Information and Software Technology 61 (2015) 52–70

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

# Requirements simulation for early validation using Behavior Trees and Datalog

Saad Zafar, Naurin Farooq-Khan *, Musharif Ahmed

*Faculty of Computing, Riphah International University, Islamabad, Pakistan*

ARTICLE INFO

*Article history:*
Received 2 June 2014
Received in revised form 1 November 2014
Accepted 9 January 2015
Available online 21 January 2015

*Keywords:*
Formal methods
Behavior Trees
Datalog
Formal analysis
Simulation
Requirements engineering

ABSTRACT

*Context:* The role of formal specification in requirements validation and analysis is generally considered to be limited because considerable expertise is required in developing and understanding the mathematical proofs. However, formal semantics of a language can provide a basis for step-by-step execution of requirements specification by building an easy to use simulator to assist in requirements elicitation, validation and analysis.
*Objective:* The objective of this paper is to illustrate the usefulness of a simulator that executes requirements and captures system states as rules and facts in a database. The database can then be queried to carry out analysis after all the requirements have been executed a given number of times
*Method:* Behavior Trees (BTs)[1] are automatically translated into Datalog facts and rules through a simulator called SimTree. The translation process involves model-to-model (M2M) transformation and model-to-text (M2T) transformation which automatically generates the code for a simulator called SimTree. SimTree on execution produces Datalog code. The effectiveness of the simulator is evaluated using the specifications of a published case study – Ambulatory Infusion Pump (AIP)[2].
*Results:* The BT specification of the AIP was transformed into SimTree code for execution. The simulator produced a complete state-space for a predetermined number of runs in the form of Datalog facts and rules, which were then analyzed for various properties of interest like safety and liveness.
*Conclusion:* Queries of the resultant Datalog code were found to be helpful in identifying defects in the specification. However, probability values had to be manually assigned to all the events to ensure reachability to all leaf nodes of the tree and timely completion of all the runs. We identify optimization of execution paths to reduce execution time as our future work.



## 1. Introduction

### 1.1. Motivation

Formal methods are known for their vigor in carrying out formal analysis of requirements [1]. They can play an instrumental role in the early detection of defects and stop them from causing a ripple effect in software development projects. The spectrum of formal methods can span from light-weight formal analysis to more rigorous model-checking and theorem-proving [1]. The process of formal analysis is not only resource intensive but also requires specialized knowledge for writing and analyzing the requirements, which is often not readily available [2–4]. The mathematical nature of formal specification and formal analysis also makes the requirements elicitation and validation a difficult process. The formally specified requirements can be executed to generate traces and validate various scenarios [5]. The simulated requirements can help improve the understanding of all stakeholders by not only executing different paths of execution but also checking for specific conditions [6]. To this end, we propose simulation of Behavior Trees (BTs) – a graphical formal notation [7–9] – to support early validation of requirements. The simulation can be used to improve understanding as well as analysis of the requirements. More rigorous analysis of BT specification is already supported through translation of BTs into other formal languages with tool supported model-checking and theorem-proving. The proposed simulator (SimTree) enables the execution of requirements to help facilitate requirements elicitation, analysis and validation. At present, SimTree is configured to produce Datalog rules and facts for a given BT. The mass storage facility of Datalog allows for capturing the simulation results. The rules and facts can be analyzed by forming Datalog queries. The advantage of translating BTs into

* Corresponding author at: Faculty of Computing, Riphah International University, I- 14, Islamabad, Pakistan. Tel.: +92 51 8446000 8x251.
*E-mail addresses:* saad.zafar@riphah.edu.pk (S. Zafar), naurin.zamir@riphah.edu.pk (N. Farooq-Khan), musharif.ahmed@riphah.edu.pk (M. Ahmed).

[1] BTs – Behavior Trees.
[2] AIP – Ambulatory Infusion Pump.

http://dx.doi.org/10.1016/j.infsof.2015.01.005
0950-5849/

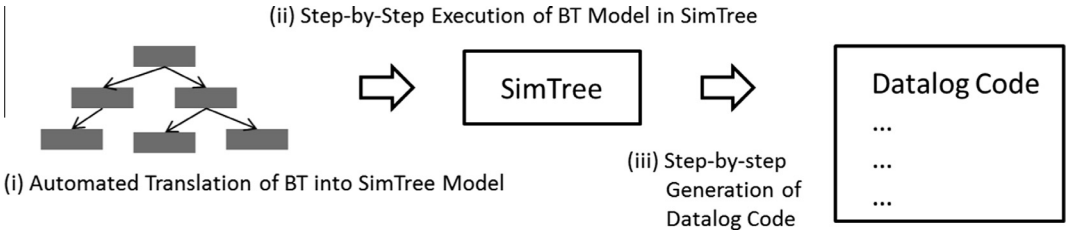



**Fig. 1.** Automated transformation of BTs specification into Datalog code using SimTree.

Datalog over other approaches is that the rules and facts that together form the model of any Datalog program can be subject to incremental increase while preserving the P-time termination of the queries. The bottom up evaluation method makes sure that the queries always terminate and ensures the tractability of the Datalog program. The semantics of Datalog makes it a good candidate for reasoning and makes it amenable to a rich analysis. Therefore the resulting deductive database can be used to carry out analysis by forming queries about critical aspects of the requirements. The formal analysis may include checking for safety properties, liveness, or deadlocks. Traces of specific paths may also be generated. As a result the elicitation and early validation of the requirements can be carried out for a desired state-space of the system. If preferred the state-space may be expanded; the simulator is able to simulate for a predefined number of runs thereby incrementally increasing the size of the Datalog model. Once enough confidence is gained through simulation we can do model checking and exhaustively verify the state-space of the system using existing BT tools and model checking support.

### 1.2. Overview

In this paper, we present the simulation of the Behavior Trees (BTs) notation by automatically transforming its core components into SimTree – to assist not only in early validation of the requirements but also in elicitation of requirements through their execution. Behavior Trees is a graphical tree-like specification language which has formally defined semantics [10,11]. The graphical form of BTs is aimed at making the formal specification easy to use and easy to validate. The specification language is accompanied by a well-defined *Behavior Engineering* (BE)[3] process [12] that is useful in creating the specification from informally specified requirements [13]. The BE process starts by the translation of individual requirements one by one in the form of a corresponding *Requirements Behavior Tree* (RBT)[4]. Once all the requirements are translated, all the independent RBTs are integrated into a single *Integrated Behavior Tree* (IBT)[5]. The IBT can then be analyzed for completeness and correctness and can be used to derive system design. This process of translation has been found useful during requirements validation and early detection of defects in industrial trials of the technique [14]. A BT specification can be formally verified through automated translation of BTs into other formal analysis environments like Symbolic Laboratory Analysis (SAL) [10] and Communicating Sequential Processes (CSP) [11]. To further enrich the BTs analysis of the requirements specification, we propose execution of BTs through a simulator SimTree with the objective of supporting not only early validation of requirements but also the elicitation process.

We use ATLAS Transformation Language (ATL) [15,16] to derive a mapping between BTs components and the components of the simulator. ATL is a model transformation language that can be used to produce a target model from any source model. An ATL program uses predefined rules to match and navigate various source model elements and then initializes the respective elements in the target model. Using the principles of Model Driven Engineering (MDE), we choose model-to-model (M2M) and model-to-text (M2T) [17] transformations for automatic generation of code for the simulator. We use a previously defined BT meta-model [18] as a source model. The source model is transformed into the simulator's target model. This target model is then used to produce java code for the simulator, which we refer to as SimTree. The execution of SimTree can help identify new requirements by analyzing different execution paths and may also help in finding elusive defects that are manifested in concurrent processes. SimTree can be configured to generate Datalog [19,20] rules and facts in a step-by-step fashion as it executes a given BT. The resultant Datalog code can then be queried for further analysis and for checking various properties of interest.

Datalog is a declarative logic programming language that is derived from Prolog [21]. It is a query language, which has a relatively simple syntax and semantics [19]. The query evaluation in Datalog is based on first order logic [22] that makes it logically sound and complete. The queries in Datalog on finite sets are also guaranteed to terminate [23]. At present, the simulator is configured to produce Datalog rules and facts for a given BT. The resulting deductive database can be used to carry out analysis by forming queries about critical aspects of the requirements. The formal analysis may include checking for safety properties, liveness, or deadlocks. Traces of specific paths may also be generated or values in the dataset may be altered to perform a wide array of analysis such as Failure Mode and Effect Analysis (FMEA) [24] to assist not only in identification of defects but also in elicitation of requirements. The simulator can also be enhanced to visualize step-by-step execution of a given BT. We use a published case study Ambulatory Infusion Pump (AIP) [25] to demonstrate the usefulness of our approach. Fig. 1 conceptually illustrates the transformation of BTs into Datalog code.

### 1.3. Paper Structure

Section 2 gives a concise introduction to the Behavior Trees (BTs) notation, its underlying semantics and the *Behavior Engineering* (BE) process followed by Datalog. It also gives a brief introduction to the AIP case study. Section 3 presents the related work. In Section 4 we illustrate the translation of BTs into Datalog using M2M transformation and M2T transformation to generate code for the simulator. In Section 5 we discuss the analysis done with Datalog queries using our simulator SimTree. We conclude our work in Section 6.

## 2. Background

### 2.1. Behavior Trees

Behavior Trees is a formal specification language used for capturing natural-language requirements in a graphical format. It is one of the three representations of the behavior modeling language (BML)[6] – the other two being Composition Trees and Structure Trees.

[3] BE – Behavior Engineering.
[4] RBT – Requirement Behavior Tree.
[5] IBT – Integrated Behavior Tree.
[6] BML – Behavior Modeling Language.

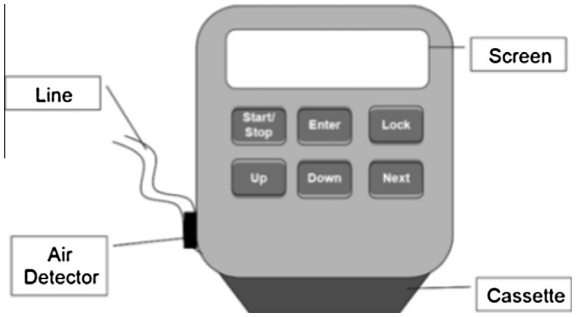



**Fig. 2.** Ambulatory infusion pump (reproduced from [25]).

BTs capture the dynamic aspect of the system. The tree-like graphical notation is aimed at making the specifications easy to read and write. The notation also covers the core building blocks of a language i.e. sequence, selection, iteration, data flow and assignment. In doing so, it aims to relieve the reader from the clutter of intricate connections, the likes of which we find in a network structure. The core Behavior Trees syntax [7] includes BT nodes of different types that can be composed sequentially, atomically and concurrently. The different node types are used to specify component states, selection and events. The BTs syntax and semantics is summarized in Section 2.3. The process of developing the BT specification is often referred to as *Behavior Engineering* (BE). BE supports a rigorous process that translates a given informal requirement into its close BT representation. The process is repeated for all the informally stated requirements. Any ambiguous or missing requirements are marked within the BT specification for easy identification during the validation phase. Once all the requirements have been translated, they are integrated into a single tree-like structure called *Integrated Behavior Tree* (IBT). IBT provides the modeler an integrated view of the requirements and any impact of changes made to the IBT can be readily visualized. The process has been found beneficial in the industrial trials of BTs [14]. Over the course of time, BTs have been enhanced to include probabilistic and timed Behavior Tree and execution of requirements using state-machines [26,12,27]. Other related work also include a theoretical model for simulation of requirements [28]. BTs have also been extended to support modeling of Role-based Access Control Requirements [25,29].

### 2.2. Ambulatory Infusion Pump (AIP) – A case study

This section introduces the case study that is used as a running example in this paper – the Ambulatory Infusion Pump (AIP). The study derived from [25], is based on a real safety–critical device that is used to deliver measured doses of a drug to patients while they are away from direct care of health professionals. The device is programmable to allow healthcare professionals to configure it to meet the patient's drug therapy needs. The AIP has an in-built pump that delivers the drug to the patient based on the programmed infusion rate. The device connects to the patient through a line with a needle assembly (see Fig. 2). It also has an occlusion sensor to detect any blockage of the line and an air detector to detect any air in the line. A voltage detector is used to monitor the status of the batteries. A screen is used to display important programming information or critical messages to the users. A beeper is used as an alarm whenever human attention is required. Table 1 lists simplified requirements of the device. Being a safety critical device, the AIP must exhibit certain safety properties. In Section 5 we use it as proof of concept and show the results of our simulations.

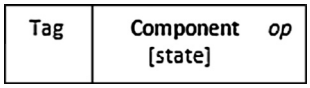



**Fig. 3.** A Behavior Tree node.

### 2.3. Behavior trees notation

A BT consists of nodes, which are composed together using edges to form a tree-like structure. At the most basic level, a node represents a *component* and its *state* (see Fig. 3). A component *AIP* that is in the On state (`AIP [On]`) is represented as illustrated in Fig. 4. To assist in traceability, the node's tag can be labeled with the corresponding natural language reference (e.g. `Req 1.1`). Alternatively, a node can be used to represent an *event*, a *guard* and a *selection* statement. A component *Sensor* can wait for an external event *Air Detected* (`Sensor ??Air_Detected??`). That is to say, the node would be executed if and when the external event becomes true (Fig. 4(b)). In comparison, a guard (e.g. `AIP ???On???`) waits for an internal state to become true (Fig. 4(c)) and the control is passed on to the next node if and when the required internal state becomes true. On the other hand, a *selection* node (e.g. `AIP ?Volume=0?`) acts as an *if* statement. The control is only passed on to the next node if the condition holds, otherwise the thread is terminated (Fig. 4(d)). A node may have an operator to indicate *reversion* ('^'), *synchronization* ('@'), *branch kill* ('–'), *goto* ('=>') or a *new thread* ('^^'). Reversion in BTs is used as a leaf node to indicate that the control flow reverts to a matching node higher in the same branch and as a result all the active threads in the path are terminated (Fig. 4(e)). The synchronization symbol is used to synchronize two parallel threads at the marked node (Fig. 4(i)). The branch kill operator is used to terminate an active branch (Fig. 4(f)) and the *goto* operator is used to indicate passing of control flow to the indicated node anywhere in the tree (Fig. 4(g)). The new thread symbol is used to spawn a new thread from the point of the matching node higher in the same branch (Fig. 4(h)). BT nodes can be composed together using atomic or sequential edges. The sequential edge is represented by a line with an arrow to indi-

**Table 1**
AIP functional requirements.

| | |
|---|---|
| Req 1 | The AIP system is turned on when the batteries are put in and is turned off when the batteries are out |
| Req 2 | To start the pump, when in stopped state, the start-stop button is held down until it beeps three times and dots are displayed on the screen |
| Req 3 | To stop the pump, when in running state, the start-stop button is held down until it beeps three times and dots are displayed on the screen |
| Req 4 | When the battery charge is low, the AIP system sends three beeps and displays battery low message on the screen |
| Req 5 | Every time the system pumps 1 ml of drug when the battery is low it sends a single beep alarm |
| Req 6 | After a set time the pump is activated by the AIP system to pump 1 ml of drug |
| Req 7 | When the volume reaches 5 ml the system emits three beeps and displays the volume low message every 1 ml as it counts down to empty |
| Req 8 | When there is no drug left, the pump enters stopped mode and the system sends a signal to the beeper to sound a continuous beeping alarm |
| Req 9 | When the line is closed/blocked or kinked the system emits a constant beeping alarm if it is in the running mode |
| Req 10 | Whenever air is detected in the line the pump is stopped and the beeper sounds a continuous beep |
| Req 11 | The main screen displays the pump status (running/stopped), battery status (normal/low) and drug volume |
| Req 12 | If no key is pressed for 2 min then the screen is reset to the main screen |

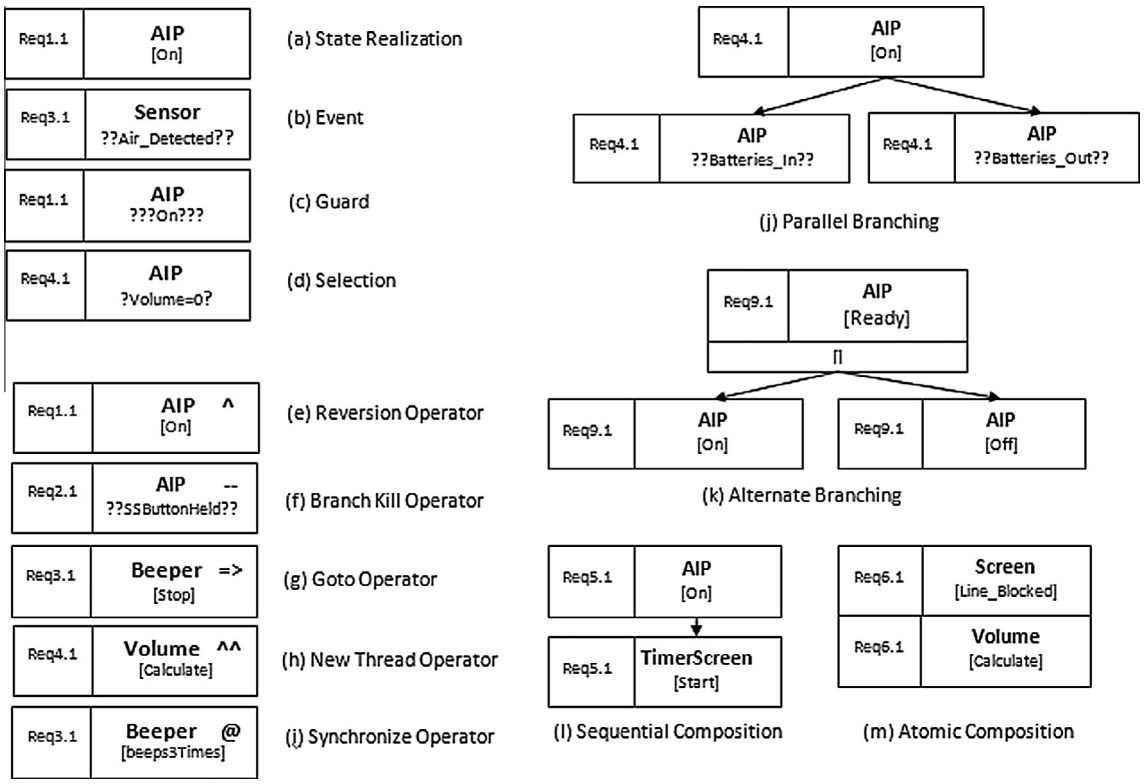



**Fig. 4.** Core elements of BTs notation.

cate the flow of control from the node where the line originates to the node pointed by the arrow (Fig. 4(l)). The atomic nodes are conjoined together. The atomic composition of two or more nodes makes the path uninterruptable once the control is passed on to the first node (Fig. 4(m)). No competing thread can interleave until all the atomically composed nodes are executed. Two or more branches can also be composed together either as parallel or alternate branches. Parallel branches in a tree are composed using normal edges (Fig. 4(j)). However, special symbol ('`[]`') is used to denote alternate branching (Fig. 4(k)). That is, the path between two or more branches is chosen non-deterministically.

### 2.4. Behavior Engineering process

The *Behavior Engineering* (BE) process is used for *translation* of natural language requirements into a formal representation using BTs [29]. The main objective of the process is to stay close to the intent of the requirements and to bridge the gap between informal and formal specification of requirements. The detailed translation process involves the identification of components, their states and how two or more components are composed together. Early defect detection is supported by marking ambiguous behavior in the requirements with a *plus* '+' sign and any missing behavior that the modeler has included to complete the translation, is marked with a *minus* '–' sign along with the requirement tag. Alternatively, nodes can be color coded: green for no defect, yellow for implied behavior and red for any missing requirement. The marked implied or missing requirements are rectified during the validation process. This defect detection process has been proved useful during the industrial trials [14]. Once the individual requirements have been translated as individual RBTs, all the requirements are grafted together in a single IBT. The integrated view can be useful for identification of integration defects along with the identification of any incompleteness in requirements. Also, the impact of any change can be readily visualized by the modeler during requirements correction or refinement. BE endorses a scalable and repeatable methodology by tightly interlinking the BML and Behavior Modeling Process (BMP).[7] BMP involves four different phases: The first phase is the formalization in which natural language requirements are translated into desired trees. They are integrated in the second phase called Fitness-for-Purpose Test followed by locating and correcting defects in the specification phase. The fourth and final phase is the design in which specifications are refined by taking design decisions. Furthermore IBT refinement patterns have also been proposed to assist in analysis and design of the system under consideration The details of the BMP and the refinement patterns are beyond the scope of this work, interested readers are directed toward [18,25] for further reading.

As an example, consider a simple requirement from the Ambulatory Infusion Pump (AIP) case study [25] used in this paper. The first requirement (Req 1) states that *the AIP system is turned on when the batteries are put in and is turned off when the batteries are out*. This statement may be translated as illustrated in Fig. 5(a). The root node in the example represents a component *AIP* in the state *Off* (textually `AIP [Off]`). The next node sequentially composed with the first one, is an event called *AIP Batteries_in* (textually `AIP ??Batteries_in??`). If and when the event becomes true, the BT branches into two sub-trees composed together as parallel branches. Semantically both the branches execute in parallel. The first node on the left is another event *AIP Batteries_out* (`AIP ??Batteries_out??`). While the system is waiting for the event *Batteries_out*, the AIP is in the *On* state in parallel. If and when the event occurs, the AIP is reverted back to the *Off* state in the next step. This behavior is represented by the reversion node at the bottom left of the tree. The reversion causes all the threads executing in parallel to be terminated. It is important to

[7] BMP – Behavior Modeling Process.

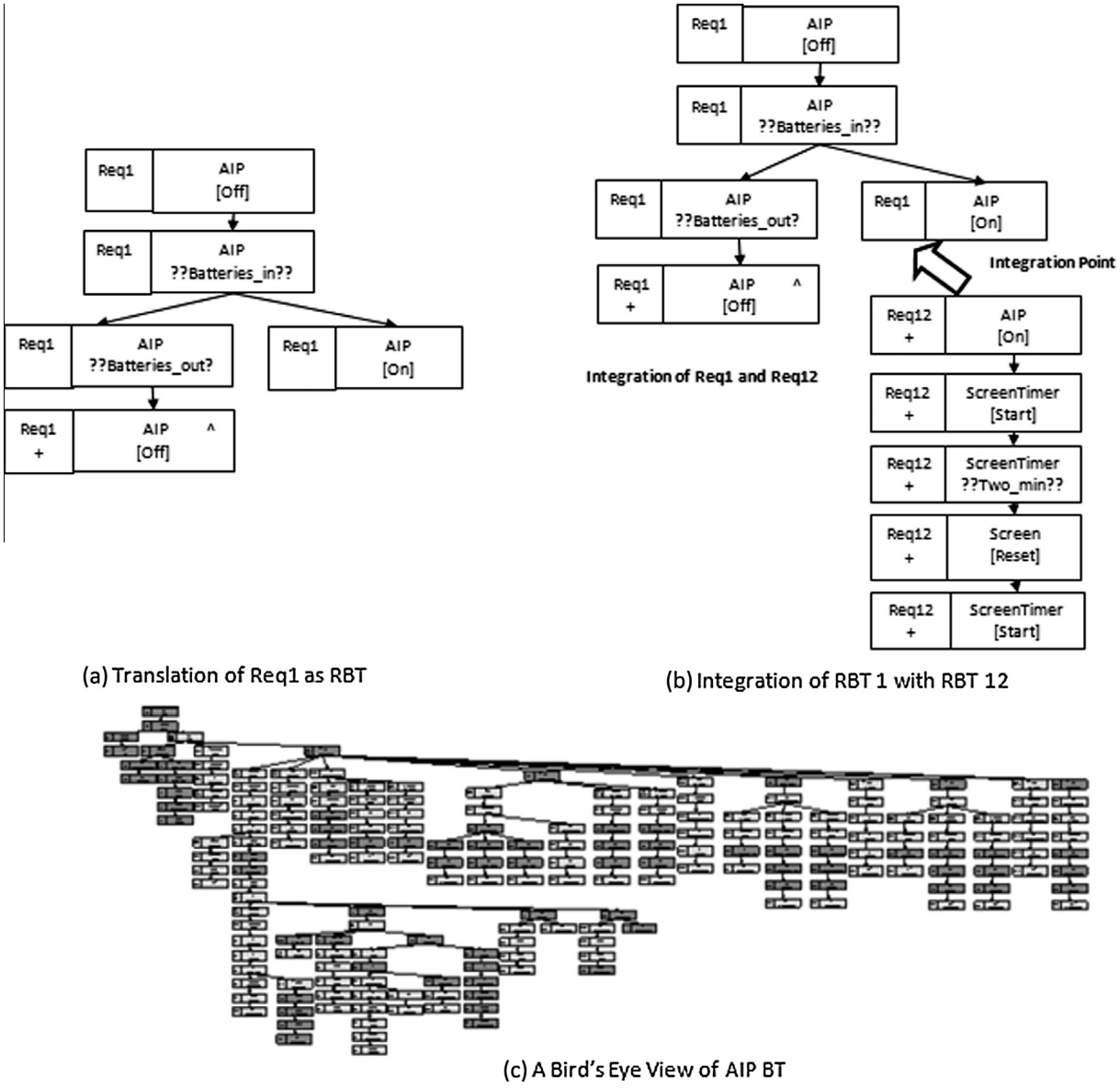



**Fig. 5.** BE process – (a) requirements translation, (b) requirements integration, (c) integrated view of requirements.

note that the reversion behavior is not explicitly stated in the requirement. Therefore, the node has been marked as implied behavior using the plus sign along with the requirement's traceability tag. Once all the requirements have been translated into respective RBTs they are grafted together during the requirements integration process. The process bears the analogy of putting pieces of jig-saw-puzzle [9] together by matching the pre-condition of a requirement with the post-condition of another requirement as illustrated in Fig. 5(b). The process is useful for identification of integration defects. All the implied and missing behavior is validated by systematically reviewing the IBT. The next step is the integration of a given set of requirements into a system design by evolving an IBT into a Design Behavior Tree (DBT).[8] The design patterns are applied and the IBT is checked for any completeness and correctness defects. The tree is also inspected for appropriate reversion points, causal behavior and protection of critical regions among other concerns. The suggested analysis and design patterns can be found in [25] which the interested reader can refer to for further reading.

The completed IBT/DBT can be translated into other formal specification languages through tool support [10,28,27,26,30]. The Symbolic Analysis Laboratory (SAL) [31] is one such language for which the tool support is available. SAL provides a rich framework for formal analysis of specification specified in its input language. The SAL environment provides not only model-checking [32] support but also simulation, deadlock checking, and well-formedness checking to name a few. Model-checking of the specification involves satisfaction of bounded and unbounded Linear Temporal Logic (LTL) [33] formulae. To translate the BT specification into the SAL input language, a set of syntax rules along with the corresponding translation scheme for each of the rules, has been developed [10]. A BT specification is represented in a single SAL transition system module. The components, their states, and messages are declared as variables in the module. The BT nodes are then translated as transitions in the SAL language. The state-realizations in BT nodes are represented as variable updates. The BT guards and selections are used to test the state of the variables.

[8] DBT – Design Behavior Tree.

A set of special variables $pc_1, \ldots, pc_n$ (program counters) is used to keep track of concurrent state transitions in the tree. These variables are also used for controlling branching and the termination of threads in BTs. The translated SAL code can be fed into many tools available in the SAL environment for formal analysis.

### 2.5. Datalog

Datalog is a rule based query language that is declarative and is a subset of first order logic [19]. Its syntax is simple and easy to understand. Another attractive feature of Datalog is its clean semantics that allow for better reasoning pertaining to problem specification. The semantics provide more expressivity and make it amenable to rich analysis. A powerful feature of Datalog is that query termination occurs in polynomial time. A Datalog program comprise of rules and facts that are represented as a Horn clause of the form

$$\boldsymbol{L0} : -\boldsymbol{L}1, \ldots\ldots.., \boldsymbol{Ln}$$

The left side of the Horn clause is called its head while the right side is called its body. ***Li*** is a literal of the form $pi(t1, \ldots, tk)$ where $n \leqslant i \geqslant 1$. ***pi*** is the predicate symbol and ***tj*** are the terms with $k \leqslant j \geqslant 1$. The number of terms of a literal is called its arity. Terms can be constants which are written in lower case or variables with capitalized words. The complete Horn clause is called a rule while without a body it is called a fact. An example of a Datalog program is as follows.

1. sister(elizabeth, mary).
2. brother(mary, john).
3. sibling(*X*, *Y*) :- sister(*X*, *Y*) ; brother(*X*, *Y*).

Datalog statements 1 and 2 are the facts which show that sister of Elizabeth is Mary and brother of Mary is John. Here, *sister* and *brother* are predicate symbols with arity two. Statement 3 is the rule with ***head*** as *sibling* predicate symbol containing variables *X* and *Y*, and ***body*** containing literals sister and brother. The rule says that *X* is the sibling of *Y* if *X* is the sister of *Y* or *X* is the brother of *Y*. The advantages of using Datalog are six fold:

#### 2.5.1. Simplicity

As we can see from the examples given above that it has a simple structure. It is due to the declarative nature of Datalog that the underlying syntax is intuitive and simple.

#### 2.5.2. Expressiveness

Datalog can capture information that is indefinite in nature and provides more expressive power than most of the relational databases. It is naturally capable of representing non-first-normal-form relations and allows recursion. Apart from that its semantics allow for better reasoning enhancing its expressivity and making it amenable to rich analysis. Another way of looking at is as a special kind of automated theorem prover that allows special kinds of formulae [34].

#### 2.5.3. Uniformity

Another feature of Datalog is its uniformity in its representational and operational form. The database of IDBs and EDBs can be queried in positive, negative or as constraint queries and they are all expressed with a homogenous formalism.

#### 2.5.4. Mass storage facility

Datalog is the extension of logic programming that has been integrated with database management. As a result of this it combines the benefits of declarative querying and efficient and reliable mass storage facility. This makes it a strong candidate for storing simulation results.

#### 2.5.5. Easy and fast retrieval

With a mass storage facility at its disposal, it is equipped with a bottom up evaluation method. The bottom up approach works by considering facts and applying them to rules. The resultant facts that are derived in this process are applied again and again until there are no more facts remaining to be derivable. The approach ascertains that the queries terminate. This possibility is not certain for any arbitrary programs and is a major issue in databases. Datalog guarantees that the queries terminate in polynomial time. The function-symbol-free property of Datalog ensures its tractability.

#### 2.5.6. Fix points semantics

Datalog is based on the fixpoint semantics. They deal with the minimal Herbrand model which means that relationships are limited to the rules and facts are those that are either explicitly mentioned or are derived by these rules. According to model theoretics given an instance I (consisting of only EDB predicates), for any Datalog program P a model consists of the entire schema i.e. the EDB predicates and IDB predicates. Datalog uses a minimal model so that such a model does not have a subset which is also a model. Queries are evaluated against such a model. The implication of such a model means that using a simulator the model can be expanded as desired.

## 3. Related work

Many publications have addressed the simulation of requirements. Use cases have been transformed into Petri-net using a recursive transformation algorithm in [35]. The simulated scenario instances are stored in a dedicated trace model as a result of which different kinds of static analyses can be performed. The Play engine tool in [36,37] allows the user to execute the behavior requirements written in Live Sequence Charts. The framework Monterey Phoenix has been presented for the development of system architecture in [38]. It allows for the architectural models to be executable on the abstract machine so that early verification of the top level system design can be achieved. Bus protocol specifications are used for simulation to automatically generate the verification aids in [39]. Counterexamples of the Simulink models have been simulated to visually interpret them, which not only accelerates the process of requirements error detection but also discover misinterpreted requirements [40]. A unified environment SQUANDER has been proposed in [41] that can execute specifications written in the JForge specification language (JFSL) along with imperative code. However these endeavors make use of input specification notations that do not address the informal-formal gap during modeling.

On the other hand, Behavior Trees (BTs) have been translated to provide model-checking capabilities. Translation into SAL in [10] provides theorem proving and model-checking capabilities. The automated analysis can be performed using Linear Temporal Logic (LTL) by a tool support. It improves the verification time by reducing the parallelism in the model. Translation of BTs into Communicating Sequential Processes (CSP) allows for automated analysis using the Failure-Divergence Refinement (FDR) model checker [42]. Automated support to perform Failure Mode and Effect Analysis FMEA is also present [30]. The analysis ascertains what type of hazard can occur in case of component failure. Similarly to add a time dimension in FMEA, automated analysis using the timed model checker UPPAAL has also been incorporated [26]. Probabilistic FMEA can be performed on BTs using the PRISM model checker [43]. The translation of BTs into model checkers provides automated analysis support. However the state-explosion problem remains a major drawback. To reduce the state-space during model-checking a Behavior Trees slicing strategy has been

proposed in [44]. Slicing in the context of formal analysis refers to removing those parts from the model under consideration that have no effect on the satisfaction of the property being checked. Thus, the technique is dependent on the property being checked for developing a judicious *slicing criterion*. This slicing criterion is derived directly from the property often stated as LTL formula.

There exist other simulation frameworks and tools for BTs in the literature. For instance theoretical basis for a BTs simulator called BTsim has been proposed in [28]. The proposed simulator can be used to generate random traces interactively or automatically, and to exhaustively generate all traces. Theoretically, the simulator can be used to execute BTs requirements. The simulation can be useful for visualization to step through each node during the execution. It can also be used to check for certain safety properties in the BTs model. However, the simple visual execution of BTs is of limited advantage in the formal analysis. More recently, execution of BT requirements using State Machines (SM) that are synchronized with the BT model has been proposed using the Text-BE tool [27]. The objective of transformation of BTs into SM models is to support MDE using UML. The strategy can also be useful in removing defects from the BT model by analyzing the dynamic aspects of the specification. The transformation of BTs into SM is based on M2M transformation using the BT meta-model defined in the Eclipse Modeling Framework (EMF) and the standard Object Management Group (OMG) Unified Modeling Language (UML) superstructure [27]. The resultant SM models the behavior originally specified in the BT model. The SM tool allows one to extend the range of software analysis and development such as including automated generation of test cases. Our approach is different from the previous ones in two ways. Firstly we integrate execution, step by step visualization during each execution step, user interaction and execution of a sub-tree in a single standalone simulation tool. Secondly the model (the Datalog facts and rules) expansion capability of our simulator allows for increasing the state-space of the system under consideration at will. Analysis can be carried out on the resultant state space by the help of Datalog queries which prove the properties of interests. Traces can be generated in the case of failed properties. The homogeneous formalism of Datalog is valuable in forming the queries and for understanding the traces generated by the Datalog over LTL formulae used in model checking.

## 4. Transformation

This section describes the transformation process using model transformation [17] in detail. One of the advantages of the model transformation is the deterministic output. The translation process comprises of a number of steps as shown in Fig. 6. It involves M2M transformation as step one – given a source model as input, it automatically generates a target model as an output. The transformation rules provide the automatic mapping between the source metamodels and the target metamodels to which the source models and the target models conforms to respectively. We have used ATL to write the mapping between the TextBT; the source metamodel (Fig. 7) and SimTree; the target metamodel (Fig. 8). In step two, M2T transformation is implemented using Java Emitter Templates (JET). Given a model, M2T transformation is responsible for printing the code – in our case the model conforms to the SimTree metamodel. The generated code is part of the simulator SimTree. In step three SimTree is executed which emulates the execution semantics of Behavior Trees (BTs), capturing them into Datalog rules and facts. Analysis can be carried out using Datalog queries on the captured Datalog rules and facts.

### 4.1. Translation rules

The first step in the transformation is to define a mapping between the BT and SimTree model elements i.e. M2M transformation. This mapping is described by translation rules written in ATL. The input to the ATL is the BT model and the execution of ATL gives the SimTree model as an output. The translation consists of the following rules; (1) entry point called rule, (2) one matched rule, (3) five lazy rules and (4) four called rules. The *entry point* rule is so called because it is executed between the initialization and the matching phases in the translation. *Matched rules* are applied to match a given set of elements using a standard rule. *Lazy rules* are typically referred from other rules and may be applied many

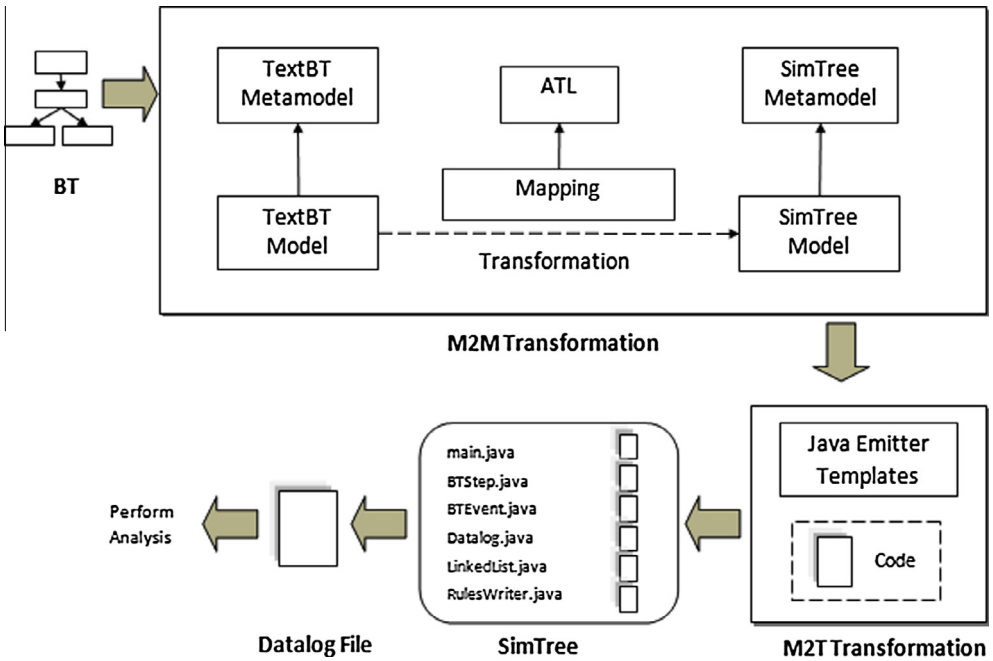



**Fig. 6.** Translation process.

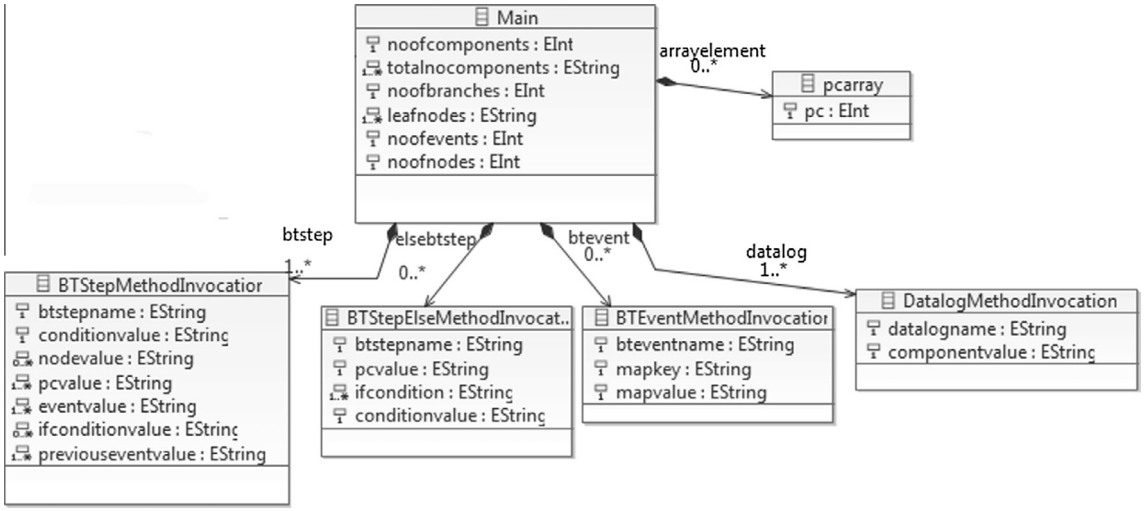



**Fig. 7.** Target metamodel.

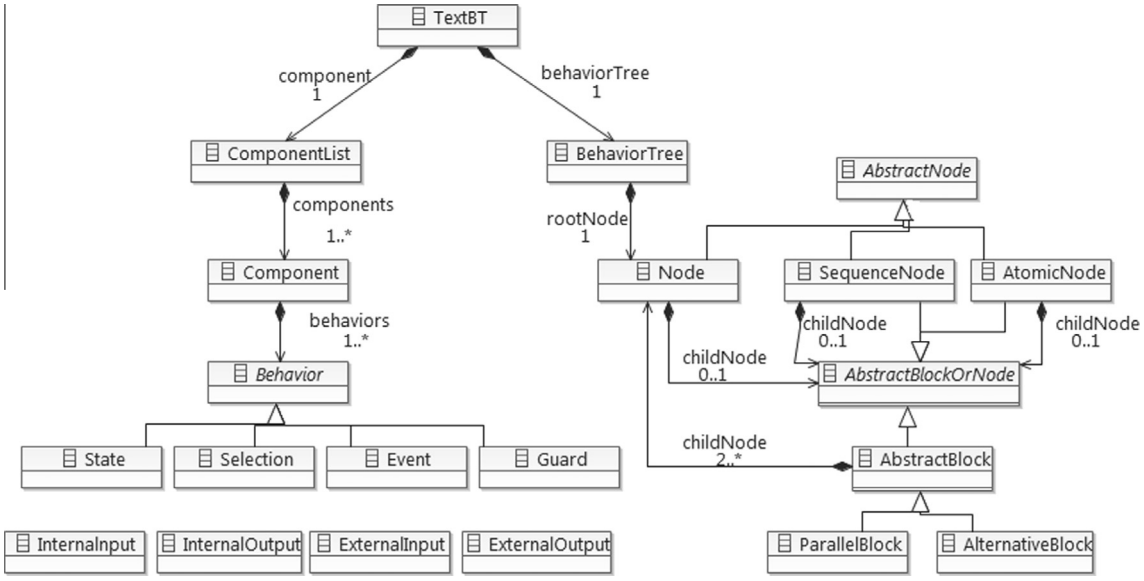



**Fig. 8.** Source metamodel.

times for each match. In addition, ATL allows for those rules that enable to explicitly generate target model elements directly from code. These rules are referred to as *called rules*. The matched rules and the five lazy rules together make up the heart of our transformation. The matched rules and called rules are used for generating the model elements of the SimTree metamodel. Our model also makes use of a number of *helpers* and *attribute-helpers* to assign values to different properties of the SimTree metamodel. Table 2 summarizes the ATL rules used for transformation of the TextBT metamodel into the SimTree metamodel. A brief description of the rules is provided in the following subsections.

#### 4.1.1. Entry point called rule

The entry point rule is implicitly invoked at the beginning of the transformation execution, before the matched rules are executed. In our translation the entry point called rules is main() – which is responsible for assigning program counter and its value (in depth first manner) to each *AbstractNode* element (except for *AtomicNode*) of the TextBT metamodel. The program counters are used to keep track of the sequence of the BT nodes during execution. They preserve the sequential, atomic, parallel and alternative control flows of the nodes. They are also responsible for the execution semantics by checking and setting their values in *lazynode2btstep* lazy and *getpcValuerule* called rules respectively. A Map data structure is used to store each node's program counter and its value – i.e. pcMap and pcCounter respectively.

#### 4.1.2. Matched rule

The matched rule provides a mean by which to specify the way SimTree target model elements are generated from TextBT source model elements. Our translation makes use of the *component2main* matched rule and maps the *ComponentList* model element of TextBT with the *Main* model element of the SimTree metamodel as shown in Fig. 9. The matched rule in turn calls a number of *attribute helpers* which assign values to the properties of the SimTree metamodel. These properties hold the information about the BT's total number of branches, leaf nodes and total number of components and are used as initializations in the simulator SimTree.

**Table 2**
ATL rules.

| Types of rules | Name of rules | Description |
|---|---|---|
| Entry point rule | Main | Assigns program counter and their values to the individual nodes of the BT in a depth first manner. Atomic nodes are exempted |
| Matched Rule | Component2main | Generates the target metamodel element – *main* with the help of attribute helpers<br>The attribute helpers<br>• Count the no of nodes of state type behavior, event type behavior and the branches of the BT<br>• Extract the node's related component and behavior if it is a state realization node and program counters and their values of all the leaf nodes<br>Places Calls to the lazy rules |
| Lazy Rules | Lazynode2arrayelement | Generates the target metamodel element – *pcarray* |
| | lazyevent2btevent | Generates the target metamodel element – *BTEventMethodInvocation* by making use of a helper function which<br>• Returns the component name and behavior of the Event type node |
| | lazystate2Datalog | Generates the target metamodel element – *DatalogMethodInvocation* by making use of a helper function which<br>• Returns the component name and behavior of the State realization type node |
| | selectionNode2elsebtstep | Generates the target metamodel element – *BTStepElseMethodInvocation* by help of a couple of helper functions which<br>• Returns the component name and behavior of the Selection type node<br>• Returns the node's associated program counter and sets it to zero |
| | lazynode2btstep | Generates the target metamodel element – *BTStepMethodInvocation by*<br>• Using a helper function which returns the node's associated program counter and its value<br>• Placing Calls to the Called rules |
| Called rules | getnodeValuerule | Returns the sequence of strings containing the node's component and behavior names. All other nodes except those having behavior of State type are filtered out |
| | geteventValuerule | Returns the sequence of strings containing the node's component and behavior names. All other nodes except those having behavior of Event type are filtered out |
| | getifconditionValuerule | Returns the sequence of strings containing the node's component and behavior names. All other nodes except those having behavior of Selection type are filtered out |
| | getpcValuerule | Returns the sequence of strings containing the program counters of specific nodes and sets their values according to the execution semantics of BTs |

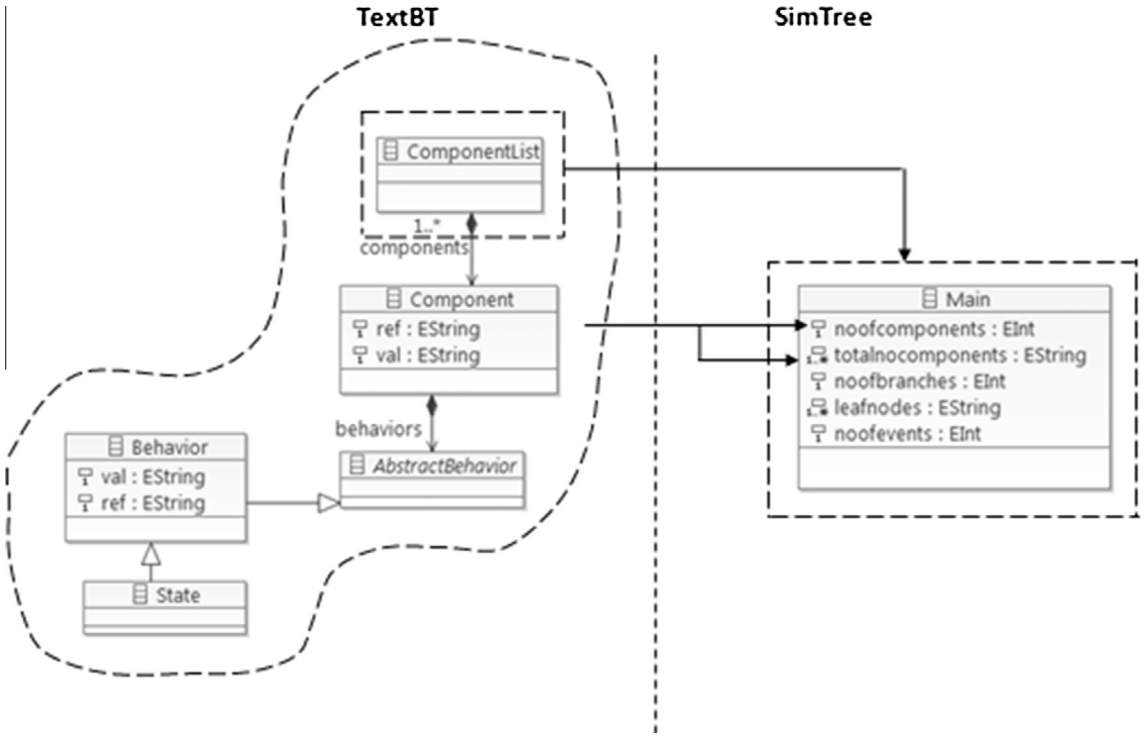



**Fig. 9.** Mapping of metamodel elements ComponentList → Main.

The matched rules also place calls to the five lazy rules which in turn generate other target metamodel elements as discussed below.

*4.1.3. Lazy rules*

The matched rule invokes calls to the five lazy rules. These lazy rules initialize the *BTEventMethodInvocation, DatalogMethodInvocation, pcarray*, *BTStepMethodInvocation* and *BTStepElseMethodInvocation* target model elements of the SimTree metamodel (Figs. 10–14). The number of elements of the target model produced by the lazy rules, is equal to the number of elements of the source model. The lazy rule *Lazynode2arrayelement* maps the *pcarray* element of target SimTree from each *Node* element of the TextBT. The *rootnode* element is exempted. A mapping strategy is used to

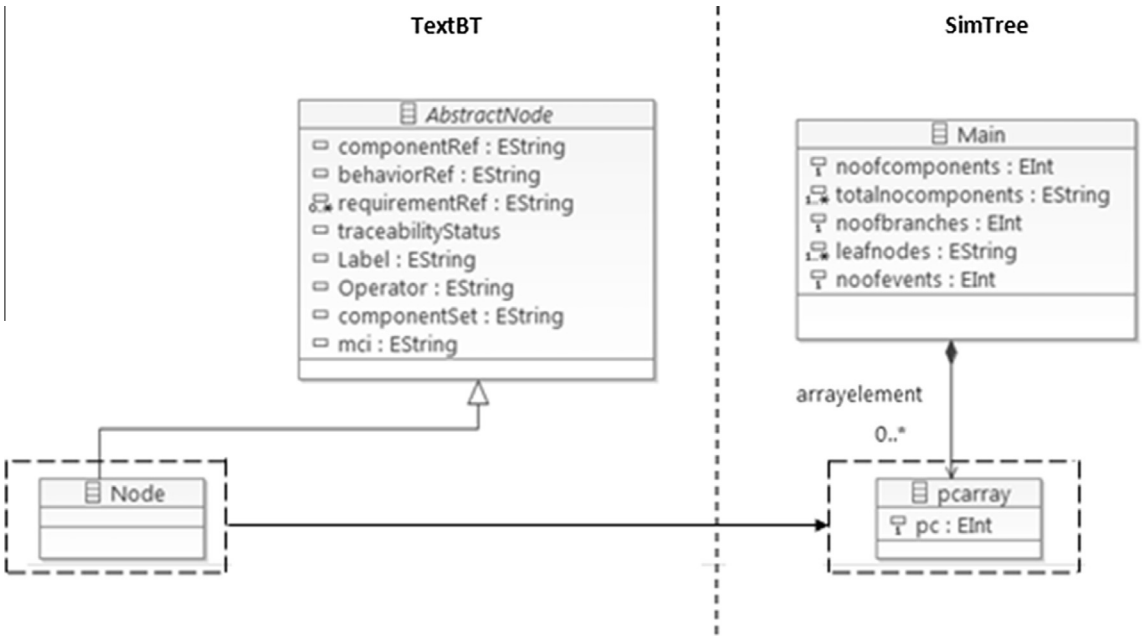



**Fig. 10.** Mapping of metamodel elements node → pcarray.

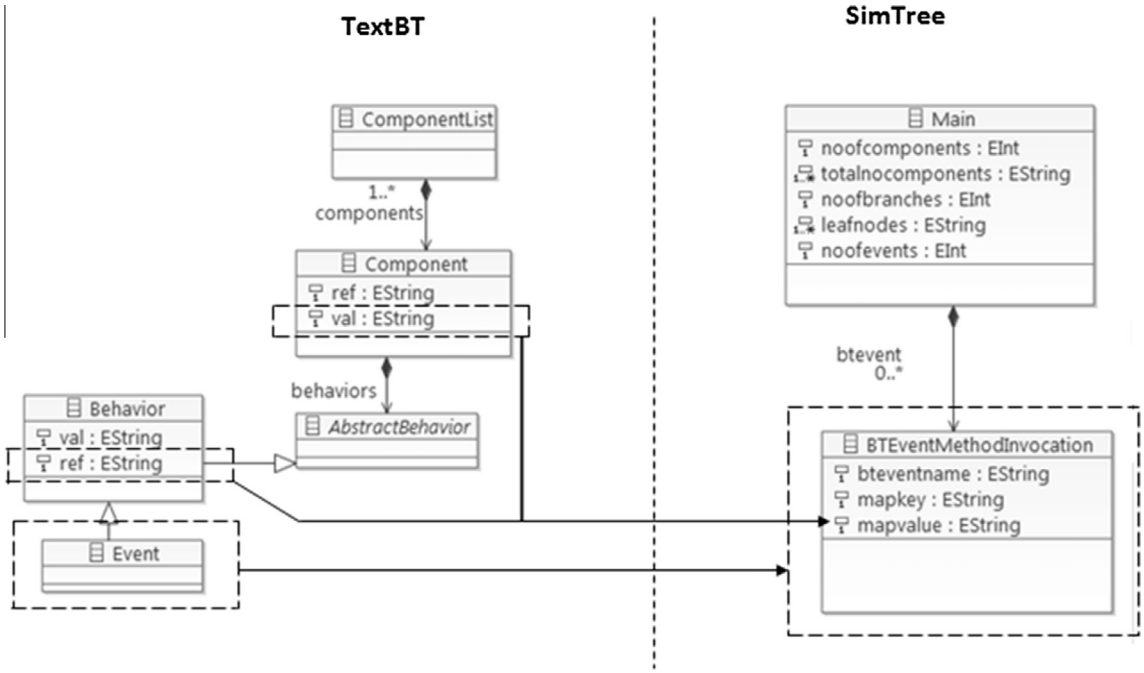



**Fig. 11.** Mapping of metamodel elements event → BTEventMethodInvocation.

initialize the program counter associated with each node in the BTs. Initially, all the program counters are set to zero except the first one to preserve the initial states of components in a given BT. This is achieved by initializing the pc property of the pcarray metamodel element.

*Lazyevent2btevent* lazy rule maps the *BTEventMethodInvocation* element of the SimTree metamodel from each *Event* element of TextBT. The purpose is to filter out the *event* nodes and extract their component and behavior names by initializing the properties of the SimTree metamodel element. These *event* nodes are then treated according to the execution semantics of the Behavior Trees (BTs) in the simulator SimTree. Similarly *Lazystate2Datalog* lazy rule maps the *DatalogMethodInvocation* elements from each *State* element. The nodes of *state* type are filtered and their component and behavior names are extracted via helper functions. The purpose is to preserve the state changes of the components into Datalog rules and facts during the execution of a given BT in SimTree.

Lazy rule *selectionNode2elsebtstep* maps the *BTStepElseMethodInvocation* element of the SimTree metamodel from each *AbstractNode* element of the TextBT which has the associated *Component Behavior* element of *Selection* type. All other *AbstractNode* elements with different associated *component behavior* elements are filtered out.

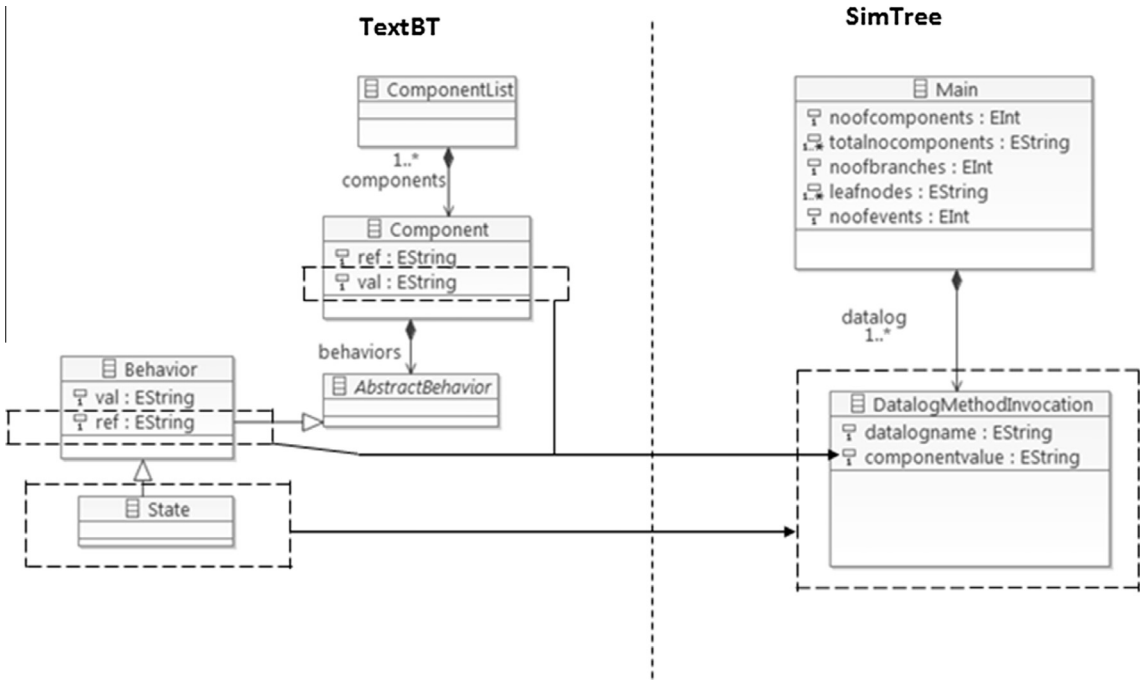



**Fig. 12.** Mapping of metamodel elements state → DatalogMethodInvocation.

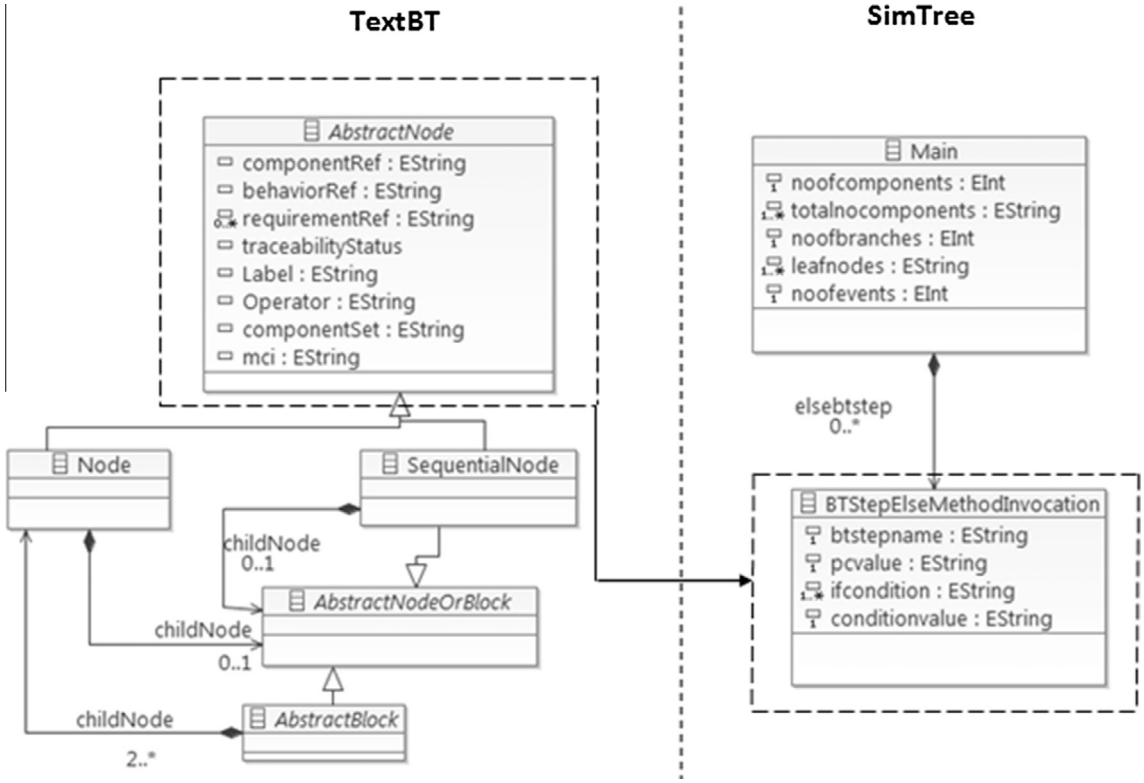



**Fig. 13.** Mapping of metamodel elements AbstractNode → BTStepElseMethodInvocation.

One helper function extracts the component and behavior names of the nodes and initializes the *ifcondition* and *conditionvalue* properties. The second helper function extracts the associated program counter and set its value to zero and initializes the *pcvalue* property. For each node a unique btstep integer ID is given.

The fifth lazy rule *Lazynode2btstep* maps the *BTStepMethodInvocation* element of the target SimTree metamodel from each *AbstractNode* element of the TextBT metamodel. However the *AtomicNode* elements are filtered. This mapping is responsible for translating each BT node to its equivalent thread in the SimTree in accordance with the execution semantics. It is achieved by giving a unique name to each BT node, obtaining each BT node's program counter and its value, setting the required program counters' values and extracting *event* and *state* behavior accordingly.

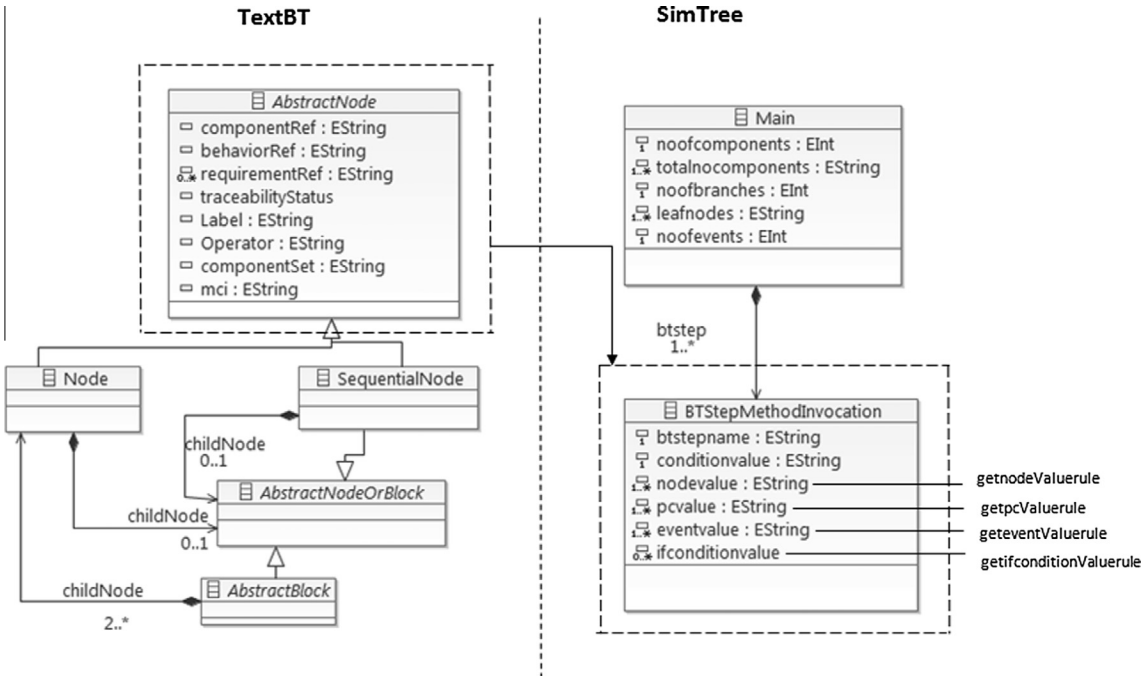



**Fig. 14.** Mapping of metamodel elements AbstractNode → BTStepMethodInvocation.

The details of this are deferred till the next Section 4.1.4 where we describe the called rules responsible for initializing the properties of the said element. The other properties of the *BTStepMethodInvocation* element are initialized as follows: The *btstepname* property for each node is initialized with the unique btstep integer ID. Each *conditionvalue* property is initialized by the nodes program counter and its associated value. The *nodevalue, eventvalue, ifconditionvalue* and *pcvalue* properties are initialized by four called rules which are described in the next subsection.

#### 4.1.4. Called rules

Called rules are responsible for populating the mapping of each BT node through the Lazynode2btstep lazy rule described in the previous Section 4.1.3. The four called rules are (1) *getnodeValuerule*, (2) *geteventValuerule*, (3) *getifconditionValuerule* and (4) *getpcValuerule*. Called rule *getnodeValuerule* takes as parameter each AbstractNode element and returns the component and behavior name associated with each BT node. If the current BT node has atomically composed child-nodes, then it recursively extracts the component and behavior names of these node. Please note that only those BT nodes are considered which have behavior of state type. Similarly the *geteventValuerule* and *getifconditionValuerule* called rules return component and behavior names in the same fashion as the previous called rule. However the former considers BT nodes having behavior as event type while the latter considers those of *selection* type. The *getpcValuerule* called rule is responsible for setting the program counters of each BT node and captures the semantics of sequential and parallel flow. It takes in an *AbstractNode* (excluding *AtomicNode* elements) as parameter and returns a sequence of program counters and their values. To capture the semantics of sequential control flow the sequentially composed BT node's program counter value is incremented as shown in Fig. 15(a). On the other hand to capture the semantics of parallel and alternative control flow, it appends the parallel/alternative composed BT nodes' program counters and values in the sequence (Fig. 15(b)). In the case of alternative control flow (Fig. 15(c)) it additionally sets the program counters of the sibling BT nodes to zero and appends it to the sequence along with the program counters. For the BT node which is a leaf node having a reversion (^) operator, the matching node is found in ancestor nodes. The program counter value of the matching node is incremented and is appended to the sequence with its associated program counter. The matching node's child-nodes are checked and program counter values are further set in the following ways.

1. If the child-node is composed sequentially (Fig. 15(d)) then all its ancestor sub-tree nodes' program counters are set to zero and are appended to the sequence along with the associated program counters.
2. If the child-nodes are composed in a parallel or alternative composition (Fig. 15(e)) then all the ancestor sub-tree nodes program counters are set to zero except for parallel/alternatively composed child-nodes. The child-nodes program counters and their values are appended into the sequence.
3. If the child-node is composed atomically, they are skipped until sequential or parallel/alternative composed nodes are encountered (Fig. 15(f)). In the case of sequential composition the program counters are set as in point 1 while in the case of parallel/alternative composition the program counters are set according to point 2.

The translation rules implemented in ATL on execution gives us the SimTree model at the end of step one. The translation is done in a way that the execution semantics of BTs are preserved. Given a SimTree model, the next step in the transformation process is the printing of the code using M2T transformation. We describe the process in detail in the next subsection.

### 4.2. Code generation

In the second step, code generation is carried out using M2T transformation. We use Java Emitter Templates (JET) in carrying out the printing of the code. The SimTree model which is the output of our M2M transformation becomes the input to this step. JET uses templates, XPath expressions and model handlers to navigate the input model and generates the desired code. In our M2T

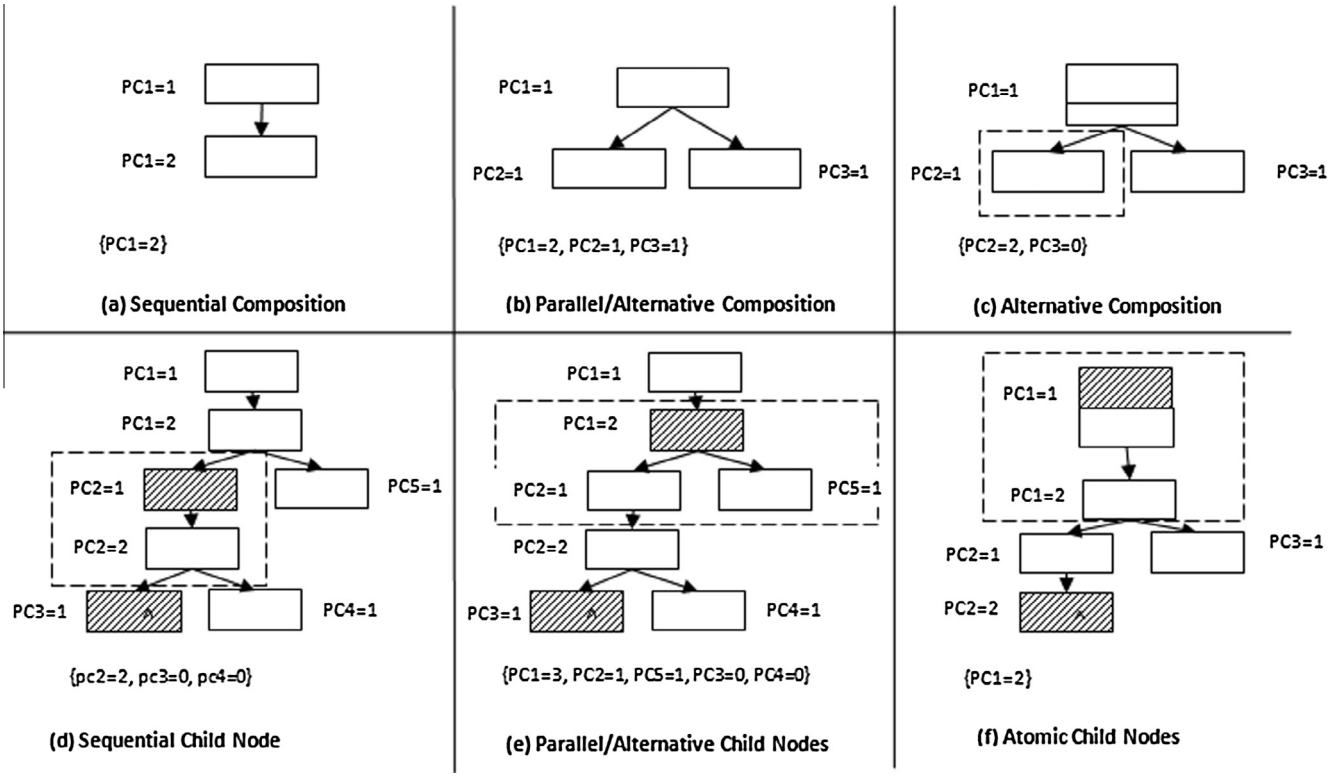



**Fig. 15.** Called rule getpcValuerule.

transformation the output is the java code which we save in main.java file. The automatically generated main.java file then becomes the part of the simulator SimTree in the third step. The simulator consists of six java classes, namely main.java, BTStep.java, BTEvent.java, Datalog.java, RulesWriter.java and the LinkedList.java. Below are the steps involved in the translation of a SimTree model into java code.

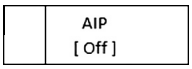
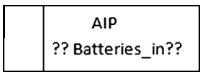
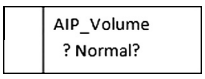
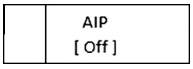
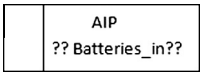
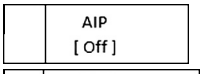
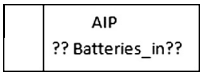
1. Each BT node except for the one which is atomically composed becomes the instance of a BTStep class.
   a. The BT node having behavior as *state* invokes the addData() method of the LinkedList class "node". The parameters passed to addData() method are the component and behavior name of the node.
   b. The BT node having behavior as *event* invokes the addData() method of the LinkedList class "events". The parameters passed to the method are component and behavior name of the node.
   c. The BT node having behavior as *selection* invokes the addData() method of the LinkedList class "ifconditions". The parameters passed to the method are the component and behavior name of the node.
   d. The BT node invokes the addData() method of the LinkedList class "conditions". The parameters passed to the addData() method is the program counter and its value associated with the node.
   e. The BT node invokes many addData() methods of the LinkedList class "pc". The parameters passed to each of these addData() method are the program counter and its value as set by the ATL transformation. These program counter values preserve the execution semantics of the BTs.
2. In the case where the BT node has behavior as *selection*, it becomes an additional instance of the BTStep class that invokes the addData() method of LinkedList classes "ifconditions", conditions" and "pc in the same spirit as point 1. It also invokes the addData() method of LinkedList class "events" with 'nil' as a parameter.
3. Each BT node which is a leaf node invokes the addData() method of the LinkedList class "leafnodes". The parameters passed to the method are the program counter and its value associated with the BT node.
4. There is one instance of the BTEvent class. Every BT node having behavior as an *event* invokes the put() method of the map data structure. The parameters passed to the put() method are the node's component and behavior name and an empty string.
5. There is one instance of the Datalog class. Every BT node having behavior as *state* invokes addData() methods of two LinkedList classes namely "component" and "totalnoofcomponents". The parameters passed to the "component" LinkedList's methods are the BT node's component and behavior name. The parameter passed to the "totalnoofcomponents" LinkedList's method is the BT node's component name only.

We show the generated main.java code by the help of each node example in Table 3 followed by the details of the SimTree in the Section 4.3.

### 4.3. Simulator SimTree

The third step of the transformation involves the Simulator SimTree. SimTree is responsible for the generation of a Datalog database. The database consists of rules and facts that are asserted upon the execution of SimTree. The Datalog rules and facts are asserted in a text file. The printed code from the M2T transformation - main.java, in the second step becomes the input to the simulator. At the moment SimTree is configured to generate Datalog rules and facts at each step during the execution of a given BT. The main.java class is part of the project along with other supporting classes, namely BTEvent.java, BTStep.java, DataLog.java,

**Table 3**
Main.java code.

| Node type | Node | BTStep | Description |
|---|---|---|---|
| State |  | btstep1.conditions.addData("pc1==1")<br>btstep1.node.addData("aip = aip_off")<br>btstep1.pc.addData("pc1 = 2")<br>btstep1.events.addData("nill") | The node's component is AIP and the behavior is Off. Also note that in the events LinkedList addData() method, nill is passed as an argument |
| Event |  | btstep2.conditions.addData("pc1==2)<br>btstep2.pc.addData("pc1 = 3")<br>btstep2.pc.addData("pc2 = 1")<br>btstep2.pc.addData("pc3 = 1")<br>btstep2.events.addData("aipbatteries_in") | The node's component is AIP and the behavior is Batteries_in |
| Selection |  | btstep45.conditions.addData("pc17==1")<br>btstep45.pc.addData("pc17 = 2")<br>btstep45.events.addData("nill")<br>btstep45.ifcondition.addData("aip_volume = normal)<br>btstep67.conditions.addData("pc17==1")<br>btstep67.pc.addData("pc17 = 0")<br>btstep67.events.addData("nill")<br>btstep67.ifcondition.addData("NOTaip_volume = normal) | The node's component is AIP_Volume and the behavior is Normal |
| **Node type** | **Node** | **Datalog** | **Description** |
| State |  | dl.component.addData("aip = aip_off")<br>dl.totalnocomponent.addData("aip") | The node's component is AIP while its behavior is Off |
| Event |  | N/A | |
| **Node type** | **Node** | **Event** | **Description** |
| State |  | N/A | |
| Event |  | btevent.map.put("aipbatteries_in,") | The node's component is AIP and its behavior is Batteries_in |

LinkedList.java and RulesWriter.java. These supporting classes are generic and are used to capture the semantics of a BT. With the help of these classes, Datalog rules and facts are generated. User can set the number of runs in order to control the simulation length. The following (Sections 4.3.1 and 4.3.2) show how BT nodes map to Datalog rules and facts.

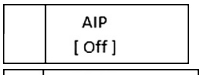
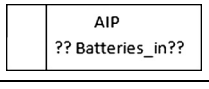
### 4.3.1. Datalog facts

The output of the SimTree is the Datalog file. The BT nodes which are of State type are asserted as facts in the file. The node's component name becomes the predicate symbol while its behavior becomes the term. These facts are asserted only once and become the header of our Datalog file. Every time a node is executed in the SimTree (as an instance of the BTStep class) more facts and rules are asserted. At the execution of a BT node of event type, a fact is asserted with the predicate symbol as "events". The arity of this fact is equal to the total number of BT nodes which are of event type in a given BT. For instance, if there are two event BT nodes then the arity of "events" fact is two. The terms of the "events" predicate symbol are the Boolean values. These values represent the event type BT nodes captured during that instance of the execution. The order of the terms differentiates the different event type BT nodes. In Table 4 we describe the generation of Datalog rules and facts with the help of the same example that we used in Table 3.

### 4.3.2. Datalog rules

A Datalog rule is asserted with the predicate symbol as "system". The arity of the "system" predicate is two more than the total number of state type BT nodes. The position of the terms distinguishes one BT node component from the other. The first term is reserved for the step number while the last term is reserved for any event occurring during the execution. The head of the rule is the "system" predicate capturing the current executing BT node while the body is the conjunction (represented by commas) of the previously executed BT node as "system" literal and other literals. The other literals depend upon what type of BT node is being executed. If the executing BT node is of type event then the literal is "events", otherwise it is the literal capturing the currently executing BT node. In the latter case the literal's predicate symbol is the component of the node while the term is the behavior of the node.

**Table 4**
Executed Datalog rules and facts.

| Node type | Node | Datalog facts | Description |
|---|---|---|---|
| State |  | aip(off) | The predicate symbol of the fact is aip which is the node's component and its term is off which is the node's behavior |
| Event |  | events(true) | The node is an event type so the Datalog fact consists of predicate symbol as events with arity one (assuming there is only one node of event type) |

Once the execution of the SimTree is complete and the generated Datalog rules and facts are stored in the file, we can carry out analysis. The Datalog file can be used via Datalog queries for this purpose. The polynomial termination time of the queries guarantees an efficient analysis. We apply our transformation on Ambulatory Infusion Pump (AIP) case study in Section 5 and discuss the results of our analysis.

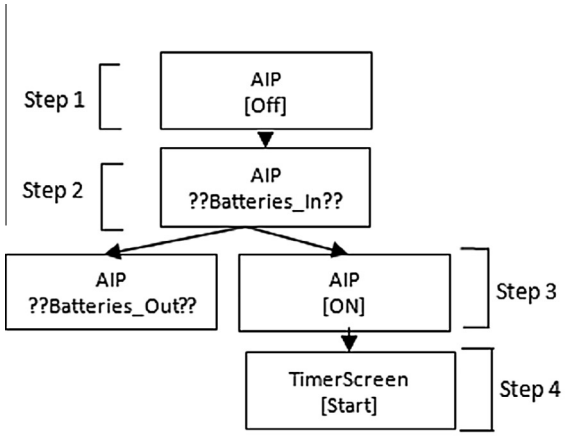
## 5. Proof of concept

As a proof of concept we apply the transformation and carry out analysis on a published case study called Ambulatory Infusion Pump (AIP). Its critical requirements are derived from [25]. The AIP BT is modeled in the TextBE textual editor and is then fed into the ATL transformation. The transformation outputs the SimTree model for AIP. The JET Transformation then generates the equivalent main.java file (which becomes part of SimTree). The Simulator SimTree is executed and the Datalog rules and facts are generated. Execution of each node results in a rule while facts are asserted in the beginning as a header. The rules and facts are stored in a textual file. Figs. 16 and 17 shows a sample sub-tree of AIP BT and its equivalent java and Datalog code. There are some critical safety concerns with the AIP device. For instance, air embolism, drug overdose and drug under delivery can lead to illness or even the death of the patient. We discuss these safety concerns as safety properties in detail and show the results of our analysis performed via Datalog queries in the following subsections.

### 5.1. Safety properties

A serious safety concern in the AIP is the presence of air in the line that can cause a serious medical condition called air embolism. The drug under-delivery can be caused by blockage in the line and similarly the mismatch in the amount of drug being infused and the drug being calculated can cause drug overdose. In order for the pump to work properly these safety concerns should be addressed. The pump should stop pumping the dose if there is blockage in the line or presence of air is detected in the line. The pump should also stop pumping immediately if no more medicine is left in the AIP. These safety issues are addressed as safety properties and they must hold for the proper functioning of the device. In order to perform the analysis, we need equivalent Datalog queries. Datalog provides both positive and negative queries and user integrity constraints. The former provides a mean to check whether the desired tuple is deduced via firing of Datalog rules and facts, while the latter can be used to query the database for any tuple of infeasible values.

**Safety Property 1 –** The First safety property states that it should always hold that whenever the occlusion sensor detects




```
.....INITIALIZATION
dl.component.addData("aip=aip_off");.....
btevent.map.put("aipbatteries_in","");.....
```

**Step 1**

```
btstep1.conditions.addData("pc1=1");
btstep1.node.addData("aip=aip_off");
btstep1.pc.addData("pc1=2");
btstep1.events.addData("nill");
btsteo1.start();
```

**Step 2**

```
btstep2.conditions.addData("pc1=2");
btstep2.pc.addData("pc1=3");
btstep2.pc.addData("pc2=1");
btstep2.pc.addData("pc3=1");
btstep2.events.addData("aipbatteries_in");
btstep2.start();
```

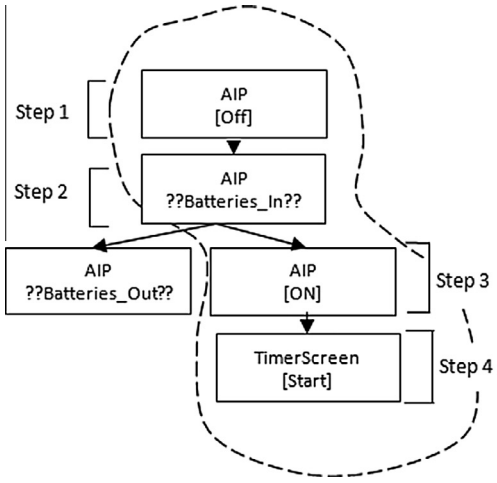
**Step 3**

```
btstep3.conditions.addData("pc3=1");
btstep3.node.addData("aip=aip_on");b
tstep3.pc.addData("pc3=2");
btstep3.events.addData ("nill");
btstep3.start();
```

**Step 4**

```
btstep4.conditions.addData("pc3=2");
btstep4.node.addData("timerscreen=ti
merscreen_start");
btstep4.pc.addData("pc3=3");
btstep4.events.addData("nill");
btstep4.start();
```

**Fig. 16.** Sample BT and its equivalent java code.




```
.....INITIALIZATION
system(0,ini,ini,.....,nill).
aip(on).
aip(off).
timerscreen(start).
```

**Step 1** `system(1,off,ini,nill):- system(0,ini,ini,nill), aip(off).`

**Step 2** `system(2,off,ini,batteriesin):- system(1,off,ini,nill),events(true, false).`

**Step 3** `system(3,on,ini,nill):- system(2,off,ini,batteriesin),aip(on).`

**Step 4** `system(4,on,start,nill):- system(3,on,ini,nill), timerscreen(start).`

**Fig. 17.** Sample BT and its equivalent datalog rules and facts.

blockage, the pump should not be pumping. We make use of the Datalog constraint :- ***A, B*** which states that when ***B*** is true, ***A*** should not be true. Using the constraints the safety property is translated into a Datalog Query (DQ1). It states that no such tuple should be computed where when occulusion_sensorline is blocked and AIP = aip_pump_running.

DQ1 :- system(A,B,C,D,E,AIP,F,G,H,I,J,occulusionsensorline_blocked),AIP = aip_pump_running

**Safety Property 2 –** The Second safety property states that it should always hold that whenever the air sensor detects air, the pump should not be pumping. We translate it into Datalog constraint (DQ2). It states that no such tuple be computed where the air sensor has detected air and AIP = aip_pump_running.

DQ2 :- system(A,B,C,D,E,AIP,F,G,H,I,J,air_sensorair_detected), AIP = aip_pump_running

**Safety Property 3 –** The Third safety property states that it should always hold that whenever the volume reaches zero, the pump must immediately be stopped. We translate it into Datalog constraint (DQ3). The query ascertains that it is always true that when aip_volume _empty it is also true that AIP = aip_pump_stopped.

DQ3 :- system(A,aip_volume_empty,B,C,D,E,AIP,F,G,H,I,J), not(AIP = aip_pump_stopped)

Table 5 shows the safety properties and their equivalent Datalog queries along with LTL formulae. The generated Datalog file of the AIP BT is consulted in Datalog Educational System (DES) software and analysis is performed by firing each Datalog query. The results of these queries show us whether the properties hold or not. Violation of the constraint results in safety properties being violated.

We performed the analysis by firing the Datalog queries in DES. We fired the equivalent Datalog queries of safety properties 1 and 2, both of which resulted in a violation of the constraint. DES output the violated values as a new Datalog tuple that caused the violation of the safety properties 1 and 2. By generating the traces with this new Datalog tuple we closely examined the cause of the violations in the AIP BT. There are four parallel sub-branches S1, S2, S3 and S4 where the interleaving is happening (as shown in the Fig. 18). Sub-branch S1 is where the Occlusion_Sensor senses the line to be blocked followed by an atomic composition of multiple nodes in which one of the nodes has AIP_Pump [Stopped]. However, running in parallel (with interleaving) is branch S4. S4 further contains parallel sub-branches S4.1 and S4.2. Sub-branch S4.2 has a sequential node with AIP_Pump[Running]. The sequence of steps that violates the safety property is sub-branch S1 node with Occlusion_Sensor ??Line_Blocked??, followed by sub-branch S4 node with SS_Button??Held??, followed by interleaving of nodes in sub-branches S3, S2 and S4, followed by sub-branch S4.2 with node having AIP_Pump[Running]. This means that when the Occlusion_Sensor has line_Blocked then through the sequence of interleaving steps in sub-branches S2, S3 and S4 the AIP_Pump is in running state. Similarly the firing of DQ2 reveals the same problem i.e. interleaving of nodes in the sub branches S2, S3, S4 and S4.2. This results in AIP_Pump in the running state even though air is detected by the air sensor.

In order to rectify the incorrect modeling of AIP BT we make the composition of all the nodes in sub-branch S1 and S2 as atomic. This makes the sub-branch behave as one node and interleaving of nodes from sub-branches S3, S4, and S4.2 is stopped. Here as soon as the Occlusion_Sensor senses that the line is blocked it immediately stops the AIP_Pump. We made the corrections in our original AIP BT and carried out the analysis again. The protection of this critical region by atomic composition in the BT ensured that the safety properties were not violated in the next run. The corrected portion of the AIP BT is shown in the Fig. 19.

### 5.2. Liveness properties

Datalog queries also help us to ascertain that something good must happen i.e. the liveness properties. A liveness property allows us to know that given an execution of a BT a certain state is reached. We make use of Datalog positive queries to know whether the deeper sub-branches of the AIP BT are reached or not. Here we check that is there a state where AIP_Volume becomes empty, as a Datalog positive query LQ1.

#### 5.2.1. LQ1: System(A,aip_volume_empty,B,C,D,E,F,G,H,I,J,K)

LQ1 is the positive query that states whether there is any tuple that is computed where AIP_Volume is empty. On firing LQ1 in DES we find that no such tuple is computed. This only shows that no such state is reached in the AIP BT where the AIP_Volume is being set to empty. This leads us to the fact that the whole AIP BT has not been executed resulting in the reachability problem. We let the simulator run for twelve hours before stopping it to perform analysis. The reachability problem precludes the execution of some of the deeper sub-branches. The reason for such a miss is that SimTree was unable to perform a single complete execution of the AIP BT. Please note that the simulator performance was not up to par with model checking. We believe that performance can be optimized in the same spirit as [44]. The reason for the poor performance was equal probability of all events thereby making the execution of nodes deeper in the tree with more than one event in the path, difficult to execute. To overcome this problem we assigned realistic probabilities to the events. For instance in Fig. 5(a) the event AIP ??Batteries_out?? was given lower probability due to the reasons that: (1) Realistically speaking the batteries are not taken out of the system that often and (2) In order to stop the BT from reverting back to the top parent node and resetting the execution from the start. While assigning probability values to all the events we made sure that all nodes are reached thereby ensuring the execution of the BT. The lowest probability value nodes were also executed since we count one run when all the nodes

**Table 5**
Safety properties of AIP.

| No. | Safety property | |
|---|---|---|
| 1 | Concern | As soon as the blockage in the line is detected the pump must be stopped. |
| | LTL | $G(occlusion\_sensor = line\_blocked \Rightarrow X(\neg(aip\_pump = running)))$ |
| | Datalog | :- system(A,B,C,D,E,AIP,F,G,H,I,J,occlusionsensorline_blocked),AIP = aip_pump_running |
| 2 | Concern | As soon as air is detected in the line the pump must be stopped. |
| | LTL | $G(air\_sensor = air\_detected \Rightarrow X(\neg(aip\_pump = running)))$ |
| | Datalog | :- system(A,B,C,D,E,AIP,F,G,H,I,J,air_sensorair_detected), AIP = aip_pump_running |
| 3 | Concern | When the drug volume reaches zero the pump must immediately be stopped. |
| | LTL | $G(\neg(aip\_volume = empty) \Rightarrow X(\neg(aip\_pump = running)))$ |
| | Datalog | :- system(A,aip_volume_empty,B,C,D,E,AIP,F,G,H,I,J), not(AIP = aip_pump_stopped) |

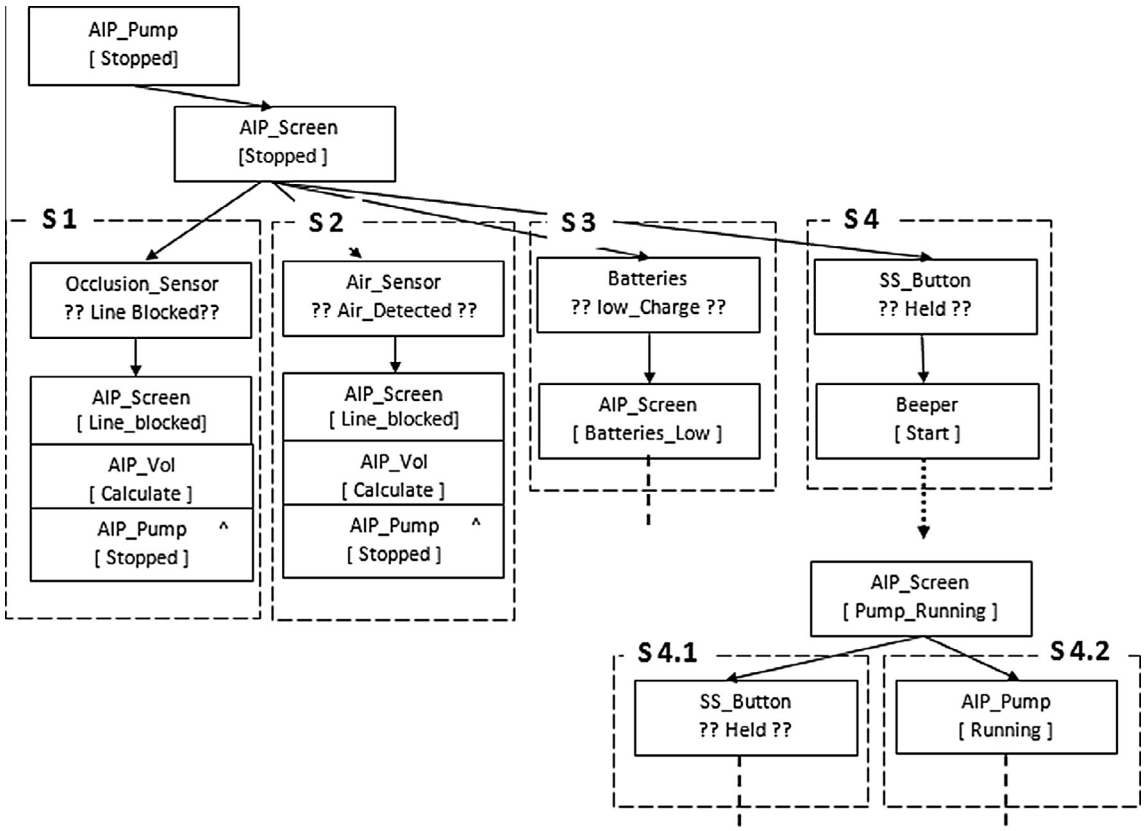



**Fig. 18.** AIP sub-tree that violates the safety property.

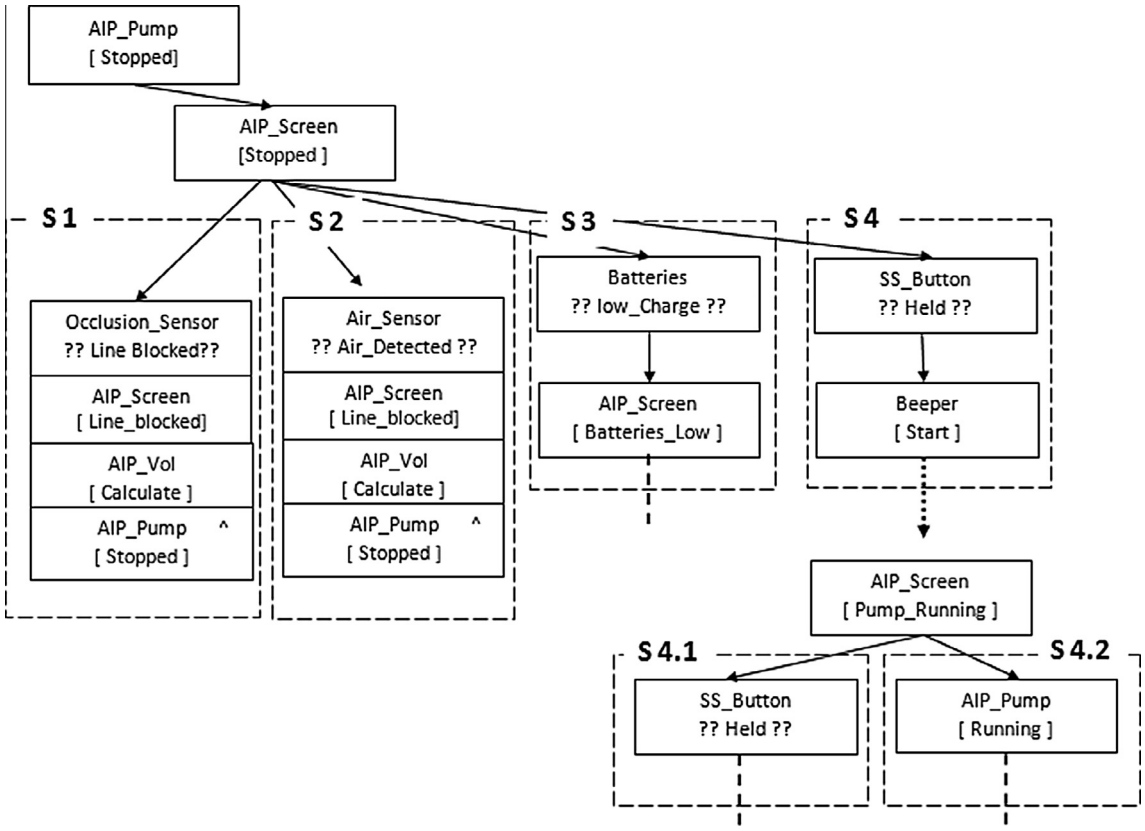



**Fig. 19.** Correction of AIP sub-tree that violates the safety property.

**Table 6**
Event type Nodes and the probability values assigned to them.

| No | Event type Nodes | Case 1 (%) | Case 2 (%) | Case 3 (%) |
|---|---|---|---|---|
| 1 | Batteries In | 90 | 90 | 90 |
| 2 | Batteries Out | 10 | 10 | 10 |
| 3 | Screen Timer Timeout After Two Minutes | 30 | 20 | 10 |
| 4 | Occlusion Sensor Blocked | 30 | 20 | 10 |
| 5 | Air Detected | 30 | 20 | 10 |
| 6 | Low Charge in Batteries | 30 | 20 | 10 |
| 7 | Beeper has Beeped Three Times | 70 | 70 | 70 |
| 8 | Button Held | 80 | 80 | 80 |
| 9 | Button Released | 40 | 80 | 40 |
| 10 | Pump Timer Timed Out | 90 | 90 | 90 |

are executed. The simulator SimTree allowed setting the number of runs in the beginning which gives more control to the user when it comes to determining how exhaustively the AIP Behavior Tree should execute. Currently we did three simulation runs, each with three different sets (Refer to Table 6) of probability values and found the third one to be the most optimal. As a result of which the analysis done using Datalog queries was reduced from hours to minutes. This strategy results in more practical and realistic simulation of the AIP BT. Having said that our future work include optimization of Datalog code. Tables 6 below show the three sets of probability values that we assign to the event type nodes.

Analysis of the AIP BT shows that the requirement simulation can help in analyzing and validating the requirements by executing them to generate traces or by checking for specific execution paths. We have used BTs to specify the natural language requirements that make use of a scalable and repeatable BE process. The industrial trials of the BE process suggest that the approach can be useful in early detection of defects in natural language requirements through translation to BTs. However the process cannot be hundred percent fool proof and defects can also be introduced during the modeling process. Through this case study we were able to identify modeling defects of critical nature which may not necessarily be introduced due to defects in the natural language requirements. The advantage here is that additional filter for early defect detection has been introduced via formal modeling and specification and by making use of simulation. To carry out the simulation we have translated Behavior Trees (BTs) into Datalog rules and facts. This translation can help generate a database of the executed BTs which can be queried for the purpose of checking the properties of interests, generating the traces and for checking the liveness properties. However, the execution time for generating Datalog and the response time of the queries suggest that a code optimization strategy needs to be developed in the same spirit as the BTs slicing strategy presented in [44].

## 6. Conclusion and future work

In conclusion, our main contribution is to provide transformation of a formal notation – BTs into the SimTree simulator. BT's salient feature is bridging the informal-formal gap that is achieved by the underlying BE process. Modeling and specification done in BTs preserves the essence of original requirements while providing support for automated analysis through different tools. We demonstrated how analysis using simulation can help in analyzing and validating the requirements by the use of SimTree simulator. We achieved this by carrying out a M2M transformation to define a mapping between BT and SimTree metamodels followed by M2T transformation to generate code for the simulator. As a proof of concept and to check the soundness of our transformation we used a published case study, the AIP, and reported the findings. We were able to simulate the requirements of the AIP BT by generating Datalog rules and facts which form the model of any Datalog program. Our approach is different from the previous ones in the sense that the Datalog model can be expanded as desired by increasing the number of runs in the simulator SimTree and analysis can be carried out for a desired state-space of the system. The model expansion capability of Datalog is very useful in capturing large simulation results. The incremental expansion of Datalog model helps in controlling how exhaustively a given BT has to be executed by specifying desired number of runs in the simulator. Simulator runs can initially start from a smaller number and can be incrementally increased. Once enough confidence is gained, more exhaustive analysis can be carried out. We queried the Datalog to check for the safety properties of the system. Such an analysis carried out using simulation is helpful in finding and removing requirement defects early in the development life cycle. We found fundamental flaws in the AIP specifications while carrying out its execution in SimTree. The Datalog queries upon violation of the constraint generated tuples showed the requirements errors. Upon generating the traces and carefully analyzing the BT, we were able to pinpoint the cause of the error. Furthermore the SimTree can be enhanced to support the step by step visualization of the execution and provide support for carrying out FMEA by controlling the components of a given Behavior Tree.

Our current implementation only translates nodes of *event*, *selection* and *state* type behavior. In the thread control we have only selected *reversion* nodes. Our future work involves capturing all the behavior and thread control of the BTs notation. Another limitation is the hard coded probability and number of runs values. A user interface can be provided to curb this limitation and is fairly simple and can be dealt with easily. We also look forward to optimize the performance of the SimTree using various statistical techniques. Doing so would upgrade the performance of the Datalog queries as the unnecessary data generation by the SimTree will be curtailed.

## References

[1] J.M. Rushby, Model checking and other ways of automating formal methods, in: Position Pap. Panel Model Checking Concurr. Programs Softw. Qual. Week San Franc, 1995.
[2] C. Heitmeyer, On the need for practical formal methods, in: Formal Techniques in Real-Time and Fault-Tolerant Systems, 1998, pp. 18-26.
[3] O. Hasan, S. Tahar, N. Abbasi, Formal reliability analysis using theorem proving, IEEE Trans. Comput. 59 (5) (2010) 579–592.
[4] P. Cousot, R. Cousot, Verification of embedded software: problems and perspectives, in: Embedded Software, 2001, pp. 97-113.
[5] A. Lamsweerde, Formal specification: a roadmap, in: Proceedings of the Conference on the Future of Software Engineering, 2000, pp. 147–159.
[6] A.M. Christie, Simulation: an enabling technology in software engineering, CROSSTALK– J. Def. Softw. Eng. 12 (4) (1999) 25–30.
[7] R.G. Dromey, Using behavior trees to model the autonomous shuttle system, in: 3rd International Workshop on Scenarios and State Machines: Models, Algorithms and Tools, (SCESM04), Edinburgh, 2004.
[8] S. Zafar, R.G. Dromey, Managing complexity in modelling embedded systems, in: Systems Engineering/Test and Evaluation Conference SETE2005, 2005.
[9] R.G. Dromey, Architecture as an emergent property of requirements integration, in: STRAW'03 Second International SofTware Requirements to Architectures Workshop, 2003, p. 77.
[10] L. Grunske, K. Winter, N. Yatapanage, Defining the abstract syntax of visual languages with advanced graph grammars—a case study based on behavior trees, J. Vis. Lang. Comput. 19 (3) (2008) 343–379.
[11] R.J. Colvin, I.J. Hayes, A semantics for Behavior Trees using CSP with specification commands, Sci. Comput. Program. 76 (10) (2011) 891–914.
[12] S. Zafar, R. Colvin, K. Winter, N. Yatapanage, R.G. Dromey, Early validation and verification of a distributed role-based access control model, in: Software Engineering Conference, 2007, APSEC 2007, 14th Asia-Pacific, 2007, pp. 430–437.
[13] C. Smith, K. Winter, I. Hayes, G. Dromey, P. Lindsay, D. Carrington, An environment for building a system out of its requirements, in: Proceedings of the 19th IEEE International Conference on Automated Software Engineering, 2004, pp. 398–399.
[14] D. Powell, Requirements evaluation using behavior trees-findings from industry, in: Australian Software Engineering Conference (ASWEC'07), 2007.
[15] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, ATL: a model transformation tool, Sci. Comput. Program. 72 (1) (2008) 31–39.

[16] F. Allilaire, J. Bézivin, F. Jouault, I. Kurtev, ATL-eclipse support for model transformation, in: Proceedings of the Eclipse Technology eXchange workshop (eTX) at the ECOOP 2006 Conference, Nantes, France, 2006, vol. 66.
[17] T. Mens, P. Van Gorp, A taxonomy of model transformation, Electron. Notes Theor. Comput. Sci. 152 (2006) 125–142.
[18] T. Myers, TextBE: a textual editor for behavior engineering, in: Proceedings of the 3rd Improving Systems and Software Engineering Conference (ISSEC), 2011.
[19] S. Ceri, G. Gottlob, L. Tanca, What you always wanted to know about Datalog (and never dared to ask), IEEE Trans. Knowl. Data Eng. 1 (1) (1989) 146–166.
[20] F. Sáenz-Pérez, Outer joins in a deductive database system, Electron. Notes Theor. Comput. Sci. 282 (2012) 73–88.
[21] E. Hajiyev, M. Verbaere, O. de Moor, K. De Volder, Codequest: querying source code with datalog, in: Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2005, pp. 102–103.
[22] F. Besson, T. Jensen, Modular class analysis with datalog, Static Anal. (2003) 1075.
[23] R. Douence, Relational Aspects for Context Passing Beyond Stack Inspection, in: Software Engineering Properties of Languages and Aspect Technologies, 2006.
[24] D.H. Stamatis, Failure Mode and Effect Analysis: FMEA from Theory to Execution, Asq Press, 2003.
[25] S. Zafar, Integration of Access Control Requirements into System Specifications, GRIFFITH UNIVERSITY, 2008.
[26] R. Colvin, L. Grunske, K. Winter, Timed behavior trees for failure mode and effects analysis of time-critical systems, J. Syst. Softw. 81 (12) (2008) 2163–2182.
[27] S.-K. Kim, T. Myers, M.-F. Wendland, P.A. Lindsay, Execution of natural language requirements using State Machines synthesised from Behavior Trees, J. Syst. Softw. 85 (11) (2012) 2652–2664.
[28] L. Wen, R. Colvin, K. Lin, J. Seagrott, N. Yatapanage, G. Dromey, 'Integrare', a collaborative environment for behavior-oriented design, Cooperative Design, Visualization, and Engineering, Springer, 2007. pp. 122–131.
[29] S. Zafar, A light-weight formal approach for modeling, verifying and integrating role-based access control requirements, in: Software Engineering Conference, 2009. APSEC'09. Asia-Pacific, 2009, pp. 257–264.
[30] L. Grunske, P. Lindsay, N. Yatapanage, K. Winter, An automated failure mode and effect analysis based on high-level design specification with behavior trees, Integrated Formal Methods (2005) 129–149.
[31] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saıdi, N. Shankar, An overview of SAL, in: Proceedings of the 5th NASA Langley Formal Methods Workshop, 2000.
[32] E.M. Clarke, O. Grumberg, D.A. Peled, Model Checking, MIT press, 1999.
[33] A. Pnueli, The temporal logic of programs, in: 18th Annual Symposium on Foundations of Computer Science, 1977, 1977, pp. 46–57.
[34] Y.K. Hinz, Datalog Bottom-Up is the Trend in the Deductive Database Evaluation Strategy, Tech. Rep. INSS 690, University of Maryland, 2002.
[35] V. Hoffmann, H. Lichter, A model-based narrative use case simulation environment, in: ICSOFT (2), 2010, pp. 63–72.
[36] D. Harel, R. Marelly, Specifying and executing behavioral requirements: the play-in/play-out approach, Softw. Syst. Model. 2 (2) (2003) 82–107.
[37] D. Harel, I. Segall, Planned and traversable play-out: a flexible method for executing scenario-based programs, Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2007. pp. 485–499.
[38] M. Auguston, C. Whitcomb, System architecture specification based on behavior models, DTIC Document (2010).
[39] K. Shimizu, D.L. Dill, Deriving a simulation input generator and a coverage metric from a formal specification, in: Proceedings of the 39th Annual Design Automation Conference, 2002, pp. 801–806.
[40] J. Barnat, L. Brim, J. Beran, T. Kratochvila, Í.R. Oliveira, Executing model checking counterexamples in Simulink, in: 2012 Sixth International Symposium on Theoretical Aspects of Software Engineering (TASE), 2012, pp. 245–248.
[41] A. Milicevic, D. Rayside, K. Yessenov, D. Jackson, Unifying execution of imperative and declarative code, in: Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 511–520.
[42] K. Winter, Formalising behaviour trees with CSP, in: Integrated Formal Methods, 2004, pp. 148-167.
[43] L. Grunske, R. Colvin, K. Winter, Probabilistic model-checking support for FMEA, in: Fourth International Conference on the Quantitative Evaluation of Systems, 2007, QEST 2007, 2007, pp. 119–128.
[44] N. Yatapanage, K. Winter, S. Zafar, Slicing behavior tree models for verification, Theoretical Computer Science, Springer, 2010. pp. 125–139.