

# Comodeling Revisited: Execution of Behavior Trees in Modelica

Toby Myers<sup>1</sup> Wladimir Schamai<sup>2</sup> Peter Fritzson<sup>3</sup>

<sup>1</sup>Institute of Intelligent and Integrated Systems, Griffith University, Australia, [toby.myers@griffithuni.edu.au](mailto:toby.myers@griffithuni.edu.au)

<sup>2</sup>EADS Innovation Works, Germany, [wladimir.schamai@eads.net](mailto:wladimir.schamai@eads.net)

<sup>3</sup>PELAB- Programming Environment Lab, IDA, Linköping University, Sweden, [peter.fritzson@liu.se](mailto:peter.fritzson@liu.se)

## Abstract

Large-scale systems increasingly consist of a mixture of co-dependent software and hardware. The differing nature of software and hardware means that they are often modeled separately and with different approaches. Comodeling is a design strategy that allows hardware/software integration issues to be identified, investigated and resolved in the early stages of development. Previous work described a comodeling approach that integrates Behavior Engineering with Modelica. This paper revisits this approach and introduces a new means of integration that natively executes Behavior Trees in Modelica rather than utilizing external functions. This enhanced integration has several benefits. Firstly, it makes comodeling easier to apply as the comodel is captured solely in Modelica. Secondly, it makes the ability to execute Behavior Trees widely available. Finally, it opens the possibility to use comodeling with other complementary approaches such as the virtual verification of system designs against system requirements.

**Keywords:** *Comodeling, Behavior Engineering, Behavior Trees, Modelica, Model Driven Engineering*

## 1. Introduction

The increasingly co-dependent nature of software and hardware in large-scale systems causes a software/hardware integration problem. During the early stages of development, the requirements used to develop a software specification often lack the quantified or temporal information that is necessary when focusing on software/hardware integration. Also early on in development, the hardware details must be specified, such as the requirements for the sensors, actuators and architecture on which to deploy the software. There is a risk of incompatibility if the software and hardware specifications contain contradicting assumptions about how integration will occur. Even if the software and

hardware specifications are compatible, it is possible that a software/hardware combination with an alternative form of integration exists that would be more advantageous.

In previous work we introduced a design strategy called comodeling [3],[4] that lets developers systematically investigate and compare different software and hardware partitions to meet a system's constraints earlier in the design process, when integration problems are easier and cheaper to resolve. Our comodeling approach used multiview modeling to separately model the system's software and hardware aspects by integrating Behavior Engineering (BE) with Modelica. BE is a system and software engineering methodology that supports the engineering of large-scale dependable software intensive systems from out of its requirements. Modelica is an equation-based, object-oriented mathematical modeling language suited to modeling complex physical systems in multiple domains. BE is initially used to formalize and integrate the natural language requirements to create an executable specification. This specification is then combined with a Modelica model of the hardware and environment to create an integrated comodel.

One limitation of our comodeling approach was that the BE and Modelica models that together formed a comodel were executed separately and integrated using external functions. In this paper, we propose a new means of integration that enables a Behavior Tree (BT) to be natively executed in Modelica. This enhanced integration has several benefits. Firstly, it makes comodeling easier to apply as the comodel is captured solely in Modelica. The BT can now directly affect the acausal equations used to model the hardware and environmental components. Secondly, it makes the ability to execute Behavior Trees widely available. Finally, it opens the possibility to use comodeling with other complementary approaches such as the virtual verification of system designs against system requirements.

The remainder of this paper is structured as follows. In Section 2 we first give some background on Modelica and BE as well as briefly revisiting the automated train protection system case study we previously used to demonstrate comodeling. In Section 3 we discuss our new approach that integrates BE and Modelica by representing BTs in Modelica. Section 4 discusses some of the design decisions taken by our approach, in

particular why we chose to use code generation rather than develop a Modelica library. Section 5 discusses how our new integration approach allows comodeling to be complemented by other approaches such as vVDR. In section 6 we present related work before concluding in section 7.

## 2. Background

### 2.1 Introduction to Modelica

Modelica [5][6][7] is an open standard for system architecture and mathematical modeling. It is envisioned as the major next generation language for modeling and simulation of applications composed of complex physical systems.

The equation-based, object-oriented, and component-based properties allow easy reuse and configuration of model components, without manual reprogramming in contrast to today's widespread technology, which is mostly block/flow-oriented modeling or hand-programming.

The language allows defining models in a declarative manner, modularly and hierarchically and combining of various formalisms expressible in the more general Modelica formalism.

The multi-domain capability of Modelica allows combining of systems containing mechanical, electrical, electronic, hydraulic, thermal, control, electric power or process-oriented components within the same application model. In brief, Modelica has improvements in several important areas:

*Object-oriented mathematical modeling.* This technique makes it possible to create model components, which are employed to support hierarchical structuring, reuse, and evolution of large and complex models covering multiple technology domains.

*Physical modeling of multiple application domains.* Model components can correspond to physical objects in the real world, in contrast to established techniques that require conversion to "signal" blocks with fixed input/output causality. That is, as opposed to block-oriented modeling, the structure of a Modelica model naturally corresponds to the structure of the physical system.

*Acausal modeling.* Modeling is based on equations instead of assignment statements as in traditional input/output block abstractions. Direct use of equations significantly increases re-usability of model components, since components adapt to the data flow context for which they are used.

Several tools implement the Modelica language, ranging from open-source products such as OpenModelica [8], to commercial products like Dymola [9] and MathModelica [10].

### 2.2 Introduction to Behavior Trees

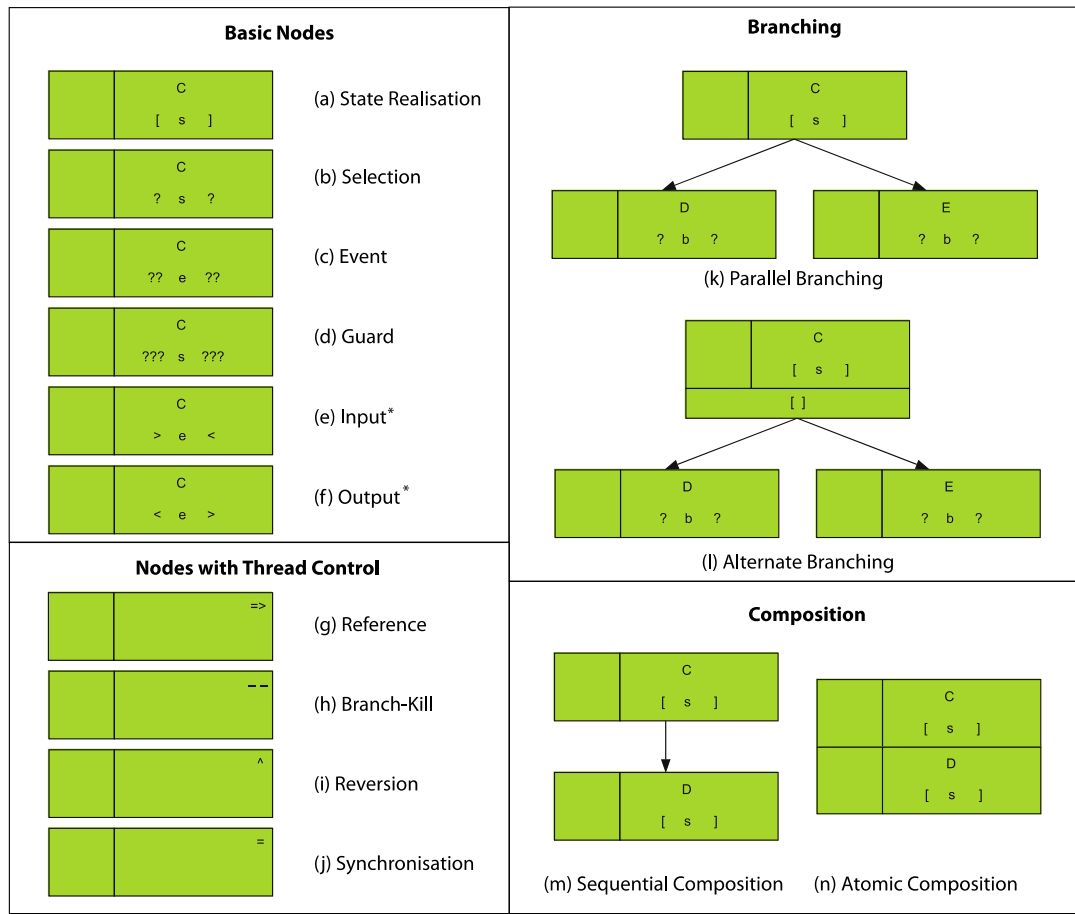
Prior to introducing BTs, it is necessary first to also have an understanding of the overall BE methodology. This is because BE is a tight interlinking of the Behavior Modeling Process (BMP) and the Behavior Modeling Language (BML). The BML consists of three integrated views: Behavior Trees, Structure Trees and Composition Trees [11],[12]. The Behavior Modeling Process (BMP) uses these views in four stages: Requirements Translation, Fitness for purpose test, Specification, and Design.

The combination of the BML and the BMP results in different uses of the views at each stage. When using BTs for example, the first stage of the BMP translates each requirement into a Requirement Behavior Tree (RBT). These are then integrated to form a single Integrated Behavior Tree (IBT), which is transformed into an executable specification in a Model Behavior Tree (MBT) and finally design decisions are applied to create a Design Behavior Tree (DBT).

Of the three views, BTs are the most relevant to the transformation to state machines as they capture dynamic behavior. A BT is a formal, tree-like graphical form that represents the behavior of individual or networks of entities which realize and change states, create and break relations, make decisions, respond to and cause events, and interact by exchanging information and passing control [11].

- (A) Tag; (B) Component Name; (C) Behavior;
- (D) Behavior Type; (E) Operator; (F) Label;
- (G) Traceability Link; (H) Traceability Status

**Figure 1.** Elements of a BT Node



(a) State Realisation: Component realises the described behavior; (b) Selection: Allow thread to continue if condition is true; (c) Event: Wait until event is received; (d) Guard: Wait until condition is true; (e) Input Event: Receive message\*; (f) Output Event: Generate message\*; (g) Reference: Behave as the destination tree; (h) Branch-Kill: Terminate all behavior associated with the destination tree; (i) Reversion: Behave as the destination tree. All sibling behavior is terminated; (j) Synchronisation: Wait for other participating nodes; (k) Parallel Branching: Pass control to both child nodes; (l) Alternate Branching: Pass control to only one of the child nodes. If multiple choices are possible make a non-deterministic choice; (m) Sequential Composition: The behavior of concurrent nodes may be interleaved between these two nodes; (n) Atomic Composition: No interleaving can occur between these two nodes. \*Note: single characters mean receive/send message internally from/to the system, double characters mean receive/send message from/to the environment.

**Figure 2.** Summary of the Core Elements of the Behavior Tree Notation

Figure 1 displays the contents of a BT node. The general form of a BT node consists of a main part (B-D) describing the name of the component and the behavior it exhibits qualified by a behavior type. The main part of the BT node may also have an optional operator (E) and a label (F). Each BT node also has a tag (A) which contains information to trace the node to the natural language requirements from which it was originally translated. The traceability link consists of an identifier (G) which is used to link the BT node to any associated requirements. The traceability status (H) indicates the status of this link using a set of values.

A summary of the core elements of the BT notation is shown in Figure 2. BTs are defined by a formal semantics defined in CSPsigma, an extension of CSP that can capture state based information [13].

### 2.3 Example: Automated Train Protection (ATP)

A key benefit of the BT notation is to provide a formal path from natural language requirements to a model of the software. To briefly demonstrate this, we will show the first two stages of the BMP using two requirements of an Automated Train Protection (ATP) system [3]. The ATP system automates the train's response to several track-side signals by sensing each signal and monitoring

the driver's reaction. If the driver fails to act appropriately, the ATP system takes control of the train and responds as required.

Table 1 shows two requirements of the ATP system. The translation of these two requirements into their corresponding RBTs is shown in Figure 4(a). Consider the RBT of requirement 6 (RBT6) with reference to the system requirements.

The first two nodes show the ATP controller receiving a value and a selection to determine if the value is a caution signal. The second node has a '+' in the tag to indicate this behavior is implied from the requirements as they do not explicitly state it is necessary to check the signal is a caution signal.

The next node shows that the Alarm is enabled, and captures that there is a relation between the Alarm and the Driver's Cab. Capturing the information about the Driver's Cab ensures that the original intent of the requirements is retained. The next BT node assumes that it is implied that the ATP Controller is responsible for observing whether the speed of the train is decreasing.

The final two BT nodes of RBT6 describe the relation between the ATP Controller and the Braking System, and the Braking System realising the activated state. Figure 4(b) also shows how integration of the two requirements proceeds using the integration point, ATP > value <.

The DBT shown in Figure 4(a) can then be used in conjunction with a Modelica model to create a comodel. This comodel can be used to determine appropriate quantified and temporal values to augment the existing requirements. For example, comodeling can be used for R6 to determine how often the speed of the train must be checked and how much of a change in speed is sufficient for the speed of the train to be considered decreasing.

R5	If a proceed signal is returned to the ATP controller then no action is taken with respect to the train's brakes.
R6	If a caution signal is returned to the ATP controller then the alarm is enabled within the driver's cab. Furthermore, once the alarm has been enabled, if the speed of the train is not observed to be decreasing then the ATP controller activates the train's braking system.

**Table 1.** Requirements R5 and R6 of the ATP system

### 3. Representing Behavior Trees in Modelica

The integration of BE and Modelica to perform comodeling was previously performed using external functions in Modelica. These external functions interacted with the Behavior Run-time Environment (BRE) where the Behavior Trees were executed. In this new approach, Modelica models are automatically generated from behavior trees captured in TextBE [14], a textual editor for capturing BE models. The generated Modelica model represents BTs using an algorithmic

section that is linked to a class for each component in the BT.

Each branch of the BT is represented by an integer variable. The nodes of each branch are represented as integer values the branch variable. This simple representation provides two benefits. Firstly, the one to one mapping of a node to the value of a branch variable makes it easy to follow the trace through the BT from the simulation output. Secondly, the branch variables simplify the representation of more complex BT constructs such as alternative branching, reversion, synchronization and branch-kill.

Each component in the BT has an integer variable, *state*, that records the current state of the component. This variable is updated using enumerated values that map an integer value to the string associated with each behavior of that component.

Several design decisions were made to simplify the execution of the BTs in Modelica. Firstly, only one node may be active at any one time. Secondly, a node is active for a user-specified delay. Finally, the execution of the BTs has been constrained to ensure fairness when multiple branches are executed in parallel.

Figure 3 shows the results of a simulation of a BT translated into Modelica that demonstrates state realisations, selections, sequential composition, alternative branching and reversion. The Modelica code and the translated BT are shown in Figure 5. The described transformation has also been included as part of the TextBE tool, together with a set of examples available at [www.behaviorengineering.org](http://www.behaviorengineering.org).

#### 3.1 Basic Nodes

**State Realisation** – A state realisation updates the *state* variable of the associated component to the enumerated value of the behavior of the BT node.

```
c.state := Integer(c.states.s);
```

**Selection** – A selection performs an equality check on the *state* variable of the associated component, comparing it to the enumerated value of the behavior of the BT node. Depending on the result of this equality check, the flow of control either continues or is terminated.

```
if c.state == c.state_s then
  ... // Continue flow of control
else
  ... // Terminate branch
end if;
```

**Guard** – A guard is similar to a selection, with the exception that the else branch is not included to ensure that the guard is continually re-evaluated until true.

```
if c.state == c.state_s then
  ... // Continue flow of control
end if;
```

**Input** – Inputs and outputs are implemented as boolean variables. When the variable is true, the input is active.

Events last for one cycle, to ensure that if an internal output is active in one branch, it can be received by an internal output in another branch. Inputs are represented with an equality check that is true if the associated event becomes active.

```
if e2 then
  ... // Continue flow of control
end if;
```

**Output** – Each output is associated with a real variable. When the boolean variable associated with the event is set to true, the real variable is set to the current time plus the value of the user-specified delay. An if statement then ensures that the output is set to false one cycle after they are activated.

```
e2 := true;
e2Delay := startTime + (delay/2);
...
if startTime > e2Delay then
  e2 := false;
end if;
```

### 3.2 Branching and Composition

**Sequential Composition** – Updates the value of the branch variable.

```
if branch1 == 1 then
  ... // Node behavior
  branch1 := 2;
elseif ...
```

**Parallel Branching** – Clears the current branch value and sets the branch value of the child branches to their first node.

```
if branch1 == 1 then
  ... // Node behavior
  branch1 := 0;
  branch2 := 1;
  branch3 := 1;
elseif ...
```

**Alternative Branching** – As per parallel branch, but when the first node of any of the child branches is activated all the sibling branches are terminated.

```
if branch2 == 1 then
  ... // Node behavior
  branch2 := 2;
  branch3 := 0;
end if;
```

**Atomic Composition** – Adds further constraint to all sibling branches of atomic composed nodes that flow of control cannot continue if the branch values of the atomic composed nodes are active.

```
if branch3 == 1 and not(branch2==1 or
branch2==2) then
  ... // Node behavior
  branch3 := 2;
```

### 3.3 Operators

**Reference** – Clears the current branch value and sets the branch value of the destination node.

```
if branch1 == 3 then
  branch3 := 0;
  branch2 := 2;
```

**Reversion** – Clears the current branch value and all sibling parallel branches and sets the branch value of the destination node.

```
if branch1 == 3 then
  branch2 := 0;
  branch3 := 0;
  branch1 := 1;
```

**Branch-Kill** – Clears the branch value of the destination node and the branch value of any of its descendants.

```
if branch2 == 2 then
  branch3 := 0;
  ... // Continue flow of control
```

**Synchronization** – One synchronisation node checks when the branch value of all nodes is set correctly and sets a boolean variable to true. All other synchronisation nodes wait until this Boolean variable is true.

```
if branch3 == 2 and branch2 == 3 then
  sync1 := true;
  ... // Node behavior
  ... // Continue flow of control

if branch2 == 3 and sync1 then
  ... // Continue flow of control
```

## 4. Libraries versus Code Generation

The approach taken to represent behavior trees in Modelica was heavily influenced by previous work involving the execution of UML state machines using Modelica [1]. This work raised two points of discussion: whether to use the declarative (equations) or imperative (algorithms) constructs of Modelica and whether to create a Modelica library or create a code generator.

The imperative portions of Modelica were found to be necessary to represent a comprehensive set of state machine concepts in Modelica such as inter-level transitions; entry, do, and exit actions of states, and; fork and synchronization of parallel regions. Algorithmic code was necessary for these concepts because a particular sequence of operations was required. A similar situation occurred with the translation of behavior trees to Modelica for concepts involving concurrent behavior such as alternative branching, atomic composition, reversion, reference, branch-kill and synchronization. We were unable to find a means to represent these concepts using equations but their representation in algorithmic sections was reasonably straightforward.

The second point of discussion involved whether the representation should be implemented using a Modelica library or if model transformation should be used to

generate the Modelica representation from a dedicated modeling editor. As with state machines, we chose to implement behavior trees in Modelica using the later approach.

This implementation choice is a specific instance of the two choices faced whenever any new domain-specific language (DSL) is created [19]: language invention and language exploitation. Language invention involves developing a new language from scratch. Language exploitation creates a DSL by extending an existing general purpose language (GPL), a language that uses generalized concepts that cut across multiple modeling dimensions.

Language invention is best applied when existing languages cannot be used to capture domain-specific concepts. Language invention creates DSLs that provide the most benefit to users but that also requires the development and maintenance of completely new tools such as an editor, compiler, debugger and simulator. Language invention is most suited to DSLs with a focus on syntax over semantics, which semantics often implicitly defined by model compilers or generators [20].

DSLs created by language exploitation can leverage any existing technology developed by the chosen GPL to simplify the implementation and mapping between multiple DSLs. It also allows the utilization of existing editors and the potential to benefit from a user's familiarity with the existing GPL. The ultimate success of language exploitation, however, depends on the effectiveness of the chosen GPL at capturing the concepts of the DSL.

If a model library approach were to be taken, editors would have to be used that were not specifically designed for visual languages such as state machines and behavior trees. It is likely that compromises would also have to be made to accommodate the representation of state machines or behavior trees in a Modelica library.

It is for these reasons that both the state machine and the behavior tree mappings to Modelica used a language invention approach for capturing and visualizing their models. This allows editors that were specifically designed for that DSL to be used. For model execution, however, a language exploitation approach is taken where the DSL is transformed into a Modelica representation. This approach allows the advantages of Modelica to be leveraged whilst also maintaining the advantages of having an editor tailored to the particular DSL. It also makes it possible to integrate BE with ModelicaML and vVDR as discussed in the following section.

## 5. Complementing Comodeling with vVDR

Comodeling, as has been outlined previously, provides a number of benefits. It leverages BE to follow a process that helps to ensure that system requirements are correct, consistent, complete, and unambiguous. It also determines the quantified and temporal information required for software/hardware interactions in the early stages of development. The effect different software and

hardware partitions have on the timing, performance and complexity of individual components and on the integrated system's behavior as a whole can then be determined by simulating different comodels.

The representation of BTs natively in Modelica, however, makes it easier to integrate comodeling, and BE in general, with other approaches such as vVDR.

### 5.1 Introduction to vVDR

vVDR is a method for a virtual Verification of system Design alternatives against system Requirements. The application of this method is illustrated in [2] using ModelicaML. ModelicaML is a UML- and Modelica-based language. It supports all textual Modelica constructs and, in addition, supports an adopted version of the UML state machines and activity diagrams for behavior modeling as well as UML class diagram and composition diagram for structure modeling. This enables engineers to use the simulation power (i.e., solving of hybrid DAEs) of Modelica by using standardized graphical notations for creating the system models.

In the vVDR method, each requirement, or more general any analysis question, is formalized as a model which evaluates the violation or fulfilment of a requirement, or more generally provides the answer to the stated analysis question.

In case of a requirement, the formalization is done by relating measurable properties, that are addressed in the requirement statement, to each other in order to express when this requirements is violated.

Then requirements models and a system design alternative model are instantiated and bound to each other in a test model. The test model also includes the test scenario, which provides stimuli for the system model. Finally, the models are translated into Modelica code, the test model is simulated and the results are analysed.

The vVDR method and its implementation in ModelicaML focus on enabling the analysis and verification of different design alternatives against the same set of requirements. It is designed to automate the composition of test model and to facilitate a flexible way to reuse formalized requirements models, test scenario models and system design alternative models in various combinations.

For example, the same test scenario model can be used to verify different system design alternatives against the same set of requirements. Similarly, the same requirements can be used for verifications that are driven by different test scenario models.

The vVDR approach does not integrate individual requirements and, hence, does not provide means to determine inconsistencies between requirements. Inconsistencies between requirements can only be detected when requirement violations are detected during simulations.

In contrast, the BE methodology provides means to ensure the consistency of the specification. Any inconsistency or incompleteness detected in the



specification is resolved and corrected so that each of the individual requirements does not conflict with any other requirement in the specification.

This is the starting point for the vVDR that assumes a set of requirements that do not conflict with each other. The advantage of using the vVDR approach is that it enables the verification of a system model against a subset of requirements that are of interest for a particular analysis. This is useful for the assessment of design alternatives by focusing on specific aspects as well as for regression testing when some requirements have changed or design model has evolved.

## 5.2 Integrating vVDR with BE and Comodeling

There are two applications that could be investigated for integrating vVDR with BE and Comodeling.

The first application is to integrate BE with vVDR by using a Model Behavior Tree as a source for the generation of both vVDR requirements violation monitors and test cases. Test cases would be used to drive a design alternative in order to determine if it fulfils the requirements by evaluating the generated requirements violation monitors.

The second application is to augment the existing comodeling approach with vVDR. vVDR would not need to provide violation monitors, as the comodels have been built from out of the requirements. Instead vVDR could provide monitors that evaluate the performance of different comodels to find the best candidate that fulfils a set of criteria.

## 6. Related Work

We are not aware of any other work that represents BTs in Modelica. Schamai et al have defined an approach to transform state machines into Modelica using code generation [1]. Modelica libraries also exist for state machines, e.g., Stategraph [15] and Petri nets [16]. Myers et al [17] have also created a transformation for converting BTs into executable UML state machines.

Wendland [18] has outlined an approach to augment the BMP to capture testing information and generate test cases from BTs. This could be integrated with vVDR and comodeling.

## 7. Conclusion

This paper presents an approach for executing BTs using Modelica in order to support comodeling and simulation of system requirements, hardware and software. Modelica is used as action language and as execution technology. By using Modelica as action language in BTs the mathematical modeling and simulation power of Modelica is leveraged. For example, Modelica equations or algorithm code is used for capturing of encapsulated behavior (e.g. inside state realization) in BTs. This enables the modeling of continuous-time, discrete-time or event-based behavior including the solving of differential algebraic equations and events handling. This way logical requirements or

software models can be simulated together with physical system behavior models using the same technology.

The model-transformation rules, described in section 3, are implemented as in the code generator that can be used in TextBE[14] as a prototype. The decision for implementing a code generator instead of developing a Modelica library for BTs is discussed in section 4. The Modelica code, that is generated from BTs models, can then be loaded and simulated using OpenModelica tools [8]. This combination is the first publically available runtime execution environment for Behavior Trees models. Moreover, using Modelica as a common modeling and simulation technology enables complementing BE methodology with other approaches, such as the vVDR method implementation in ModelicaML[2], as discussed in section 5.

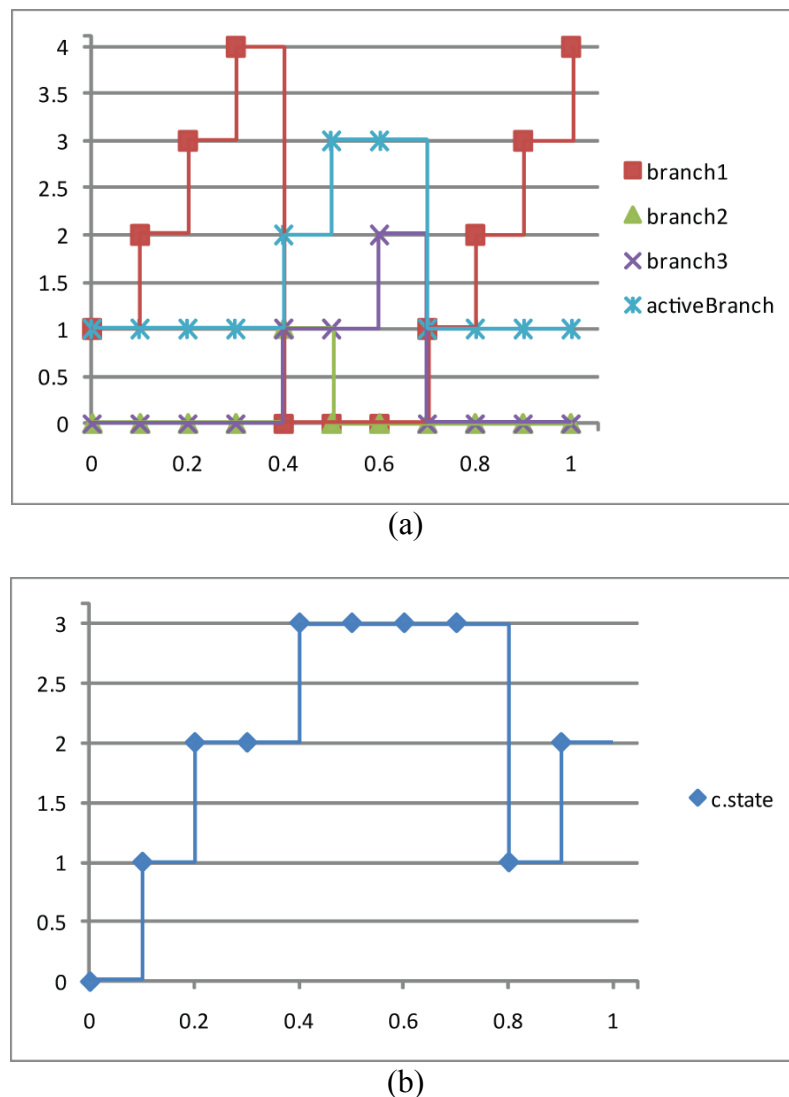
## Acknowledgements

This research was supported under Australian Research Council Linkage Projects funding scheme (project number LP0989363) and by VINNOVA the Swedish Governmental Agency for Innovation Systems in the OPENPROD project.

## References

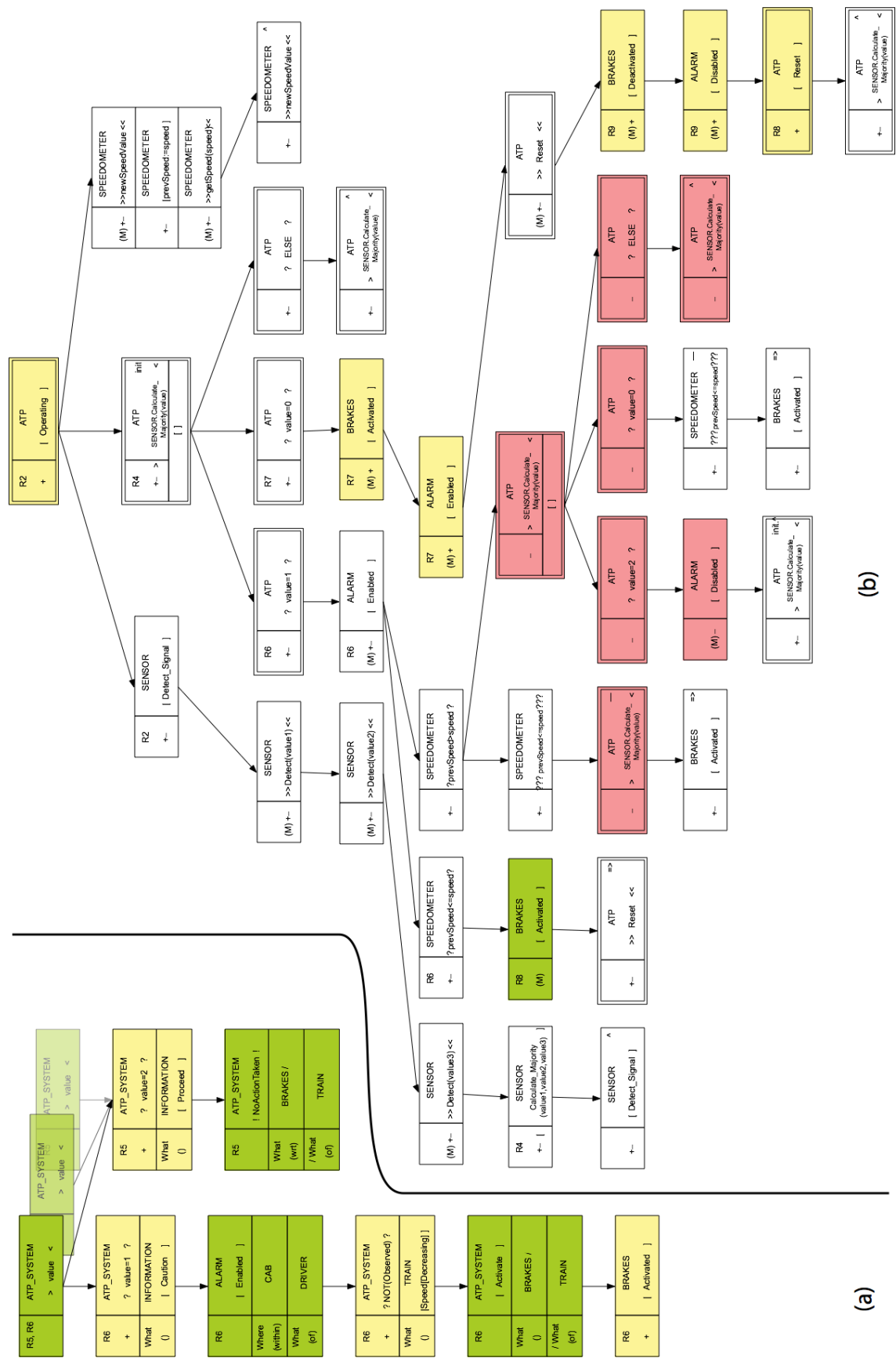
- [1] Wladimir Schamai, Uwe Pohlmann, Peter Fritzson, Christian J.J. Paredis, Philipp Helle, Carsten Strobel. Execution of UML State Machines Using Modelica In *Proceedings of the 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, (EOOLT 2010), Published by Linkoping University Electronic Press, [www.ep.liu.se](http://www.ep.liu.se), In conjunction with MODELS 2010, Oslo, Norway, Oct 3, 2010.
- [2] Wladimir Schamai, Philipp Helle, Peter Fritzson, Christian Paredis. Virtual Verification of System Designs against System Requirements, In *Proc. of 3rd International Workshop on Model Based Architecting and Construction of Embedded Systems* (ACES 2010). In conjunction with MODELS 2010. Oslo, Norway, Oct 4, 2010.
- [3] Toby Myers, Geoff Dromey, Peter Fritzson. Comodeling: From Requirements to an Integrated Software/Hardware Model, *IEEE Computer* 44(4) pp.62-70, April 2011 doi: 10.1109/MC.2010.270.
- [4] Toby Myers. *The Foundations for a Scaleable Methodology for Systems Design*, PhD Thesis, School of Computer and Information Technology, Griffith University, Australia, 2010.
- [5] Modelica Association. Modelica: A Unified Object-Oriented Language for Physical Systems Modeling: Language Specification Version 3.0, Sept 2007. <http://www.modelica.org>.
- [6] Michael Tiller. Introduction to Physical Modeling with Modelica. Kluwer Academic Publishers, 2001.
- [7] Peter Fritzson. Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. Wiley-IEEE Press, 2004.
- [8] Open Source Modelica Consortium. OpenModelica. <http://www.openmodelica.org>.
- [9] Dynasim. Dymola. <http://dynasim.com>.
- [10] MathCore. Mathmodelica. <http://www.mathcore.com>.

- [11] Geoff Dromey. From requirements to design: Formalizing the key steps. in Proc. Conf. on Software Engineering and Formal Methods (SEFM). IEEE Computer Society, pp. 2-13, 2003.
- [12] Geoff Dromey. Climbing over the “no silver bullet” brick wall. IEEE Software, vol. 23, pp. 120, 118-119, 2006.
- [13] Robert Colvin and Ian Hayes. A semantics for Behavior Trees using CSP with specification commands. *Science of Computer Programming*, In Press, Corrected Proof, Available online 9 December 2010.
- [14] Toby Myers. TextBE: A Textual Editor for Behavior Engineering. *Proceedings of the 3rd Improving Systems and Software Engineering Conference (ISSEC)*, Sydney, Australia, 2-5 August 2011 (Accepted).
- [15] Martin Otter, Martin Malmheden, Hilding Elmquist, Sven Erik Mattsson, and Charlotta Johnsson. A New Formalism for Modeling of Reactive and Hybrid Systems. In *Proceedings of the 7th International Modelica Conference*, Como, Italy, September 20-22, 2009.
- [16] Sabrina Proß and Bernhard Bachmann. A Petri Net Library for Modeling Hybrid Systems in OpenModelica.
- Proceedings of the 7th International Modelica Conference*, Como, Italy, 20-22 September 2009.
- [17] Toby Myers, M. Wendland, S. Kim, Peter Lindsay. From Requirements to Executable UML State Machines: A Formal Path using Behavior Engineering and M2M Transformations, *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, Wellington, New Zealand, 16-21 October 2011 (submitted).
- [18] M. Wendland, I. Schieferdecker, A. Vouffo-Feudjio. Requirements driven testing with behavior trees. In *Proceedings of the ICST Workshop Requirements and Validation, Verification & Testing (ReVVerT 2011)*. 2011. Accepted.
- [19] M. Mernik, J. Heering and A. M. Sloane. When and how to develop domain-specific languages, *ACM Computing Surveys (CSUR)*, vol. 37(4), 2005. pp. 316-344.
- [20] J. Heering and M. Mernik, Domain-specific languages in perspective, Tech. rep., CWI, sEN-E0702. 2007.



**Figure 3.** Simulation Plots of Selection Example Translation (a) branch and activeBranch variables (b) c.state variable





**Figure 4.** Behavior Engineering model of the Automated Train Protection System (a) Integration of Requirements R5 and R6 (b) Design Behavior Tree

```

model Selection
  Integer branch1(start=1),branch2,branch3;
  Integer activeBranch(start=1),
  lastActiveBranch;
  Boolean next(start=true);
  C c;
  Real startTime;
  constant Real delay = 0.1;

  algorithm
  when time>startTime + delay then
    while branch1==pre(branch1) and
    branch2==pre(branch2) and
    branch3==pre(branch3) and next loop
      if activeBranch==1 then
        if branch1==1 then
          c.state := Integer(c.states.s);
          branch1:=2;
        elseif branch1==2 then
          c.state := Integer(c.states.t);
          branch1 := 3;
        elseif branch1==3 then
          if c.state==c.state_t then
            branch1 := 4;
          else
            branch1 := 0;
          end if;
        elseif branch1==4 then
          c.state := Integer(c.states.u);
          branch1 := 0;
          branch2 := 1;
          branch3 := 1;
        end if;
      elseif activeBranch==2 then
        if branch2==1 then
          if c.state==c.state_s then
            branch2 := 2;
            branch3 := 0;
          else
            branch2 := 0;
          end if;
        elseif branch2==2 then
          c.state := Integer(c.states.v);
          branch2 := 0;
        end if;
      elseif activeBranch==3 then
        if branch3==1 then
          if branch2==0 then
            branch3 := 2;
          end if;
        elseif branch3==2 then
          branch3 := 0;
          branch1 := 1;
        end if;
      end if;

      activeBranch := activeBranch + 1;
      if activeBranch==4 then
        activeBranch := 1;
      end if;
      if activeBranch==1 and branch1==0 then
        activeBranch := activeBranch + 1;
      end if;

```

```

      if activeBranch==2 and branch2==0 then
        activeBranch := activeBranch + 1;
      end if;
      if activeBranch==3 and branch3==0 then
        activeBranch := 1;
      end if;
      if activeBranch==1 and branch1==0 then
        activeBranch := activeBranch + 1;
      end if;
      if activeBranch==2 and branch2==0 then
        activeBranch := activeBranch + 1;
      end if;
      if activeBranch==lastActiveBranch then
        next := false;
      end if;
    end while;
    startTime := time;
    lastActiveBranch := activeBranch;
    next := true;
  end when;
end Selection;

class C
  type states = enumeration(s,t,u,v);
  constant Integer state_s =
  Integer(states.s);
  constant Integer state_t =
  Integer(states.t);
  Integer state;
end C;

```

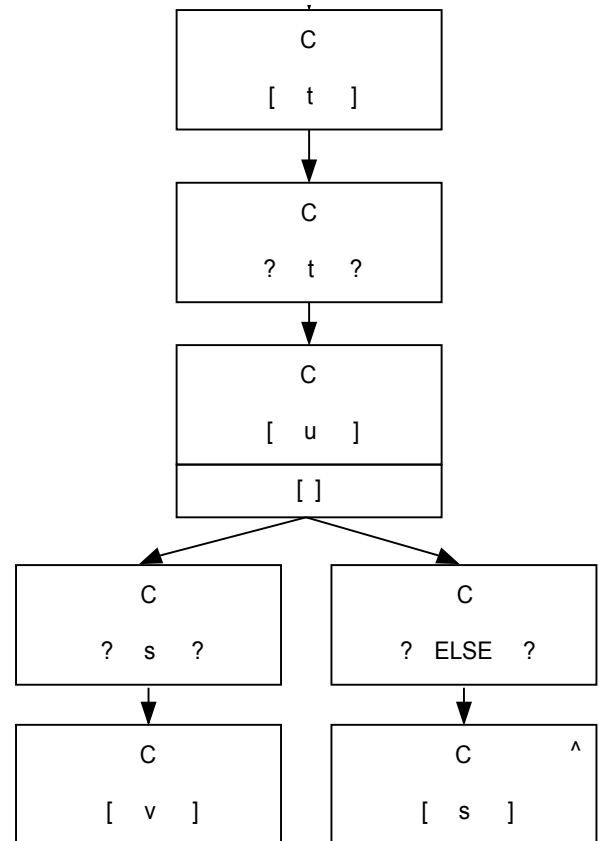


Figure 5. Example Translation of a Behavior Tree into Modelica