the integrative approach.

#### A. Behavior Tree integration in the formal framework

The Behavior Tree approach to integration is to identify integration points through matching nodes. To relate this to our framework we must describe the meaning of  $r \rightsquigarrow p$  and  $p \rightsquigarrow r$ , and relate the syntactic integration to the integration rules in Section III.

In the Behavior Tree context, a precondition  $p$  is a predicate on the state of the components in the system, and a behaviour  $r$  is a sequence of operations on the components (tests and updates), and possibly communication between components. Hence,  $r \rightsquigarrow p$  holds if  $r$  modifies the components in some way that makes  $p$  true. Similarly,  $p \rightsquigarrow r$  if the state of the components identified by  $p$  enables  $r$ .

In the following we note that  $\mathcal{R}$  contains both the precondition and the required behaviour, that is  $\mathcal{R} = p \rightsquigarrow r$ . Furthermore, we reduce the precondition  $p$  to one single node, the *matching* root node. For instance, the precondition part in  $\mathcal{R}_1$  is that the oven is idle ( $P_1$ ), while the rest of the requirement specifies the sequence of the button being pressed and the button state, power tube, and oven being modified ( $S_1$ ). We have  $\mathcal{R}_1 = P_1 \rightsquigarrow S_1$ . It becomes quite clear that  $\mathcal{R}_6$ , i.e., the leaf node of  $\mathcal{R}_6$ , establishes  $P_1$ , hence,  $\mathcal{R}_6 \rightsquigarrow P_1$ . The requirements may be integrated at the leaf node of  $\mathcal{R}_6$ .

More specifically, the following relationship relates the structure of individual requirement trees to the requirements framework we have outlined (we assume that behaviour  $p_i \rightsquigarrow r_i$  is modelled by a Behavior Tree  $\mathcal{R}_i$ , which is usually a sequence).

$$leaf(\mathcal{R}_i) = root(\mathcal{R}_j) \implies r_i \rightsquigarrow p_j$$

The condition  $r_i \rightsquigarrow p_j$  is the antecedent of Rule (1), and hence the syntactic matching of a root node to a leaf node, as with  $\mathcal{R}_1$  and  $\mathcal{R}_6$  in our example, justifies integrating them via a sequential composition of Behavior Trees.

The integration of two Behavior trees to form a branching tree, as with integrating  $\mathcal{R}_2$  and  $\mathcal{R}_5$ , has the following relationship:

$$root(\mathcal{R}_i) = root(\mathcal{R}_j) \implies p_i = p_j$$

That is, if the root nodes are identical, the preconditions are identical. The condition  $p_i = p_j$  is exactly the antecedent of Rule (2). The resulting integrated Behavior Tree introduces a branching construct after the matching root nodes, i.e., following the notation used in Rule (2),  $p_i \rightsquigarrow (r_i \oplus r_j)$ .

Finally, we consider iteration, which is achieved in Behavior Trees through reversion tags. As described in the last section, reversion is added to the tree when a matching node of a leaf appears as a root. For example, reversion was added to  $\mathcal{R}_2$ .

$$root(\mathcal{R}) = leaf(\mathcal{R}) \implies r \rightsquigarrow p \wedge p \rightsquigarrow r$$

That is, the behaviour  $r$  of tree  $\mathcal{R}$  establishes its own precondition if it finishes where it started. The condition

gives the context for applying Rule (3). As a result the behaviour of  $\mathcal{R}$  iterates back to  $r$ .

Of course, the relationships listed above will not always hold. The Behavior Tree philosophy is that they hold sufficiently often to form the basis of constructive modelling (without needing to explicitly collect all preconditions). This notion provides us with rules for a purely *syntactic* integration. The situations where the relationships do not hold are candidates for ambiguity or inconsistencies in the original requirements, and hence worth identifying.

We have shown how syntactic matching works in the simple case of integrating two individual requirement trees in the Behavior Tree notation. However, as the integration process continues, the trees involved become larger and will have more complex structure. In the next section we provide a more general version of the rules to handle full integration.

## VI. MODELLING WITH TREE STRUCTURES

In this section we generalise the earlier rules to accommodate more complex integrations. This generalisation leads to *tree rewriting rules* which allow us to *incrementally* integrate a number of requirements. Assume therefore that the requirements are given as a set of tree terms (e.g., abstract syntax trees of expressions representing behaviour). Syntactically, trees can have the following form where ‘ $\emptyset$ ’ denotes the empty tree and ‘;’ the sequential composition of a single node and a tree.  $\mu x.Tree$  denotes the recursive tree term and  $x$  an identifier which is bound by the  $\mu$  operator, i.e., it is only defined in the context of a tree  $t$  which occurs in the term  $\mu x.t$ .

$$\begin{aligned} Tree & ::= \emptyset \mid Node;Tree \mid x \\ & \mid Tree \parallel \dots \parallel Tree \\ & \mid Tree \parallel \dots \parallel Tree \\ & \mid \mu x.Tree \end{aligned}$$

Note that with the introduction of recursive loops the structure is no longer strictly a tree, i.e., our concept of *Tree* as defined above has a special form that suits our purposes.

In the following let  $t$  and  $t_i$  be (sub-)trees,  $n$  a node,  $s$  and  $s_i$  sequences of nodes and  $\langle \rangle$  the empty sequence (of nodes). The concatenation of sequences is marked by ‘ $\wedge$ ’, and with  $\oplus$  we denote an operator that can be either  $\parallel$  or  $\parallel$ .

Tree terms can be graphically depicted as directed, rooted trees for which we can define the notion of *paths* of a tree as follows. Each path is a sequence of nodes.

$$\begin{aligned} paths(\emptyset) & = \{\langle \rangle\} \\ paths(n; t) & = \{s : paths(t) \cdot \langle n \rangle \wedge s\} \cup \{\langle \rangle\} \\ paths(t_1 \oplus \dots \oplus t_n) & = paths(t_1) \cup \dots \cup paths(t_n) \\ paths(\mu x.t) & = paths(t). \end{aligned}$$

Note that paths can not contain the identifier  $x$  and as a consequence do not include loops. We also assume that the

sets of paths of different sub-trees are disjoint since each node can be distinguished by tagging it with its location in the tree (even if nodes may be syntactically identical).

Within a tree we can use indexing by a path to refer to a particular point in the tree where a particular sub-tree is located. Indexing on trees is defined inductively as follows.

$$\begin{aligned} t[\langle \rangle] & = t \\ (\langle n \rangle; t)[\langle n \rangle \wedge s] & = t[s] \\ (t_1 \oplus \dots \oplus t_n)[s] & = t_i[s] \quad \text{if } s \in paths(t_i) \\ (\mu x.t)[s] & = t[s] \end{aligned}$$

### A. Tree rewriting rules

To define the rules for integrating tree terms we use the notion of *substituting* sub-trees in a (super-) tree. Let  $\mathcal{T}$  be a tree and  $s$  a path from the root node of  $\mathcal{T}$  down to some node within  $\mathcal{T}$ . We denote with  $\mathcal{T}[s \mapsto t]$  the tree in which the path  $s$  leads to a point at which we substitute a sub-tree (possibly the empty tree  $\emptyset$ ) with the new sub-tree  $t$ . That is, we insert  $t$  at a particular point in  $\mathcal{T}$ , namely the point to which  $s$  leads. This leads to the following inductive definition. Assume  $s_i \in paths(t_i)$  as above then

$$\begin{aligned} \mathcal{T}[\langle \rangle \mapsto t'] & = t' \\ (n; t)[\langle n \rangle \wedge s \mapsto t'] & = n; (t[s \mapsto t']) \\ (t_1 \oplus \dots \oplus t_i \dots \oplus t_n)[s \mapsto t'] & = t_1 \oplus \dots \oplus t_i[s \mapsto t'] \dots \oplus t_n \\ & \quad \text{if } s \in paths(t_i) \\ (\mu x.t)[s \mapsto t'] & = \mu x.t[s \mapsto t'] \end{aligned} \tag{1}$$

This tree notation allows us now to generalise Rules (1) and (2) in Section III into a single rule. If a path  $s$  in the tree  $\mathcal{T}$  establishes the precondition of a tree  $t_2$ , then  $t_2$  can be integrated into  $\mathcal{T}$  by substituting sub-tree  $t_1 = \mathcal{T}[s]$  by a branching structure  $t_1 \oplus t_2$ . The modeller chooses either parallel or nondeterministic choice between the sub-tree  $t_1$  and the behaviour  $t_2$  and graft the branching structure at the end of path  $s$ . This is formalised in the following Rule (4).

$$\frac{p \rightsquigarrow s \rightsquigarrow p_2 \quad s \in paths(\mathcal{T}) \quad t_1 = \mathcal{T}[s]}{(p \rightsquigarrow \mathcal{T}, p_2 \rightsquigarrow t_2) \rightarrow p \rightsquigarrow \mathcal{T}[s \mapsto (t_1 \oplus t_2)]} \tag{4}$$

Note that this syntactic operation preserves the super-structure  $\mathcal{T}$  at all other places.

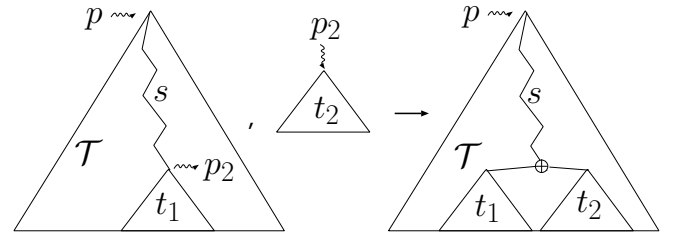


Figure 10. Rule 4 for integrating tree structures assuming  $s \rightsquigarrow p_2$

Figure 10 graphically depicts the integration of tree structures. Here we assume a top-to-bottom flow of control. Path



$s$  is followed by some sub-tree  $t_1$ . Note that the leaf nodes of this sub-tree line up with the leaf nodes of the super-structure  $\mathcal{T}$  (the outermost triangle). Integration with  $t_2$  results again in a sub-tree within  $\mathcal{T}$  (shown on the right) which has further branching at the point of integration. Note that  $t_2$  might be either a simple sequence or a tree structure.

Rule (4) collapses to the case for simple sequential composition (Rule (1)) by choosing  $\mathcal{T} = r_i$ ,  $s = r_i$ ,  $t_2 = r_j$  and  $t_1 = \emptyset$ , and choosing  $\oplus$  as the parallel operator  $\parallel$ . On the right-hand side of the transition we have  $r_i; (\emptyset \parallel r_j)$ , in which case we may eliminate the  $\emptyset$  to obtain  $r_i; r_j$ .

Rule (4) also collapses to Rule (2) (choosing  $s = \langle \rangle$  and  $p = p_j$ ), and is more general in that it allows the integration of  $r_j$  into the middle of a behaviour  $r_i$  (if  $s \neq \langle \rangle$ ).

*Iteration:* A second tree rewriting rule introduces iteration into a tree structure. Assume a tree  $\mathcal{T}_1$  in which the path  $s_1$  leads to the sub-tree  $\mathcal{T}_2$  and establishes precondition  $p_2$ . In  $\mathcal{T}_2$  the path  $s_2$  which leads to a leaf node of  $\mathcal{T}_2$ , also establishes precondition  $p_2$ . Then we can replace  $\mathcal{T}_2$  with the recursive term  $\mu x. \mathcal{T}_2[s_2 \mapsto x]$  that loops back from the leaf node that is reached via paths  $s_2$  to the root of  $\mathcal{T}_2$ .

$$\frac{\begin{array}{l} p \rightsquigarrow s_1 \rightsquigarrow p_2 \quad p_2 \rightsquigarrow s_2 \rightsquigarrow p_2 \\ s_1 \in \text{paths}(\mathcal{T}_1) \quad \mathcal{T}_2 = \mathcal{T}_1[s_1] \\ s_2 \in \text{paths}(\mathcal{T}_2) \quad \mathcal{T}_2[s_2] = \emptyset \end{array}}{p \rightsquigarrow \mathcal{T}_1 \rightarrow p \rightsquigarrow \mathcal{T}_1[s_1 \mapsto \mu x. \mathcal{T}_2[s_2 \mapsto x]]} \quad (5)$$

Note that we have introduced a binding, and scope, to the recursive variable  $x$ . The name must be chosen such that it is unique in the super-structure  $\mathcal{T}_1$ , i.e., a fresh variable name for each recursive term is necessary. Figure 11 graphically depicts this step that results in a looping tree structure.

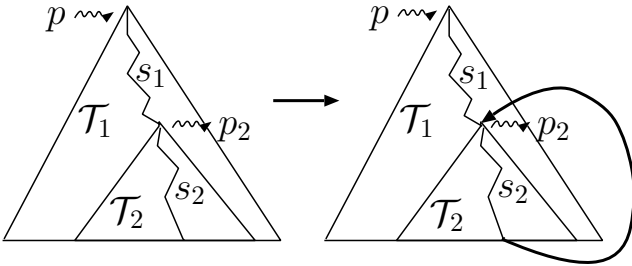


Figure 11. Rule (5) for iteration graphically

Note that any integration into a recursive sub-tree  $\mathcal{T}_2$  (i.e., within  $s_2$ ) will be part of the recursion. Thus, the effect of this step must be carefully checked, particularly if concurrency is introduced. Ideally, recursion is introduced last during the integration process so that side-effects on other parts of the (integrated) tree are visible.

### B. Relating to Behavior Trees

We now have the framework of rules that allows us to iteratively integrate a consistent set of individual requirement

trees to form a single tree. We show by means of examples how Rules (4) and (5) can be applied to achieve this.

Assume we want to integrate  $\mathcal{R}_1$  with  $\mathcal{R}_2$ . That is, we choose  $\mathcal{T} = s = \mathcal{R}_1$  and  $p_2 \rightsquigarrow t_2 = \mathcal{R}_2$  such that  $p_2$  is the node `Oven[Cooking]` and  $t_2$  is the sub-tree that follows this node in  $\mathcal{R}_2$ .  $\mathcal{R}_1$  establishes precondition  $p_2$ . Thus the condition for applying Rule (4) is satisfied (with  $t_1 = \emptyset$ ) and we can transform the tuple  $(\mathcal{R}_1, \mathcal{R}_2)$  into the sequence  $\mathcal{R}_1; t_2 = \mathcal{R}_{12}$ .

In a second step, we might want to integrate trees  $\mathcal{R}_{12}$  and  $\mathcal{R}_5$ . Again Rule (4) applies:  $\mathcal{R}_{12} = \mathcal{T}$ ,  $s = \mathcal{R}_1$ ,  $t_1 = \mathcal{R}_2$ , and  $\mathcal{R}_5 = p_2 \rightsquigarrow t_2$ , with  $p_2 = \text{Oven[Cooking]}$  and  $t_2$  is the sub-tree that follows, similarly to the example above.  $p_2$ , the precondition for  $\mathcal{R}_5$ , is established by sequence  $s$ . Thus, we can apply the rule and transform the tuple  $(\mathcal{T}, p_2 \rightsquigarrow t_2)$  into the tree  $\mathcal{T}[s \mapsto (t_1 \parallel t_2)] = \mathcal{R}_{125}$ .

$\mathcal{R}_{125}$  can be integrated with  $\mathcal{R}_7$  using Rule (4) with  $\mathcal{R}_{125} = \mathcal{T}$  and  $s = \mathcal{R}_1$ . This time  $t_1$  is a branching tree rather than a simple sequence. Furthermore,  $\mathcal{R}_7 = p_2 \rightsquigarrow t_2$ , with  $p_2$  as above, the oven is cooking, and  $t_2$  being the sub-tree that follows  $p_2$ . The rule applies similarly to the cases explained above.

Figure 12 shows the fully integrated Behavior Tree (chosen from the modelling space of possible integrations) as it was published by Geoff Dromey in [3]. Note that this tree includes an additional requirement, R8, that was identified as missing from the original requirements document during the model construction phase. Moreover, multiple points of iteration have been introduced using reversion nodes. Firstly, the reversion at the end of individual requirement tree  $\mathcal{R}_2$  back to the branching point `Oven[Cooking]`. Secondly, the reversion at the leaf node of  $\mathcal{R}_7$  which leads back to the node `Oven[Idle]`, and last the reversion at the leaf node of  $\mathcal{R}_5$  which reverts back to the root of the whole tree. Note that the iterations introduced first are contained in the sub-terms.

## VII. DISCUSSION

In this paper we have discussed a *constructive* approach to modelling a system, treating individual requirements as the building blocks. In the complete system model the individual requirements are preserved, facilitating traceability, and providing a clear mechanism for tracking and adapting to requirements change. The downside of the approach, in comparison with the usual *posit and prove* approach, is that the constructed model may be lacking in abstraction and design, since its structure will reflect the structure of the requirements document. However, using refactoring techniques, the constructed model may be transformed to have a better design, while tools and formal techniques may be employed to ensure that the new model is a faithful implementation of the constructed model, which is itself faithful to the original requirements.

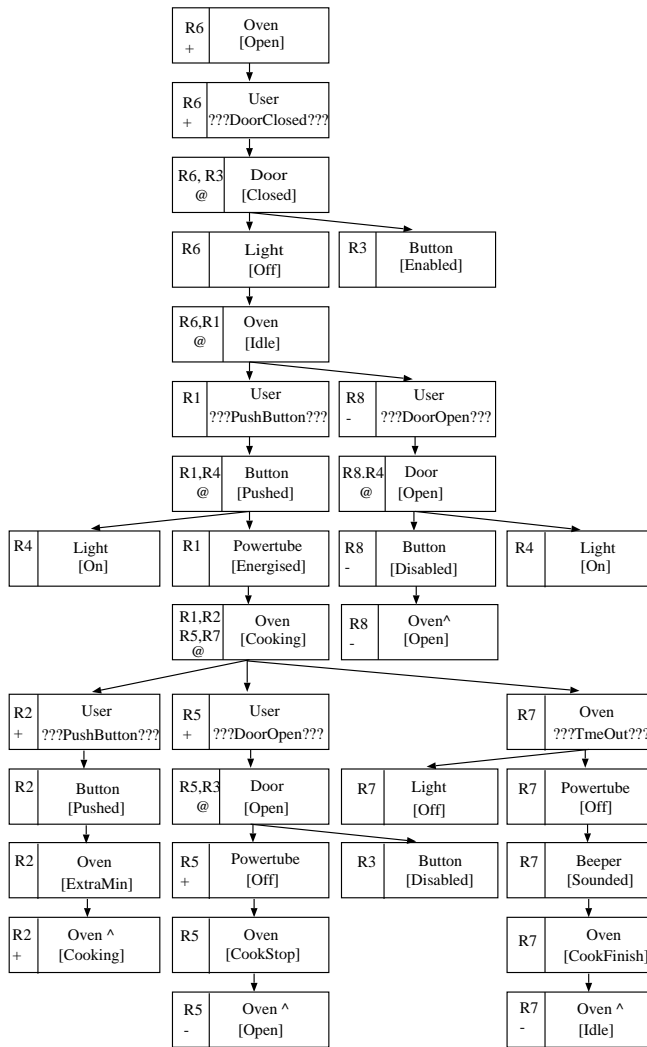


Figure 12. Fully integrated Behavior Tree model of the microwave oven

This constructive approach was based on the Behavior Engineering methodology [3], [4], [14], designed by Geoff Dromey to tackle problems encountered in large-scale system development associated with requirements capture. It has been applied with success in industry [15] to identify problems with large requirements documents. The methodology employs a relatively simple scheme to use syntactically matching nodes as the basis for integration, and thereby uncovering inconsistencies, ambiguities, and incompleteness. We have shown how this syntax-driven integration process relates to the more general task of constructing a model out of a set of smaller sub-models. While a pure syntax-driven approach cannot suffice in general to build a completely consistent model, it is a useful, fast and effective basis for guiding modellers to an integrated system model that faithfully, and traceably, preserves the original requirements, and is in a form readable by the client.

With our rules we intended to give a flavour of how a framework for integration could be defined. The rules given are general rules that work for the most common cases but they need to be customised to cater for the specialities of the notation they are applied to.

#### ACKNOWLEDGMENT

This work was undertaken with the financial support from the Australian Research Council Linkage Project grant LP0989363.

#### REFERENCES

- [1] J. A. McDermid, *Software Engineer's Reference Book*. Newton, MA, USA: Butterworth-Heinemann, 1991.
- [2] I. Sommerville, *Software Engineering*, 8th ed. Pearson Education, 2006.
- [3] R. G. Dromey, "From requirements to design: Formalizing the key steps," in *Proc. of Int. Conf. on Software Engineering and Formal Methods (SEFM 2003)*. IEEE Computer Society, 2003, pp. 2–13.
- [4] —, "Genetic design: Amplifying our ability to deal with requirements complexity," in *Models, Transformations and Tools*, ser. LNCS, vol. 3466. Springer-Verlag, 2005, pp. 95–108.
- [5] P. Zave and M. Jackson, "Conjunction as composition," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 2, no. 4, p. 411, 1993.
- [6] E. Brottier, B. Baudry, Y. L. Traon, D. Touzet, and B. Nicolas, "Producing a global requirement model from multiple requirement specifications," in *Proc. of IEEE International Conference on Enterprise Distributed Object Computing (EDOC 2007)*. IEEE Computer Society, 2007, pp. 390–404.
- [7] A. Goknil, I. Kurtev, and K. van den Berg, "A metamodelling approach for reasoning about requirements," in *Proc. of European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2008)*, ser. LNCS, I. Schieferdecker and A. Hartman, Eds., vol. 5095. Springer, 2008, pp. 310–325.
- [8] A. Goknil, I. Kurtev, K. van den Berg, and J.-W. Veldhuis, "Semantics of trace relations in requirements models for consistency checking and inferencing," *Software and Systems Modeling*, 2009. [Online]. Available: <http://dx.doi.org/10.1016/j.ic.2007.12.004>
- [9] G. Perrouin, E. Brottier, B. Baudry, and Y. L. Traon, "Composing models for detecting inconsistencies: A requirements engineering perspective," in *Proc. of International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2009)*, ser. LNCS, M. Glinz and P. Heymans, Eds., vol. 5512. Springer, 2009, pp. 89–103.
- [10] L. Wildman, "Requirements reformulation using formal specification: a case study," in *Workshop on the use of Formal Methods in Defence Systems*, C. Lakos, R. Esser, L. Branstetter, and J. Billington, Eds. Australian Computer Society, 2002, pp. 75–83.

- [11] “Behavior engineering,” <http://www.behaviorengineering.org>.
- [12] R. Colvin and I. J. Hayes, “A semantics for Behavior Trees,” The University of Queensland, Tech. Rep. SSE-2010-03, May 2010. [Online]. Available: <http://espace.library.uq.edu.au/view/UQ:204809>
- [13] —, “CSP with hierarchical state,” in *Proc. of Int. Conf. on Integrated Formal Methods (IFM 2009)*, ser. LNCS, M. Leuschel and H. Wehrheim, Eds., vol. 5423. Springer, 2009, pp. 118–135.
- [14] P. Lindsay, “Behavior Trees: from systems engineering to software engineering,” in *Proc. of Int. Conference on Software Engineering and Formal Methods (SEFM 2010)*, 2010, this volume.
- [15] D. Powell, “Behavior Engineering: A scalable modeling and analysis method,” in *Proc. of Int. Conference on Software Engineering and Formal Methods (SEFM 2010)*, 2010, this volume.