Seamlessly Integrating Software & Hardware Modelling for Large-Scale Systems

Toby Myers[1] Peter Fritzson[2] R. Geoff Dromey[1]

[1]School of Information and Computing Technology, Griffith University, Australia,
toby.myers@student.griffith.edu.au, g.dromey@griffith.edu.au

[2]Department of Computer and Information Science, Linköping University, Sweden, petfr@ida.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 modelled separately and with different approaches. This can cause failures later in development during the integration of software and hardware designs, due to incompatible assumptions of software/hardware interactions. This paper proposes a method of integrating the software engineering approach, Behavior Engineering, with the mathematical modelling approach, Modelica, to address the software/hardware integration problem. The environment and hardware components are modelled in Modelica and integrated with an executable software model designed using Behavior Engineering. This allows the complete system to be simulated and interactions between software and hardware to be investigated early in development.

Keywords software-hardware codesign, large-scale systems, Behavior Engineering, Modelica.

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.

One approach of evaluating software/hardware integration is to build prototypes of the software and hardware. This approach allows software/hardware interactions to be investigated, but also diverts attention away from the individual modelling of the respective software and hardware models. Investigating integration using software/hardware prototypes also has the disadvantage of occurring later in development, requiring decisions to have already been made as to how integration will occur.

Addressing the issues involved with integrating software and hardware models of systems earlier in development can reduce the risk of incompatibilities between the software and hardware specifications. Earlier investigation of software/hardware interactions minimises changes that must be made later in development when they are harder and more expensive to fix. If the method of investigating integration uses simulation of specifications, it allows many different integration configurations to be evaluated to assist in finding the best solution. The simulation of software/hardware co-specifications uses abstract models of the software and hardware to focus on timing of the interactions between the hardware and software. Co-specification simulation is used by many system design tools such as STATEMATE and MATLAB [6].

The principle of separation of concerns advocates that due to the differing nature of software and hardware, different modelling techniques should be used. Software modelling consists of capturing the required functionality, and how the functionality can best be organised to facilitate future reuse, extensibility, etc. Hardware modelling focuses on interactions with the physical environment through sensors and actuators which is best described mathematically. Currently, UML is the dominant graphical modelling notation for software, whereas Modelica is the major equation-based object-oriented (EOO) mathematical modelling language for modelling complex physical systems.

Previous work in this area resulted in the ModelicaML UML profile [15, 14] partly based on the SysML profile [13]. ModelicaML combined the major UML diagrams with Modelica graphic connection diagrams. However, there are problems with this approach. The imprecise

2nd International Workshop on Equation-Based Object-Oriented Languages and Tools. July 8, 2008, Paphos, Cyprus.

EOOLT 2008 website:
http://www.eoolt.org/2008/

semantics and portability problems of UML create difficulties for executable specifications. Moreover, there is no well-defined process of precisely capturing and converting informal software requirements into more formal representations that can be analysed and further transformed into executable models.

Fortunately, the Behavior Engineering (BE) approach (see Section 3) addresses several of these problems. BE is a systems & software engineering approach of modelling software-intensive systems that has precise requirements capture. The behavioral view of BE has a formal semantic described in process algebra. BE also supports model-checking, simulation, and the code-generation of executable models.

Thus, we propose an integrated approach, where BE is used to model and capture requirements of the software aspects of a product, whereas Modelica is used for high-level modelling of the system's environment and hardware components. We consider the integration method to be seamless, as the software and hardware models are combined in an inconspicous way which allows both formalisms to focus independently on their respective domains. We also propose this method is suited to be applied to large-scale systems, as both BE and Modelica have been used independently to model large-scale systems [16, 7]. This distinguishes this approach from co-design approaches such as COSMOS[9] and Polis[2] which are focused towards the more fine-grained software/hardware interactions of embedded systems.

Adoption of an integrated approach to product/system design should allow for a much more effective product development process since a system can be analysed and tested in all stages of development. The integration of BE and Modelica models supports this through allowing different hardware/software configurations to be investigated, such as:

- The periodic/aperiodic sampling of sensors and the action of actuators on the physical environment can be simulated to determine the effect on the software of the system. This may also involve simulating the failure of a sensor/actuator or errors in communication.
- The capabilities of the various combinations of hardware and software platforms on which the software could be deployed can be simulated by choosing periodic/aperiodic frequencies at which to allow interactions between the Modelica and BE models.
- The hardware and software can be tested in different simulated environments and scenarios.

In this paper we combine these two formalisms for the first time, in a study of the integrated software/hardware modelling of an Automated Train Protection (ATP) system. BE is used to model the control software of the ATP system, and Modelica is used to model physical components like the train, the driver, actuators, sensors, etc. The modelled ATP system is used to illustrate the benefits of investigating the integration of software/hardware specifications early in development.

In Section 2 & 3 we first give some background on Modelica and BE, before presenting the details of our integration method in Section 4. The integration method is then applied to a case study of the system modelling and simulation of an ATP system in Section 5.

2. Modelica Background

Modelica [12, 17, 8] is an open standard for system architecture and mathematical modelling. It is envisioned as the major next generation language for modelling 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 modelling 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.

A component may internally consist of other connected components, i.e., as in Figure 1 showing hierarchical modelling.

The multidomain 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 modelling.* 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 modelling 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 modelling, the structure of a Modelica model naturally corresponds to the structure of the physical system.

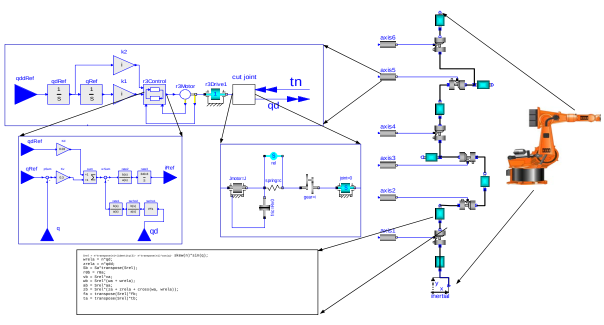


Figure 1. Hierarchical Modelica model of an industrial robot

- *Acausal modelling.* Modelling 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 support the Modelica specification, ranging from open-source products such as OpenModelica [12], to commercial products like Dymola [5] and MathModelica [11].

3. Behavior Engineering Background

BE [3] is an integrated approach that supports the engineering of large-scale dependable software intensive systems at both the systems engineering and software engineering level. BE has been proven as a useful technique in requirements analysis of large-scale industry projects, detecting defects at a rate approximately two to three times higher than conventional techniques [16]. The BE approach uses the Behavior Modelling Language (BML) and the Behavior Modelling Process (BMP) to transform a system described in natural language requirements to a design composed of a set of integrated components.

3.1 The Behavior Modelling Language

The BML is a graphical, formal language consisting of three tree-based views: Behavior Trees, Composition Trees and Structure Trees[1].

A Behavior Tree (BT) is a "formal, tree-like graphical form that represents behavior of individual or networks of entities which realize or change states, make decisions, respond-to/cause events, and interact by exchanging information and/or passing control." [4]. The formal semantics of BTs are described in the Behavior Tree Process Algebra (BTPA) language [1]. BTPA supports simulation, formal verification by model-checking and is a foundation for BT execution. BTs can describe multiple threads of behavior. Coordination is achieved using either message-passing (events), shared variable blocking or synchronisation. A summary of the BT notation is shown in Figure 2.

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

3.2 The Behavior Modelling Process

The BMP is closely tied with the BML. The BMP consists of a number of distinct stages: Translation, Integration, Refinement and Design. Each of these stages utilises the BML to address the problems of scale, complexity and imperfect knowledge that arise when dealing with systems described by a large number of natural language requirements.

Translation proceeds one requirement at a time, resulting in a Requirement Behavior Tree (RBT) that is created from the original natural language description. As each RBT is translated, the Requirement Composition Tree (RCT) should be updated to include any new information such as additional components, states, etc. Also, in order to ensure the translation process is as rigorous as possible, it is important not to add or remove information but to capture the intention that is expressed in the natural language description.

Being able to deal with one requirement at a time, localises the information that the modeller must absorb and helps to control the complexity of modelling the system. It also makes it possible for a team of translators to work on modelling the system in parallel, using the RCT to coordinate their work.

Two example RBTs are shown in Figure 4. Discussion of the translation of an example RBT from the original requirements is discussed in section 5.1.

Once all the requirements have been translated they are integrated to form an Integrated Behavior Tree (IBT) which can then be used to gain a holistic understanding of the problem space. The process of integration itself also helps to discover imprecise, conflicting and missing requirements in the description of the system. This is because forming the IBT is a fitness test for the requirements, if requirements cannot integrate it indicates there are problems with the description of the system.

When the IBT has been completed, the integrated view of the system's behavior helps to detect further defects in the original natural language requirements. Resolution of these defects produces a specification of the system, known as a Model Behavior Tree (MBT).

As the specification is still in the problem space, design decisions must be made to move to the solution space. The result is a Design Behavior Tree (DBT). Important design decisions include determining the boundaries between the system and the environment and the system and the components. The system-environment boundary determines how the system described by the DBT interacts with the environment, essentially determining the interface of the system. The system-component boundary involves a tradeoff between shifting complexity to either the DBT or to the components.

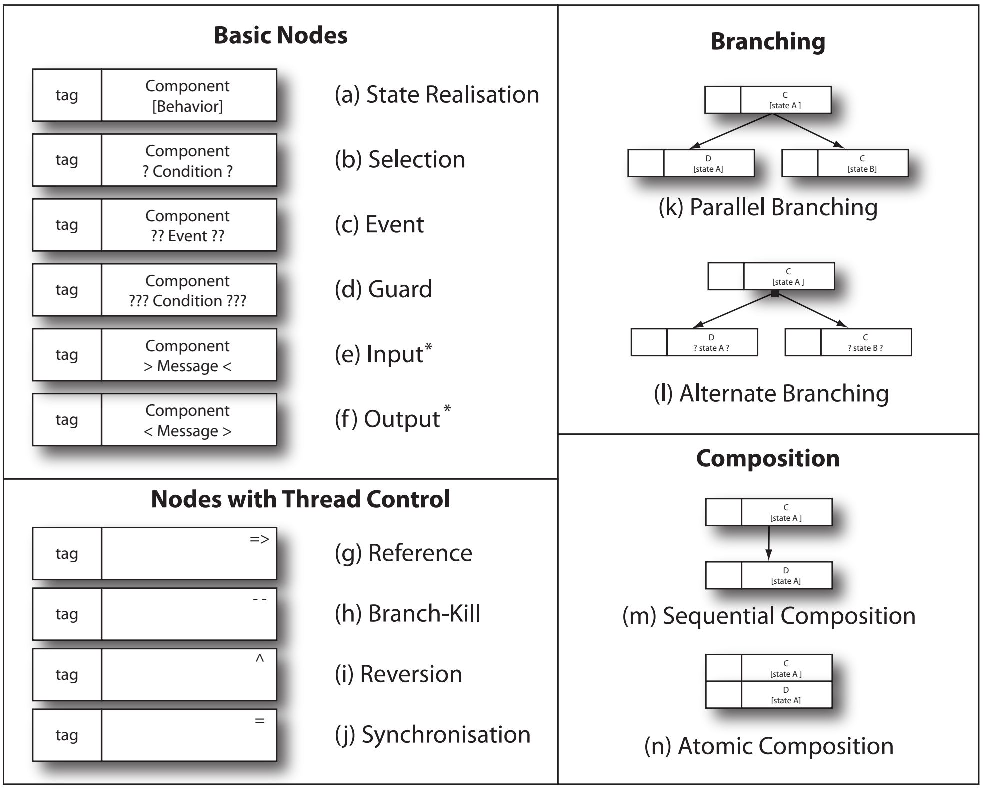
An example DBT is shown in Figure 5. The design decisions used to make this DBT are described in Section 5.1.

3.3 Executing a BE Model

BTs contain a description of the functionality of the system which makes them the primary interest when discussing executable models.

One approach to execute a BE model is to consider a BT as a set of interconnected interleaved state machines. Each component can be implemented by decomposing its individual state machine and implementing it. The BT is

[1] Due to space restrictions Structure Trees will not be discussed



(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

also implemented as a state machine which coordinates the component state machines.

Another approach to execute a BE model is to consider BTs as a model to describe multiple-threaded behavior, making each BT node a process. This allows traditional process control schedulers consisting of New, Ready, Blocked, Running, and Exit states to be applied to BTs.

For example, a state realisation node would take the following path through the scheduler: New, Ready, Running, Exit. Moving from the Ready to Running State is determined by a scheduling algorithm, ranging from simple examples such as First In, First Out (FIFO) to more complex priority-based schedulers. When a state realisation node is in the running state any encapsulated computation associated with the component's state is executed. Upon reaching Exit its child nodes are added to the scheduler in the New state to continue execution.

Alternatively, a guard node would take the following path through the scheduler: New, Blocked, Ready, Running, Exit. The guard node stays in the Blocked state until a change in another thread of behavior causes its condition to become true, upon which it changes to the Ready state and progresses similarly to the state realisation node. The scheduler also consists of more complex rules for BT execution such as alternative branching and atomic composition.

The benefit of the process control approach is that code generation from a BT is easily automatable. All that is required in addition to the automatable code generation is a version of the scheduler for the platform on which the executable BE model is deployed.

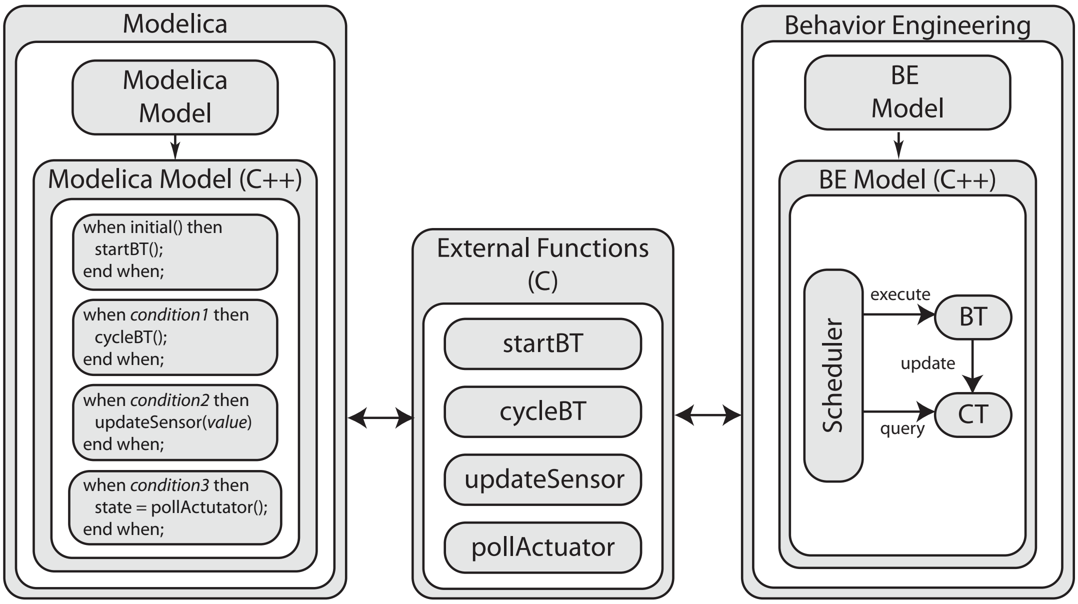


Figure 3. Interactions between Modelica and BE Models

4. Integrating Modelica & BE Models

Integration of Modelica and BE models occurs after the models are compiled/code generated into C++ source files. Integration between the Modelica model and BE model is performed using Modelica external functions mapped to C source code. The 'C' external functions are then linked to the 'C++' implementation of the BE model. This method of integration makes the Modelica model responsible for managing all interactions with the BE model.

Figure 3 shows the integration of a Modelica model and a BE model. There are three possible types of interaction: starting/cycling the BT scheduler; adding an event to the scheduler containing sensor information; or, polling the scheduler for an actuator command. The *initial()* function is used to start the execution of the BT. Boolean conditions are then used to determine when to cycle the BT scheduler, pass on sensor information or receive actuator commands.

If interactions are periodic, a boolean clock setup with a sample function can be used to set the frequency with which the interaction will occur. If the interaction should occur based upon a physical event simulated in Modelica, the event can change the boolean condition which will initiate the interaction with the BE model. More complex aperiodic, randomised, or interactions with losses in communication or failures of components can also be simulated using Modelica constructs. Failures of sensors, actuators or the communication between them and the software can be simulated by mearly not performing the interaction that would normally occur.

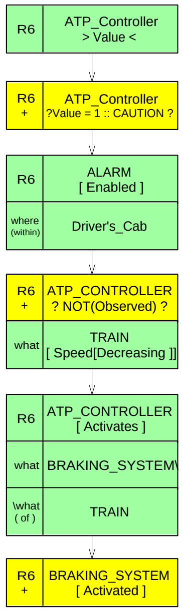
This method of interaction ensures that the details of the interactions that are simulated are documented as part of Modelica model. It also allows many possible designs to be simulated by considering how they will effect the timing of the interactions between the physical and software systems. For example, if the software is to be run on a multi-threaded operating system, the boolean condition could consist of a timing profile which emulates at what times the BT scheduler will be executed. This timing profile could be randomised to determine how the system operates under different loads, or may just address one specific or worst-case scenario. If more than one operating system is being considered, a timing profile could be setup for each operating system and multiple simulations peformed to determine the differences, if any, on the system as a whole.

5. Case Study: An Automated Train Protection System

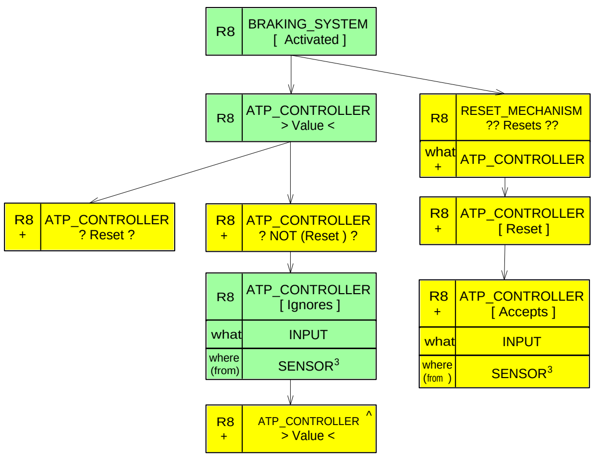
Most rail systems have some form of train protection system that use track-side signals to indicate potentially dangerous situations to the driver. The simplest train protection systems consist of signals with two states: green to continue along the track and red to apply the brake to stop the train. More sophisticated systems include detailed information such as speed profiles for each section of the track.

Accidents still occur using a train protection system when a driver fails to notice or respond correctly to a signal. To reduce the risk of these accidents, Automated Train Protection (ATP) systems are used that automate the train's response to the 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.

The ATP system used for this paper has three track-side signals: proceed, caution and danger. When the ATP system receives a caution signal, it monitors the driver's behavior to ensure the train's speed is being reduced. If the driver fails to decrease the train's speed after a caution signal or the ATP system receives a danger signal then the train's



(a) RBT for Requirement 6



(b) RBT for Requirement 8

Figure 4. Example Requirement Behavior Trees of the ATP System

brakes are applied. The complete requirements of the ATP system can be found in Table 1. The requirements of the ATP system have been used previously in related work to demonstrate composition of components using exogenous connnectors [10].

Section 5.1 discusses the construction of the BE model of the ATP system from the requirements and Section 5.2 discusses the Modelica model of the ATP systems physical components and environment.

5.1 ATP - Behavior Engineering Model

Figure 4 shows two example RBTs of the ATP system. 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 condition 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. Relations should be read as questions that can be asked of the primary behavior, which the associated relational behavior answers. For example, "Where is the Alarm enabled? Within the Driver's Cab". Capturing the information about the Driver's Cab ensures that the original intent of the requirements is not removed. The next BT node assumes that it is implied that the ATP Controller observes 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.

During integration of the RBTs of the ATP system the following problems were found:

- *Conflicting Behavior (R7-R8).* After the Braking System is activated, R7 states that a proceed signal disables the Alarm whereas R8 states all sensor input is ignored until the ATP Controller is reset.
- *Conflicting Behavior (R7-R9).* After the Braking System is activated, R7 states that a proceed signal disables the alarm whereas R8 states that the Reset Mechanism deactivates the Train's Brakes and disables the Alarm.
- *Missing Behavior (R6).* What should the ATP Controller do if the Train's speed is observed to be decreasing?
- *Missing Behavior.* What should the ATP Controller do if an undefined signal is returned to the ATP Controller?

Each of these problems would need to be resolved with the client to ensure that the system behaves as is desired. However for the purposes of this case study the following assumptions were made:

- R8 and R9 were given priority over R7. That is, a proceed signal can only disable the Alarm after the Alarm has been enabled but prior to the Brakes being activated. After the Brakes have been activated all sensor input is ignored until the ATP Controller is reset. Also, resetting the ATP Controller after the Brakes have been activated causes the Train's Brakes to be deactivated and disables the Alarm.
- If the ATP Controller observes the train's speed to be decreasing then: if a danger signal is received the Brakes are immediately activated; or, if a proceed signal is received the Alarm is disabled. However, if the train's speed increases before either of these signals are received then the ATP Controller should activate the Train's Braking System.

Figure 5 shows the DBT of the ATP system resulting from design decisions made to the MBT. A (M) in the tag shows the nodes of the DBT where interaction occurs with the Modelica model. The following design decisions were made to the MBT:

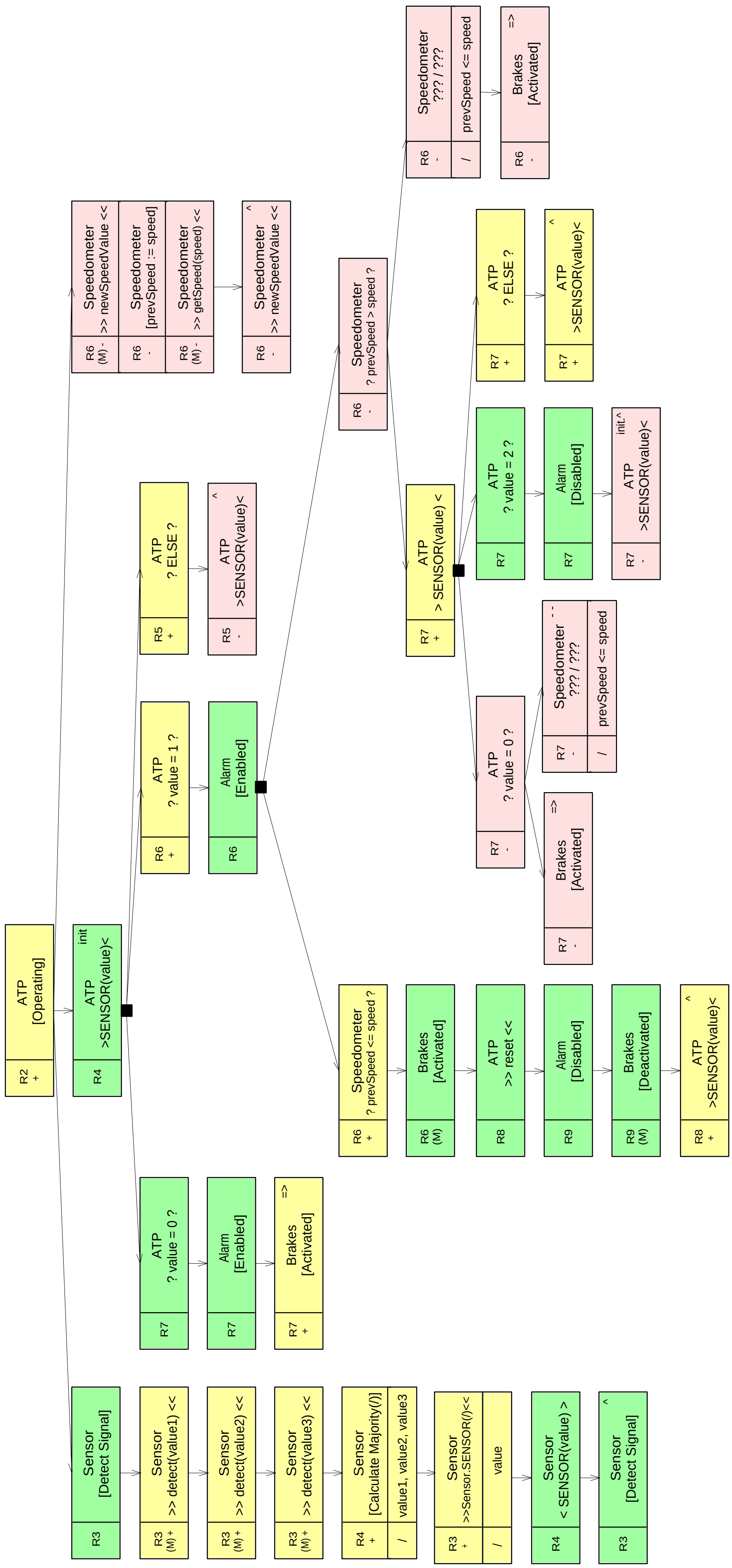


Figure 5. Design Behavior Tree of ATP system

- Train, Signal, the individual Sensors, Driver's Cab, Reset Mechanism, and Noise components are outside the boundaries of the DBT.
- A Speedometer component is required to receive the train's speed and store the previous speed value so that changes in the speed of the train may be determined.
- Alternative branching and atomic composition was added to ensure appropriate threaded behavior. Atomic composition is required for when the speedometer component's speed value is updated. This is because for a small period of time the current speed equals the previous speed causing the prevSpeed<=speed guard to evaluate to true, regardless of the new speed value. Alternative branching ensures that once one of the mutually exclusive branches has been taken (e.g. value=0), none of the other branches can be executed (e.g. value=1, ELSE).

5.2 ATP - Modelica Model

The Modelica model describes the physical components that make up the environment in which the ATP system will operate. It consists of components such as the Train, the Driver, the Train Track, and the Sensors of the track-side signals. Figure 6 shows the component diagram of the Modelica model. The Driver component is responsible for controlling the Train's speed and resetting the ATP system. The Train component simulates its velocity and position on the Track based upon its mass, maximum acceleration power and maximum brake force. The Train Track provides the signal sensors with the signal value at the signal position. The signal sensors then simulate the presence of noise, occasionally misreading a signal value. The sensor values, Train speed and driver reset are all provided to the ATP controller which in turn provides whether to apply the Train's Brakes. A simplified version of the Modelica textual model of the ATP environment is shown in Figure 7.

5.3 Integration of the Modelica and BE Models

Simulating the integrated Modelica/BE models provides plots which graphically show the interactions between software and hardware in reference to time. This allows

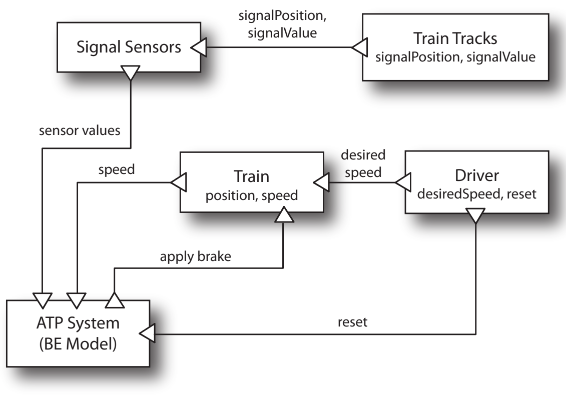


Figure 6. Component diagram of the Modelica ATP Environment model

the investigation and documentation of scenarios in a clear way. The types of scenarios that can be investigated are:

- The frequency of the execution of the BE model relative to the Modelica model simulates the performance capabilities of the hardware platform on which the BE model will be deployed.
- The sampling frequency/response time of sensors and actuators can be simulated by the frequency of interaction between the Modelica model and the BE model.
- The system can be tested with different Trains, Drivers, Train Tracks, etc.

Figure 8 shows four example simulations of the integrated model of the ATP System. All the simulations are performed with a train model based on a British Rail Class 57 diesel locomotive, which has a mass of 120 tonnes, a maximum speed of 120.7 km/h, a maximum brake force of 80 tonnes and a power at rail of 1860 kW with an assumed 80% efficiency due to losses in pressure and friction.

The train's braking time of two seconds is due to its low velocity (approximately 45km/h) and small weight due to the absence of carriages. The same train operating at 100 km/h would take approximately eight seconds to brake, and at 200 km/h would take 32 seconds. The addition of carriages would further increase the time the train would take to brake to a complete stop. These braking times highlight the need to test software-hardware integration under numerous circumstances.

The simulations performed on this case study show the ATP system operating with the same configuration of sensors, actuators and hardware platform. The change that is tested is the driver's response to the signals on the track, the results of which now ensures that the ATP system is functioning as specified by the requirements. Further simulations could now be performed to investigate the ATP system operating both in different scenarios and also the suitability of different sensors, actuators and hardware platforms.

6. Conclusion

This paper investigates the software/hardware integration problem caused by the increasing codependancy of software and hardware in large-scale systems. An integrated approach is described, which integrates separate software and hardware models to aid the investigation of software/hardware interaction through simulation. An ATP system is used as a case study to describe both separate software/hardware modelling with BE and Modelica and software/hardware integration and investigation. This integrated approach allows various software/hardware interactions to be investigated such as software execution speed, sensor sampling frequencies, and actuator response times. It also provides a graphical and documentable output of the investigation the behavior of the software and hardware in different scenarios.

Acknowledgments

This work was produced with the assistance of funding from the Australian Research Council (ARC) under the ARC Centres of Excellence program within the ARC Centre of Complex Systems (ACCS), the Swedish Vinnova under the Safe and Secure Modeling and Simulation project and the Swedish Research Council (VR).

References

[1] Robert Colvin and I. J. Hayes. A Semantics for Behavior Trees. ACCS Technical Report ACCS-TR-07-01, ARC Centre for Complex Systems, April 2007.

[2] Tullio Cuatto, Claudio Passeronge, Luciano Lavagno, Attila Jurecska, Antonino Damiano, Claudio Sansoè, A. Sangiovanni-Vincentelli, and Alberto Sangiovanni-Vincentelli. A case study in embedded system design: an engine control unit. In *Proceedings of the 35th annual conference on Design automation (DAC '98)*, pages 804–807, New York, NY, USA, 1998. ACM.

[3] R. G. Dromey. Formalizing the Transition from Requirements to Design. In Jifeng He and Zhiming Liu, editors, *Mathematical Frameworks for Component Software - Models for Analysis and Synthesis*, pages 156–187. World Scientific Series on Component-Based Development, 2006. Invited Chapter.

[4] R.G. Dromey. From Requirements to Design: Formalizing the Key Steps. In *IEEE International Conference on Software Engineering and Formal Methods*, pages 2–11, Brisbane, Sept 2003. SEFM-2003. Invited Keynote Address.

[5] Dynasim. Dymola. http://dynasim.com.

[6] Rolf Ernst. Codesign of embedded systems: Status and trends. *IEEE Design and Test*, 15(2):45–54, 1998.

[7] Peter Fritzon, Vadim Engelson, Andreas Idebrant, Peter Aronsson, Håkan Lundvall, Peter Bunus, and Kaj Nyström. Modelica âĂŞ A Strongly Typed System Specification Language for Safe Engineering Practices. In *Proceedings of the SimSAFE Conference*, Kralskoga, Sweden, June 2004.

[8] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, 2004.

[9] Tarek Ben Ismail, Mohamed Abid, and Ahmed Jerraya. Cosmos: a codesign approach for communicating systems. In *Proceedings of the 3rd international workshop on Hardware/software co-design (CODES '94)*, pages 17–24, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[10] Kung-Kiu Lau, Ling Ling, and Zheng Wang. Composing Components in Design Phase using Exogenous Connectors. In *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO '06)*, pages 12–19, 2006.

[11] MathCore. Mathmodelica. http://www.mathcore.com.

[12] Modelica Association. Modelica: A Unified Object-Oriented Language for Physical Systems Modeling: Language Specification Version 3.0, Sept 2007. http://www.modelica.org.

[13] OMG. System Modeling Language (SysML). http://www.omgsysml.org.

[14] Adrian Pop, David Akhvlediani, and Peter Fritzon. Integrated UML and Modelica System Modeling with ModelicaML in Eclipse. In *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications (SEA'07)*, 2007.

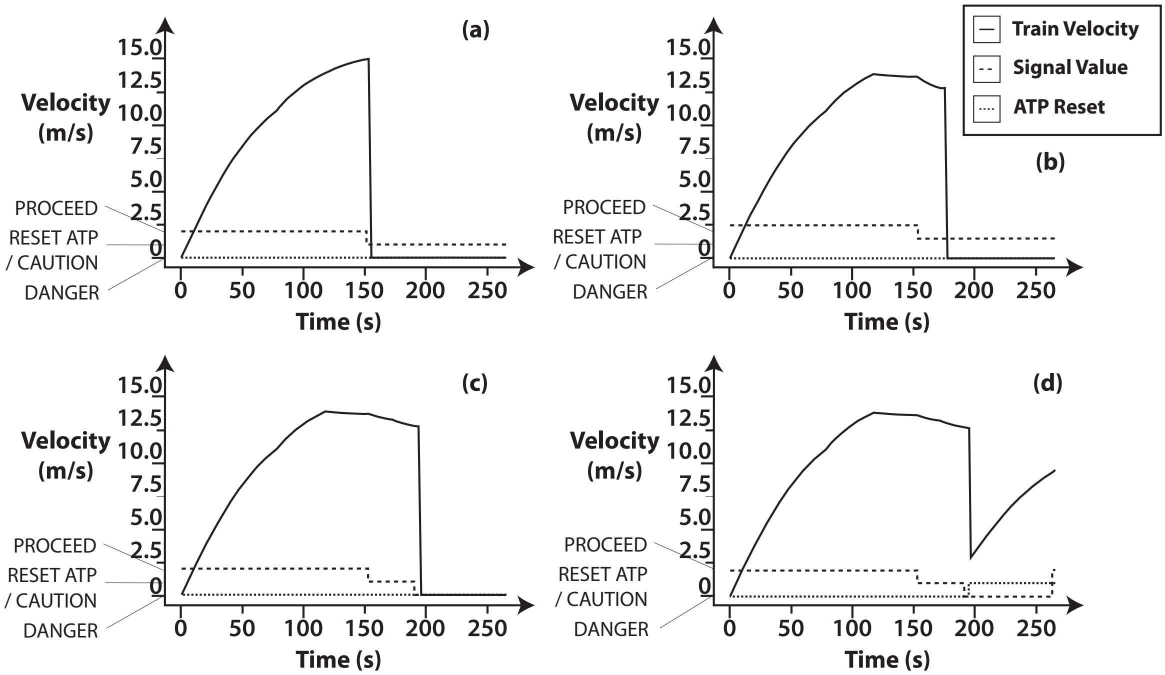
[15] Adrian Pop, David Akhvlediani, and Peter Fritzon. Towards Unified System Modeling with the ModelicaML UML Profile. In *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools (EOOLT'07)*, pages 13–24, 2007.

[16] Danny Powell. Requirements evaluation using behavior trees - findings from industry. In *Australian Software Engineering Conference (ASWEC'07)*, April 2007.

[17] Michael Tiller. *Introduction to Physical Modeling with Modelica*. Kluwer Academic Publishers, 2001.

```
// External Functions included here

model Track
  discrete Integer currentSignalValue "Value of Last Signal displayed to Driver/ATP System";
  parameter Real[:] signalPosition "Positions of Signals on the Track";
  parameter Integer[:] signalValue "Values of Signals on the Track";
equation
  // Determine current signal value
end Track;

model Train
  Real s, v, m, maxSpeed, maxBrakeForce, maxAccelerationPower, maxAccelerationForce;
  parameter Real accPowerEff = 0.80 "Engine Efficiency in %";
equation
  maxAccelerationPower/accPowerEff = maxAccelerationForce*v;
end Train;

record Driver
  Real desiredAcceleration;
  parameter Real[:] desiredSpeed;
  parameter Real[:] position;
end Driver;

model Main
  // Define track, train, driver parameters

  parameter Real[10] sensor1 = {0,0,1,2,0,0,2,2,0,0} "Sensor1 value at signalPosition";
  Real sensor1Reading "Current Sensor1 reading";
  // Similar for Sensor 2 & 3

  Real fa, fd, doBrake(start=0), minAccelerationForce, desiredAccelerationForce;
  discrete Boolean clock1, clock2, ...;
  // Define clock frequencies

equation
  when initial() then startBT(0); end when;
  when clock1 then cycleBT(0); end when;
  when clock2 then doBrake = if (train1.v >= 0) then getBrake(0) else 0;
  // if driver reset's ATP send message
  // if signal changes send new sensor values

  fa = if doBrake>0 then 0
  elseif // ensure not over maximum Acceleration force
  else desiredAccelerationForce;
  fd = if doBrake>0 then train1.maxBrakeForce else 0;

  a = (fa-fd)/train1.m;
  der(v) = a;
  der(track1.s) = train1.v;
  // if train passing signal then update sensors
  // determine driver's desired acceleration (a = (desiredSpeed - train1.v)/ (2*distance))
end Main;
```

Figure 7. Simplified Textual Modelica model of the ATP Environment



[(a)] Driver ignores caution signal and increases speed, brakes are activated
[(b)] Driver sees caution signal and reduces speed but then increases speed, brakes are activated
[(c)] Danger signal, brakes are activated regardless of driver already decreasing speed
[(d)] Danger signal, brakes are activated, ATP is reset and brakes are deactivated

Figure 8. Simulation of the ATP System

Requirement	Description
R1	The ATP system is located on board the train. It involves a central controller and five boundary subsystems that manage the sensors, speedometer, brakes, alarm and a reset mechanism.
R2	The sensors are attached to the side of the train and detect information on the approach to track-side signals, i.e. they detect what the signal is displaying to the train driver.
R3	In order to reduce the effects of component failure three sensors are used. Each sensor generates a value in the range 0 to 3, where 0, 1 and 2 denote the danger, caution, and proceed signals respectively. The fourth sensor value, i.e. 3, is generated if an undefined signal is detected, e.g. may correspond to noise between the signal and the sensor.
R4	The sensor value returned to the ATP controller is calculated as the majority of the three sensor readings. If there does not exist a majority then an undefined value is returned to the ATP controller.
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.
R7	In the case of a danger signal being returned to the ATP controller, the braking system is immediately activated and the alarm is enabled. Once enabled, the alarm is disabled if a proceed signal is subsequently returned to the ATP controller.
R8	Note that if the braking system is activated then the ATP controller ignores all sensor input until the system has been reset.
R9	If enabled, the reset mechanism deactivates the train's brakes and disables the alarm.

Table 1. Requirements of the ATP system