Probabilistic Timed Behavior Trees

Robert Colvin, Lars Grunske, Kirsten Winter

ARC Centre for Complex Systems,
School of Information Technology and Electrical Engineering
University of Queensland, Australia

Abstract The *Behavior Tree* notation has been developed as a method for systematically and traceably capturing user requirements. In this paper we extend the notation with *probabilistic* behaviour, so that reliability, performance, and other dependability properties can be expressed. The semantics of probabilistic timed Behavior Trees is given by mapping them to probabilistic timed automata. We gain advantages for requirements capture using Behavior Trees by incorporating into the notation an existing elegant specification formalism (probabilistic timed automata) which has tool support for formal analysis of probabilistic user requirements.
Keywords: Behavior Trees, probabilities, timed automata, model checking

1 Introduction

Representing the user requirements of a large and complex system in a manner that is readable by the client and preserves their vocabulary and intention (validatable), while also having a formal underpinning (verifiable), is an important task for systems engineering. The *Behavior Tree* (BT) notation [1] is a graphical language that supports a behaviour-oriented design method for handling real-world systems [2]. The notation facilitates systematic and traceable translation of natural language requirements which structures the compositional and behavioural information. The notation includes a core subset which has a formal basis [3] and can be model checked [4].

Currently the Behavior Tree notation does not have a syntax for expressing probabilistic behaviour. Such behaviour is important in system specification as many systems specify, for instance, hardware dependability requirements or probabilistic measures on performance. In this paper, we extend the Behavior Tree (BT) notation to include probabilistic choice, thereby increasing the expressiveness of the language and also allowing stochastic properties to be model checked. The new notation, which we call *probabilistic timed Behavior Trees* (ptBTs), is an extension of *timed Behavior Trees* (tBTs), which are introduced in [5]. It allows the user to model timed as well as probabilistic behaviour.

The contributions of the paper are: 1) an operational semantics for timed Behavior Trees in terms of timed transition systems, based on their mapping to *timed automata* [6] given in [5]; and 2) the syntax and semantics of probabilistic timed Behaviour Trees, which extend those for timed Behavior Trees, and

are based on *probabilistic timed automata* [7] and probabilistic timed transition systems. We use two examples to demonstrate the extension, and describe how probabilistic timed Behaviour Trees can be model checked.

The paper is structured as follows. Section 2 introduces Behavior Trees and their timed extension, and in Section 3 their semantics is given in terms of timed automata. In Section 4 probabilities are introduced to the timed BT notation and a semantics is given in terms of probabilistic timed automata. Section 5 gives two examples of probabilistic timed Behavior Trees and explains how they were model checked using PRISM [8].

2 Preliminaries on Behavior Trees

As preliminaries we introduce the Behavior Tree notation and their extension to timed Behavior Trees.

2.1 Behavior Trees

The *Behavior Tree* (BT) notation [1] is a graphical notation to capture the functional requirements of a system provided in natural language. The strength of the BT notation is two-fold: Firstly, the graphical nature of the notation provides the user with an intuitive understanding of a BT model - an important factor especially for use in industry. Secondly, the process of capturing requirements is performed in a stepwise fashion. That is, single requirements are modelled as single BTs, called *individual requirements trees*. In a second step these individual requirement trees are composed into one BT, called the *integrated requirements tree*. Composition of requirements trees is done on the graphical level: an individual requirements tree is merged with a second tree (which can be another individual requirements tree or an already integrated tree) if its root node matches one of the nodes of the second tree. Intuitively, this merging step is based on the matching node providing the point at which the preconditions of the merged individual requirement tree are satisfied. This structured process provides a successful solution for handling very large requirements specifications [1,2].

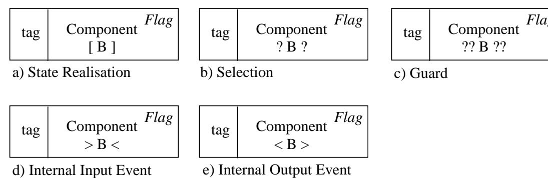

Figure1. BT node types

The syntax of the BT notation comprises nodes and arrows. The notation contains a rich variety of node types for expressing behaviour; in this paper we

focus on a core subset of the language which models state tests and updates and event initiation and response. Each BT node type in Figure 1 refers to a particular *component*, C, and a *behaviour*, B, and is optionally marked by one or more *flags*. Nodes also contain a *tag*, which is used for traceability; since the tags have no effect on the semantics, we will ignore them for the purposes of this paper. The nodes types in Figure 1 are described below.

(a) A *state realisation*, where B is either a simple state name or an expression. A state realisation node models that C realises (enters) state B. For example, the root node of Figure 2 models that initially the *Power* component is in state *on*.
(b) A *selection*, where B is a condition on C's state; the control flow terminates if the condition evaluates to false.
(c) A *guard*, where B is a condition on C's state, as with (b); however, the control flow can only pass the guard when the condition holds, otherwise it is blocked and waits until the condition becomes true.
(d-e) An event modelling communication and data flow between components within the system, where B specifies an event; the control flow can pass the internal event node when the event occurs (the message is sent), otherwise it is blocked and waits; the communication is synchronous.

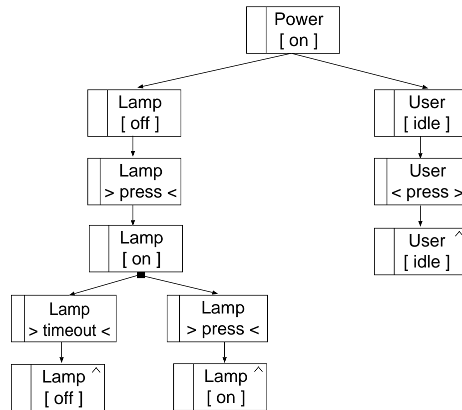

Figure2. Example: A simple lamp and its user

The control flow of the system is specified by either a single arrow leaving a node, for *sequential* flow, or multiple arrows leaving a node, for *concurrent* or *alternative* flow. In addition, *atomic* flow is specified by an line with no arrowhead; this indicates that the behaviour of the child node occurs immediately after the behaviour of the parent node. We note that more than one input/output event is not allowed within an atomic block of nodes, since this could induce deadlock (for a more detailed discussion see [5]).

The example in Figure 2 shows three types of edges: after the initialisation the control flow branches into two concurrent threads, the left modelling the behaviour of the lamp, and the right modelling an abstract user. The lamp thread contains alternative flow, when either a timeout event happens which causes the lamp to switch off, or another press signal is send by the user. The lamp waits for either of these events to occur. The first one to happen determines the flow of control. This alternative flow is marked by the black box on the branching edges.

A flag in BT node can specify: (a) a *reversion* node, marked by '^', if the node is a leaf node, indicating that the control flow loops back to the closest matching ancestor node (a matching node is a node with the same component name, type and behaviour) and all behaviour begun at that node initially is terminated; (b) a *referring* node, marked by '$\sim$', indicating that the flow continues from the matching node; (c) a *thread kill* node, marked by '$--$', which kills the thread that starts with the matching node, or (d) a *synchronisation* node, marked by '$=$', where the control flow waits until all other threads with a matching synchronisation node have reached the synchronisation point. Every leaf node in Figure 2 is marked as a reversion node; we do not utilise the other flags in the examples in this paper.

2.2 Timed Behavior Trees

Timed Behavior Trees (tBTs), originally introduced in [5], extend BTs with the notion of real-valued clocks for expressing timing behaviour. The timing information expressed by timed automata [6] was adopted. All clocks are initialised to zero and progress simultaneously with the same rate. Clocks can be reset at any time, and they can constrain the behaviour in terms of guards and invariants: a guard over a clock restricts the time when a step can be taken, and an invariant restricts the time a component can remain in a state without changing to the next state. The tBT notation therefore extends a BT node by three slots: a *guard* **G** over clock values, a *reset* **R** of clocks, and an *invariant* **I** over clocks. (If not relevant to a particular node, the slots may be omitted.) As with timed automata, we restrict clock invariants to be expressions of the form $x \oplus t$, where x is a clock, t evaluates to an integer value, and $\oplus$ is one of $<, \leq, =, \geq, >$.

As an example, in Figure 3 we augment the lamp Behavior Tree of Figure 2 with explicit timing constraints. The thread on the left hand side introduces the clock variable x, which is reset as soon as the event *press* is received. When the lamp realises the state *on* it must satisfy the invariant $x \leq 5$, modelling that the lamp can remain in state *on* for at most 5 time units before switching off (after exactly 5 time units) or the user presses the button. If the user presses the button while the lamp is on, the lamp may stay on for an additional 5 time units, as indicated by the reset of clock x. In the right-hand thread, a second clock y enforces a more specific timed behaviour in that the user cannot press the button twice within 1 time unit.

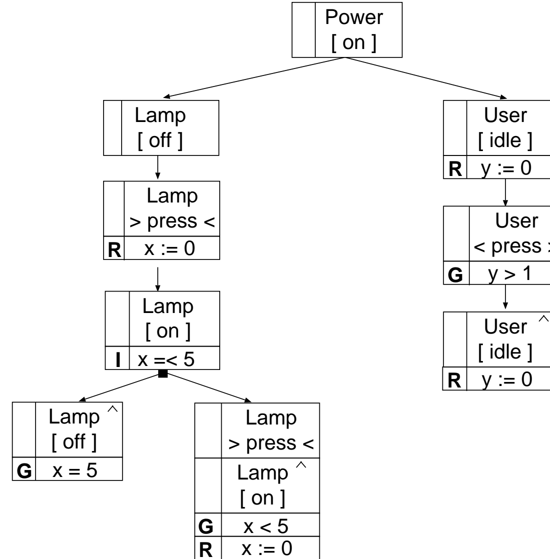

Figure3. Timed system of a lamp and its user in tBT

3 Semantics of Timed Behavior Trees

A timed Behavior Tree (tBT) can be defined as a finite automaton, which contains state variables and clock variables, a finite set of *locations*, and a finite set of labelled *edges*. Locations model the states of the system, abstracting from the evaluation of state and clock variables. Edges symbolise the transitions between the locations. A transition from one location to the next can be guarded and it can perform one or more actions as well as a number of clock resets. Each edge also has an optional synchronisation event.

Components in a tBT are treated as state variables in a timed automaton, while events are treated as timed automaton synchronisations. A Behavior Tree node represents an *edge* in a timed automaton, as nodes embody state changes and events. Each arrow in a BT (except for atomic flow) corresponds to a *location* in a timed automaton. The guards and updates (including resets) of clock and state variables, and synchronisation events, are therefore added to the edges, though clock invariants are pushed to the location following the edge. The general mapping for a node is given in Figure 4.

Nodes. More concretely, the nodes in Figure 1 may be represented as follows (altering the *component_behaviour* section in Figure 4). State realisations are mapped to an update of the relevant component, while a guard node is mapped to a guard on the component. Both input and output events are mapped to synchronisations of the same name, with input events decorated with '?' and output events with '!'. The transfer of data through events may be modelled

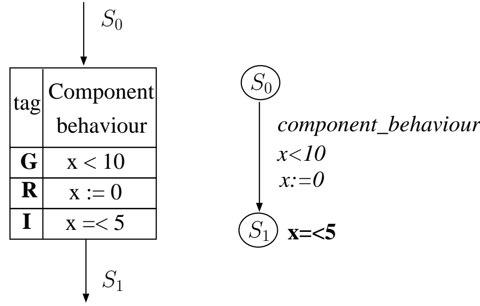

Figure4. tBT node (left) and corresponding timed automaton (right)

using state variables. A selection node requires the addition of an extra edge and terminal state (S_2) for the case where the condition is not satisfied; this is shown in Figure 5.

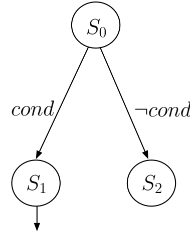

Figure5. Behaviour for selections

Control flow. Sequential flow, as mentioned above, maps to a location, while alternative flow maps straightforwardly to nondeterministic choice. Nodes joined by atomic flow are joined together so that their updates and guards are combined into one transition. Because we restrict atomic flow in tBTs to contain only one synchronisation, this representation is straightforward.

Concurrency. We will call tBTs without concurrent flow of control *sequential tBTs*. A sequential tBT maps to a single timed automaton as described above. Timed BTs with concurrent branching, called *concurrent tBTs*, map to a network of automata, acting concurrently and synchronising on events. Each thread maps to a single automaton, which has to be invoked at a particular point in the control flow, namely the branching point that starts the thread. Therefore, each single automata has an initial location which models the thread being disabled. The location *disabled* can be exited only via an edge that is labelled with the special synchronisation event *invoke?*. The process that starts the thread sends the matching synchronisation event *invoke!* and terminates, i.e., goes itself to the location *disabled*.

Flags. We may now specify how a tBT node's flags are represented. Firstly we note that both reversion nodes and thread kill nodes terminate the behaviour of processes at an arbitrary point in their execution. For each automaton p that may be killed by process q, we introduce a synchronisation event *kill_q?*, and augment p with edges labelled with *kill_q?* leading from each location in p to the disabled location (this approach introduces less overhead than if we were to take the approach of associating *kill*? events with the process being killed). This way, an automaton's behaviour is terminated whenever the corresponding *kill_q*! event is received. In Figure 6 we depict the user thread from Figure 3 as a timed automaton in a network system, assuming that it may be killed by process q at any time.

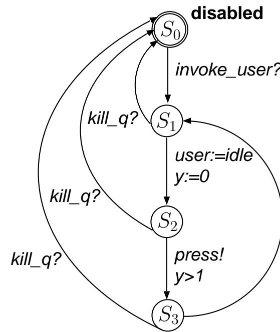

Figure6. Timed automaton simulating the user thread within the lamp system

A node with a reversion flag is typically modelled as a transition from the current location to the location immediately after the edge representing the target node. In general we choose the after-location rather than the before-location to allow for resetting of local clocks (see [5] for why reversions may have different clock resets to their matching node). However, when there are no resets in the reversion node, or if the resets are identical, we can more simply represent reversions as an edge to the before-location. For the examples in this paper, we adopt this more straightforward approach. The reversion transition is labelled with *kill_q!*, which terminates the behaviour of all automata, if any, that are invoked after the matching node (in Figure 6 there are none). A node with a referring flag is modelled simply as a transition from the current location to the location preceding the target node. A node with a kill flag generates a *kill!* event, and a node with a synchronisation flag is modelled directly as a synchronisation transition in a timed automaton.

3.1 Operational Semantics

The semantics of a sequential tBT is given as as a timed automaton. To a large extent our definitions follow the definitions of the operational semantics of timed

automata in [6]. We divert from these, however, where it is suitable for modelling Behavior Trees, e.g., we separate actions (state updates) from synchronisations, and explicitly define state variables to represent components, which can be of any type, whereas in [6] state variables are treated as a special kind of clock variable.

Let $\mathcal{V}$ be a finite set of state variables, and $\mathcal{C}$ be a finite set of clocks. Let $\Sigma(\mathcal{V})$ denote the set of actions (representing state updates), and let $\mathcal{G}(\mathcal{V})$ denote the set of conditions over state variables (representing guards and selections). Let $\mathcal{G}(\mathcal{C})$ be a set of guards over clocks, $\mathcal{R}(\mathcal{C})$ a set of clock resets, and $\mathcal{I}(\mathcal{C})$ a set of clock invariants. We write *skip* to represent an action or clock reset which does has no effect, and *true* for variable and clock guards which are always satisfied. Let Θ be the synchronisation events, with $\varepsilon \in \Theta$ a distinguished element which represents an internal step, i.e., no synchronisation.

Definition 1. *A sequential tBT is a tuple* $\langle L, l_0, E, I \rangle$ *where*

- L *is a finite set of locations*
- $l_0 \in L$ *is the initial location*
- $E \subseteq L \times \Theta \times \mathcal{G}(\mathcal{C}) \times \mathcal{G}(\mathcal{V}) \times \mathcal{R}(\mathcal{C}) \times \Sigma(\mathcal{V}) \times L$ *is the set of edges*
- $I : L \to \mathcal{I}(\mathcal{C})$ *is the mapping that assigns clock invariants to locations.*

We use the notation $l \xrightarrow{s,g,c,r,a} l'$ *if* $(l, s, g, c, r, a, l') \in E$.

As an example, consider the sequential tBT representing the user in Figure 6. There are four locations, with the initial location S_0. The invocation edge is the tuple $(S_0, \mathit{invoke_user}?, \mathit{true}, \mathit{true}, \mathit{skip}, \mathit{skip}, S_0)$. The $\mathit{User}[\mathit{idle}]$ edge is the tuple $(S_1, \varepsilon, \mathit{true}, \mathit{true}, (y \coloneqq 0), (\mathit{User} \coloneqq \mathit{idle}), S_2)$, while the $\mathit{User}\langle \mathit{press} \rangle$ edge is the tuple $(S_2, \mathit{press}!, (y > 1), \mathit{true}, \mathit{skip}, \mathit{skip}, S_3)$. The reversion is represented as an edge back to the location S_1, i.e., $(S_3, \varepsilon, \mathit{true}, \mathit{true}, \mathit{skip}, \mathit{skip}, S_1)$. The edge corresponding to a *kill_q* event occurring while the lamp process is in location S_3 is $(S_3, \mathit{kill_q}?, \mathit{true}, \mathit{true}, \mathit{skip}, \mathit{skip}, S_0)$.

Before giving the operational semantics we introduce some notation. We use *clock assignments* to denote the progress of time and with it changing clock values, and *variable assignments* to monitor the evaluation of the state variables. A clock assignment is a mapping from clocks $\mathcal{C}$ to non-negative real numbers $\mathbb{R}_+$. If u is a clock assignment, then $u + d$ denotes the clock assignment that maps all $c \in \mathcal{C}$ to $u(c) + d$. Resetting clocks is denoted as $[\overline{r} \mapsto 0]u$ which maps all clocks in $\overline{r} \subseteq \mathcal{C}$ to 0 and leaves all other clocks in $\mathcal{C} \setminus \overline{r}$ unchanged. Let v be a variable assignment mapping all variables in $\mathcal{V}$ to a value in their domain. Updating state variables is denoted as $[\overline{x} \mapsto \overline{e}]v$ which changes the variable assignment to map variables $\overline{x} \subseteq \mathcal{V}$ to corresponding values in $\overline{e}$ and leaves all other variables unchanged.

The semantics of a sequential tBT can be given as a timed transition system, in which a state of a sequential tBT can be given as a tuple consisting of a location, a variable assignment, and a clock assignment, i.e., $\langle l, v, u \rangle$.

There are two types of transitions possible: the system either delays for some time (*delay step*) or takes one of the enabled transitions (*action step*).

Definition 2. *The semantics of a sequential tBT is a timed transition system with states $\langle l, v, u\rangle$ and transitions as follows.*

- $\langle l, v, u\rangle \xrightarrow{d} \langle l, v, u+d\rangle$ *(delay step)*
  if u and $u+d$ satisfy $I(l)$ for a $d \in \mathbb{R}_+$
- $\langle l, v, u\rangle \xrightarrow{\alpha} \langle l', v', u'\rangle$ *(action step)*
  if $l \xrightarrow{s,g,c,r,a} l'$, u satisfies g, v satisfies c,
  $v' = [\overline{x} \mapsto \overline{e}]v$ if $a = (\overline{x} \mapsto \overline{e})$,
  $u' = [\overline{r} \mapsto 0]u$, and u' satisfies $I(l')$.

According to this definition, if a process is in a location from where no action step is enabled by the time the clock evaluation violates the location invariant, no further step is possible (the delay step is also disabled) and the process halts. Furthermore, this definition allows for indefinitely many delay steps if the automaton is in a state for which no location invariant is specified (i.e., any u and $u+d$ will satisfy *true*).

3.2 Concurrent timed Behavior Trees

The semantics of a concurrent tBT can now be given as a *network* of timed automata, i.e., parallel automata that operate in an interleaving fashion using a handshake synchronisation mechanism. A state of a network with n concurrent processes is formalised as a tuple $\langle ls, v, u\rangle$ with ls being a vector of length n of the current locations in each process, v the variable assignment[1] and u the clock assignment. Let l_i denote the i-th element of location vector ls and $ls[l'_i/l_i]$ denote the vector ls with the element l_i being substituted by l'_i. With $I(ls)$ we denote the conjunction of invariants on all locations, i.e., $I(ls) = \bigwedge_i I(l_i)$.

Let $s?, s! \in \Theta$ symbolise reading and sending of a synchronisation event, respectively, also recalling $\varepsilon \in \Theta$ denotes an internal action of the system.

A network can perform three types of steps: a delay step and an action step, both similar to the steps in a single automaton, and also a synchronisation step.

Definition 3. *The semantics of a concurrent tBT is a network of timed transition systems with states $\langle ls, v, u\rangle$ and transitions as follows.*

- $\langle ls, v, u\rangle \xrightarrow{d} \langle ls, v, u+d\rangle$ *(delay step)*
  if u and $u+d$ satisfy $I(ls)$ for a $d \in \mathbb{R}_+$
- $\langle ls, v, u\rangle \xrightarrow{\varepsilon} \langle ls[l'_i/l_i], v', u'\rangle$ *(action step)*
  if $l_i \xrightarrow{\varepsilon,g,c,r,a} l'_i$,
  u satisfies g, v satisfies c,
  $v' = [\overline{x} \mapsto \overline{e}]v$ if $a = (\overline{x} \mapsto \overline{e})$,
  $u' = [\overline{r} \mapsto 0]u$, and u' satisfies $I(ls[l'_i/l_i])$.

[1] State variables are not related to a particular process but treated as global and are therefore accessible by any process.

- $\langle ls, v, u\rangle \xrightarrow{\varepsilon} \langle ls[l'_i/l_i][l'_j/l_j], v', u'\rangle$ *(synchronisation step)*
  if there exists $i \neq j$ *such that*
  1. $l_i \xrightarrow{s?, g_i, c_i, r_i, a_i} l'_i$, $l_j \xrightarrow{s!, g_j, c_j, r_j, a_j} l'_j$
     and u *satisfies* $g_i \wedge g_j$ *and* v *satisfies* $c_i \wedge c_j$
     and
  2. $v' = [\overline{x_i}/\overline{e_i}]([\overline{x_j}/\overline{e_j}]v)$
     if $a_i = (\overline{x_i} \mapsto \overline{e_i})$ *and* $a_j = (\overline{x_j} \mapsto \overline{e_j})$ *and*
  3. $u' = [\overline{r_i} \cup \overline{r_j} \mapsto 0]u$ *and* u' *satisfies* $I(ls[l'_i/l_i][l'_j/l_u])$.

Note that in a synchronisation step the sending process updates the state variables (if its action a contains updates) before the receiving process, facilitating synchronous message passing.

4 Probabilistic timed Behavior Trees

In this section we extend timed Behavior Trees to *probabilistic timed Behavior Trees* (ptBTs). We follow the well-established and expressive approach of annotating transitions with a *probability* that the transition will take place. In the Behavior Tree notation, this means we associate with each node an optional probability slot which contains a number from 0 to 1, i.e., in the range $[0, 1]$. As an example, in which probabilistic choice is used to model component failures, consider the ptBT in Figure 7 which extends the lamp example with a 1% chance that the lamp will fail (e.g., blow a fuse) whenever it is switched from off to on.

For clarity, and without loss of generality, we impose a well-formedness condition on ptBTs that either every *child* node of a node has an associated probability, or none do (a child node is a direct descendant). We have therefore introduced *probabilistic branching* in addition to alternative and concurrent branching. The probabilities in the child nodes must sum to less than or equal to 1. If the probabilities sum to P, and P is less than one, it is understood that with probability $1 - P$ no transition is taken.[2] In the lamp example, the probabilities of the child nodes of the $Lamp\langle press\rangle$ node sum to 1, indicating that one of the actions must be taken. In the user thread, however, the probability of the user pressing the button is 0.3. Thus it is implicit there is a 70% chance that the user will not press the button as time passes.

The mapping of ptBTs to probabilistic timed automata follows that of tBTs for the non-probabilistic constructs in the language, with the addition that probabilities are added to the corresponding edges in the automaton, if the sum of the probabilities is 1. If the probabilities of all the edges leaving a location sum to P for $P < 1$, in general an additional edge looping back to itself with probability

[2] This models an exponentially distributed delay before the next action is taken. In continuous timed systems, such behaviour can also be represented using real-valued *rates*, giving the expected incidence of events per time unit. For simplicity we define ptBTs to contain probabilities only (values in the range [0,1]), but in Section 5 we describe how rates may be introduced for model checking.

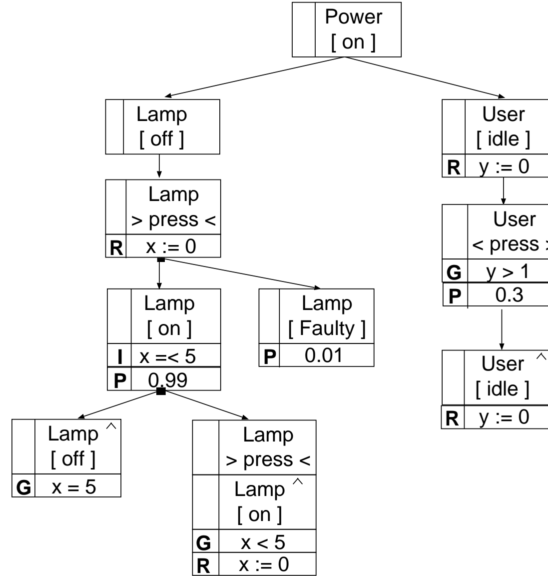

Figure7. Probabilistic timed system of a lamp and its user

$1 - P$ is added, which includes any clock guards – see Figure 8 for an example. An exception to this is if the node is an output event. Because it is not possible to label only one edge in a probabilistic choice in a timed automaton with a synchronisation, an intermediate location is added between the probabilistic choice and the synchronisation – an example is given later in Figure 9.

To specify probabilistic properties we may choose from several different specification languages: continuous stochastic logic (CSL) [10], if the model is deterministic and uses continuous time; probabilistic computation tree logic (PCTL) [11], if the model uses discrete time; or PCTL's extension to probabilistic timed computation tree logic (PTCTL) [12], if the model is nondeterministic and uses continuous time. As an example, if the global time is recorded in clock G, the dependability property "with what probability will the Lamp enter the faulty state before X time units" can be formally stated in PCTL as

$$\mathcal{P}_{=?}(true\ \mathcal{U}\ (lamp = faulty \wedge G \leq X))$$

where $\mathcal{U}$ is the temporal *until* operator, and a property $true\ \mathcal{U}\ \phi$ models *eventually* ϕ.

4.1 Semantics of probabilistic timed Behavior Trees

We give the meaning of ptBTs as probabilistic timed automata. There are several ways in which probabilities may be added to the timed automaton model, e.g., by associating them with edges [13] or with locations [14]. We follow the former

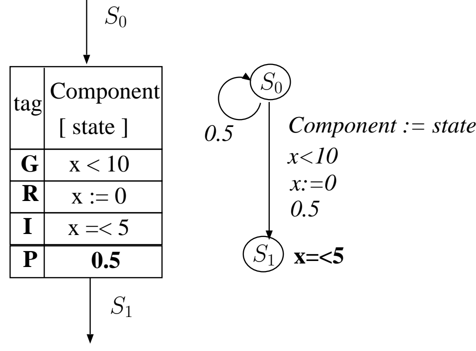

Figure8. ptBT node (left) and corresponding probabilistic timed automaton (right)

approach, and replace the target locations in edges with a *probability mapping* on clock resets, variable updates, and target locations. A probability mapping on type T is a partial function from elements of T to values in the range $[0, 1]$, which sum to 1, i.e.,

$$\mathsf{Dist}(T) \mathrel{\widehat{=}} \{p : T \nrightarrow [0, 1] \mid \sum_{t:\, \mathrm{dom}\, p} p(t) = 1\}$$

Definition 4. *A sequential probabilistic tBT is a tuple* $\langle L, l_0, E, I\rangle$ *where*

- L *is a finite set of locations*
- $l_0 \in L$ *is the initial location*
- $E \subseteq L \times \Theta \times \mathcal{G}(\mathcal{C}) \times \mathcal{G}(\mathcal{V}) \times \mathsf{Dist}(\mathcal{R}(\mathcal{C}) \times \Sigma(\mathcal{V}) \times L)$ *is the set of edges*
- $I : L \to \mathcal{I}(\mathcal{C})$ *is the mapping that assigns clock invariants to locations.*

In Figure 9 we give the graphical representation of the sequential probabilistic timed automaton for the user process in the lamp example, alongside its representation as a tuple. By including resets and updates with the target locations in the distribution we may enforce differing resets and updates depending on how the probabilistic choice is resolved.

The semantics of a probabilistic timed automaton is updated so that an *action step* from location l_1 to l_2 may be taken only if the associated probability is greater than 0. This is given by the second line in the action step constraint below, which is otherwise identical to Definition 2.

Definition 5. *The semantics of a sequential ptBT is a probabilistic timed transition system with states* $\langle l, v, u\rangle$ *and transitions as follows.*

- $\langle l, v, u\rangle \xrightarrow{d} \langle l, v, u + d\rangle$ *(delay step)*
  if u *and* $u + d$ *satisfy* $I(l)$ *for a* $d \in \mathbb{R}_+$
- $\langle l, v, u\rangle \xrightarrow{\alpha} \langle l', v', u'\rangle$ *(action step)*
  if $(l, s, g, c, D) \in E$ *and* $D(r, a, l') > 0$*, and*
  u *satisfies* g*,* v *satisfies* c*,*
  $v' = [\overline{x} \mapsto \overline{e}]v$ *if* $a = (\overline{x} \mapsto \overline{e})$*,*
  $u' = [\overline{r} \mapsto 0]u$*, and* u' *satisfies* $I(l')$

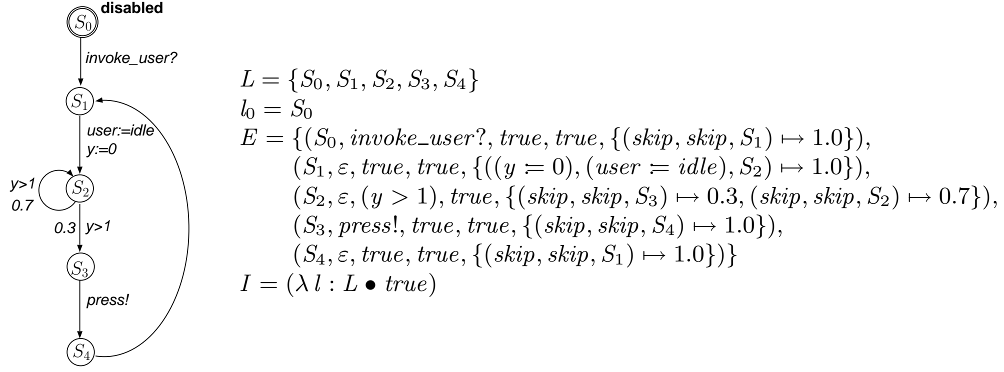

Figure9. Sequential ptBT of the user process as a probabilistic timed automaton, with its representation as a tuple

By augmenting edges with distributions, which therefore represent a set of "probable" edges, rather than assigning a *single* probability for that edge, we are able to more succinctly capture the "sum to one" property, and express nondeterministic behaviour. A non-probabilistic timed automaton can be mapped to a probabilistic timed automaton by replacing each edge (l, s, g, c, r, a, l') with the edge (l, s, g, c, D), where the distribution D maps (r, a, l') to 1.0. Such distributions are called *point distributions*. When a nondeterministic choice is made between two locations, this is represented by two edges, each of which has a point distribution on the target edge.

4.2 Semantics of concurrent probabilistic timed Behavior Trees

The semantics of concurrent ptBTs must also be extended in a similar way to Definition 3, so that both probabilities in a synchronisation step are greater than 0. The definition below differs from Definition 3 in that probabilities are checked to be non-zero in action and synchronisation steps.

Definition 6. *The semantics of a concurrent ptBT is a network of probabilistic timed transition systems with states $\langle ls, v, u\rangle$ and transitions as follows.*

- $\langle ls, v, u\rangle \xrightarrow{d} \langle ls, v, u+d\rangle$ *(delay step)*
  if u and $u+d$ satisfy $I(ls)$ for a $d \in \mathbb{R}_+$
- $\langle ls, v, u\rangle \xrightarrow{\varepsilon} \langle ls[l_i'/l_i], v', u'\rangle$ *(action step)*
  if $(l_i, \varepsilon, g, c, D) \in E_i$ and $D(r, a, l_i') > 0$,
  u satisfies g, v satisfies c,
  $v' = [\overline{x} \mapsto \overline{e}]v$ if $a = (\overline{x} \mapsto \overline{e})$,
  $u' = [\overline{r} \mapsto 0]u$, and u' satisfies $I(ls[l_i'/l_i])$.

- $\langle ls, v, u\rangle \xrightarrow{\varepsilon} \langle ls[l'_i/l_i][l'_j/l_j], v', u'\rangle$ *(synchronisation step)*

  if there exists $i \neq j$ *such that*

  1. $(l_i, s?, g_i, c_i, D_i) \in E_i$ *and*
     $(l_j, s!, g_j, c_j, D_j) \in E_j$ *and*
     $D_i(r_i, a_i, l'_i) \cdot D_j(r_j, a_j, l'_j) > 0$ *and*
     u *satisfies* $g_i \wedge g_j$ *and* v *satisfies* $c_i \wedge c_j$ *and*
  2. $v' = [\overline{x_i}/\overline{e_i}]([\overline{x_j}/\overline{e_j}]v)$
     if $a_i = (\overline{x_i} \mapsto \overline{e_i})$ *and* $a_j = (\overline{x_j} \mapsto \overline{e_j})$ *and*
  3. $u' = [\overline{r_i} \cup \overline{r_j} \mapsto 0]u$ *and* u' *satisfies* $I(ls[l'_i/l_i][l'_j/l_u])$.

The addition of probabilities does not greatly affect the semantics, however it has important implications for model checking. In an execution of the probabilistic Lamp system in Figure 7, the probability of each transition is recorded, and hence it is possible, in exhaustive analysis, to determine the probabilities of each execution. We explore this in more detail in the next section.

5 Model checking probabilistic timed Behavior Trees

In this section we describe how probabilistic timed Behavior Trees may be model checked using the model checker PRISM [8], and provide two examples. PRISM (Probabilistic Symbolic Model Checker) provides model checking facilities for three types of probabilistic models: deterministic time Markov chains (DTMCs), continuous time Markov chains (CTMCs), and Markov decision processes (MDPs) (for an overview of the three models and how they may be model checked, see [9]). To utilise PRISM, we must therefore translate ptBTs into one of the three model types. DTMCs and CTMCs are deterministic, and hence are suitable for deterministic ptBTs. MDPs, which are generalisations of DTMCs, contain nondeterministic choice, though, like DTMCs, are limited to discrete time. Because ptBTs contain nondeterministic choice, we will typically translate a ptBT model into a PRISM MDP for model checking, following guidelines given in [13]; in Section 5.1 we give an example of this using the Lamp ptBT. In Section 5.2 we give a deterministic ptBT, which we model check as a PRISM CTMC.

5.1 Case Study 1 - Lamp Example

Consider the user probabilistic timed automaton given in Figure 9. Its translation into a PRISM MDP is given below.

```
module user
  L: [0..4] init 0;
  user: [0..1] init 0;
  y: [0..MAX_Y] init 0;

  [invoke_user] L=0         -> (L'=1);
  []            L=1         -> (L'=2) & (y'=0) & (user' = 1);
  [time]        L=2 & y>1   -> 0.3: (L'=3) & INC(y)  +
                               0.7: (L'=2) & INC(y);
  [press]       L=3         -> (L'=4);
  []            L=4         -> (L'=1);
  [time]        !(L=2 & y> 1) and !(L=3)
                            -> INC(y);
endmodule
```

The variable L represents the locations, and *user* the user component (0 representing the user's initial state, and 1 representing the *idle* state). We also declare local clock y, which for efficiency reasons is at most MAX_Y. (Within the code, the abbreviation $INC(Y)$ increments y up to the maximum MAX_Y.) Each action line corresponds to an element of E, and is of the form

[sync] *guard* -> *action*

for non-probabilistic behaviour, or

[sync] *guard* -> *prob*1 : *action*1 + *prob*2 : *action*2 + ...

for probabilistic behaviour. The start of a line gives the synchronisation event enclosed in square brackets (which are empty for internal actions), and the guards combine the conditions on clocks and state variables (there is no distinction in PRISM). The action part updates variables and clocks. The translation is straightforward, except that we must explicitly model the advancement of time, as observed in local clock y. Any action which results in the passing of time is synchronised with the global clock (see below) on the event *time*, and any local clocks are incremented (since this is a discrete time model). In addition, the last action line allows time to advance with no action being taken (the guard is used to prevent doubling up of time increments, and to ensure no time passes between the probabilistic choice and the *press* event occuring).

Global time is maintained via the *time* module. After invocation, the *time* module increments the global clock synchronously with all modules which maintain their own local clocks.

```
module time
    time_L : [0..1] init 0;
    global_clock: [0..MAX_GLOBAL_CLOCK] init 0;

    [invoke_clock]  time_L=0 -> (time_state'=1);
    [time]          time_L=1 -> (time_state'=1) & INC(global_clock);
endmodule
```

The full PRISM model also contains a process representing the lamp, and a process which initiates the lamp, timer, and user processes. Having translated the model into an MDP, we may check whether it satisfies reliability requirements written in probabilistic computation tree logic (PCTL) [11]. For example, “The probability that the lamp fails within 100 time units should be less than 10 percent” is given as

```
Pmin<0.1 [ true U (lamp_state = faulty & global_clock <= 100) ]
```

where *faulty* is an abbreviation for the corresponding integer-valued lamp state.

Given a failure probability of 0.01 for the lamp, and that the user presses the button with a probability of 0.3, the model checking showed the model fulfills this requirement.

5.2 Case Study 2 - Viking Example

In this section we give a model with no nondeterminism or local clocks, which can be model checked as a PRISM CTMC. The example involves a group of four Vikings attempting to cross a bridge, from the “unsafe” side to the “safe” side, though the bridge may hold only one at a time – see Figure 10. An individual Viking can step on to the bridge, and then cross to the safe side. However if more than one steps on to the bridge at the same time, they begin arguing, which may result in one or more of the Vikings backing down and returning to the unsafe side of the bridge. (The behaviour of the system can be likened to processes competing for access to a critical section.) For reasons of space we show the thread for only one Viking – the other three are similar.

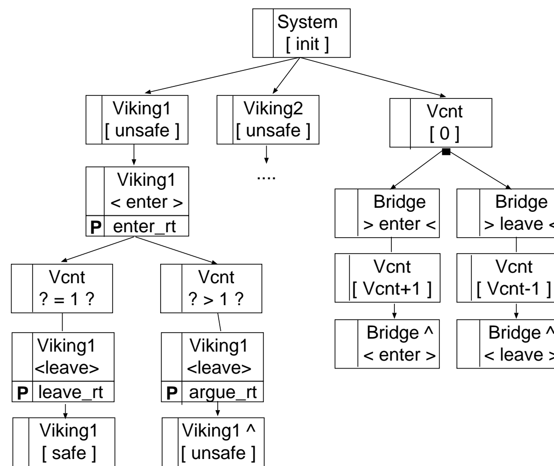

Figure10. Viking Behavior Tree

The Viking BT is specified as follows: initially each Viking is unsafe, and with a *rate* (described below) given by `enter_rt` they enter the bridge. Note that the passage of time is implicit in this model, and that as time passes it becomes more and more likely that an individual Viking will enter the bridge. Entering a bridge is modelled by a synchronisation with the bridge process, which is described below. After entering the bridge, the Viking may leave if he is the only occupant, and does so at the rate given by `leave_rt`. If there is more than one Viking on the bridge, he begins "arguing" with the other Vikings, and may back down and leave the bridge, returning to the unsafe side, at a rate given by `argue_rt`. The bridge process simply maintains a count of the number of Vikings on the bridge by synchronising on leave and enter events from each Viking.

This model may be translated into a CTMC, because it is deterministic and does not contain local clocks. However, CTMCs operate on *rates*, rather than probabilities, and thus the value in the probability slot must be interpreted as a rate. This is straightforward, unless the rate of an event occurring is more than once per time unit (since this would require a value greater than one, i.e., could not be interpreted as a probability). In this example we set our rates to be less than once per second, and leave the extension of ptBTs to use rates as future work. The translation of the Viking ptBT to a PRISM CTMC is straightforward, since time does not need to be explicitly modelled.

We may check various properties of the Viking model, as with the lamp. Because the model is a CTMC, we use continuous stochastic logic (CSL) [10] instead of PCTL. In this case, we may determine which combination of probabilities gives the best performance, as measured by how quickly all Vikings make it to the safe side of the bridge. As an example, we fix the leaving probability at 1.0, i.e., Vikings will exit the bridge to the safe side at the earliest opportunity, and observe the change in probability that results from varying the probabilities of entering and arguing (also called a parameter sweep). This is checked against the following property, which queries the probability of all four Vikings becoming safe within G time units (*all_safe* is an abbreviation for each Viking being in the safe state).

```
P=? [true U<G all_safe]
```

The results of the analysis in PRISM are presented in Figure 11. The graphs show the probabilities that all vikings are safe for varying values of the argue and enter rates with the leave rate set at 1. The graph on the left gives the probabilities for the case where $G = 10$, i.e., all Vikings are safe within 10 time units, while the graph on the right is for the case where $G = 20$. In both cases a higher argue rate gives better performance, but over a longer time span a less aggressive enter strategy gives better performance (optimal enter rate for $G = 10$ is approximately 0.8, while for $G = 20$ it is approximately 0.4).

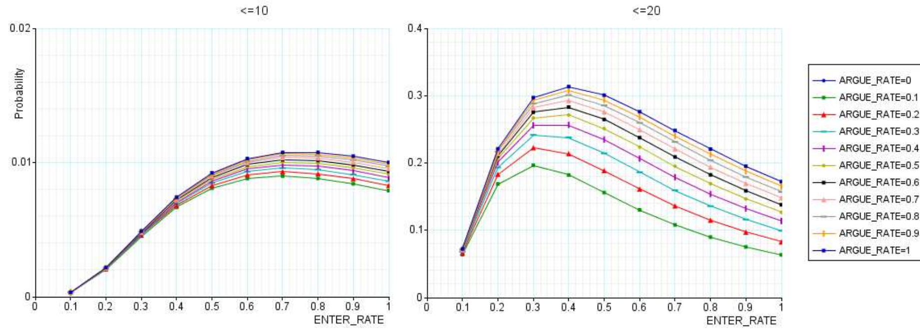

Figure11. Result of PRISM model checking for the Viking example with a parameter sweep over the enter and arguing probabilities

6 Related Work

Various notations have been extended to enable modelling of probabilistic behaviour, e.g., stochastic Petri Nets [15], probabilistic timed CSP [16], stochastic π calculus [17], probabilistic action systems [18], and probabilistic statecharts [19].

From the perspective of probabilistic requirements capture, the closest work to our own is that of Jansen et al. [19], who extend UML statecharts with probabilities. Statecharts, like Behavior Trees, is a graphical notation to support modelling of system requirements. Similarly to our work, the semantics of probabilistic statecharts, which is given in terms on Markov Decision Processes (MDP), provides an interface to the model checker PRISM. In contrast to probabilistic timed BTs, however, probabilistic statecharts do not allow modelling timed behaviour since a notion of clocks is not included. Furthermore, capturing requirements with statecharts is not supported in the same stepwise and individually traceable fashion as in the BT approach outlined in Section 2.

7 Conclusion and Future Work

In this paper we have given a probabilistic extension to the timed Behavior Tree notation and a semantics in terms of probabilistic timed automata (see, e.g., [7]), as well as given a more rigorously defined semantics of timed Behavior Trees [5]. The extension was demonstrated with two examples, which were also model checked in PRISM [8]. The notation extension was designed to be straightforward for system modellers to incorporate when capturing probabilistic requirements, as well as allow the probabilistic behaviour of faulty components to be specified. Probabilistic system properties may then be formally verified after translation to probabilistic timed automata.

The addition of probabilistic choice to Behavior Tees was particularly motivated by the need for modelling faulty behaviour of safety-critical embedded

systems. Consequently, in future work, we will enhance the automatic Failure Mode and Effect Analysis (FMEA) for Behavior Trees [4]. The procedure currently uses fault injection experiments and model checking of the resulting Behavior Tree to determine whether the injected failure leads to a hazard condition, which is specified as a normal temporal logical formula. However, a limitation of this procedure is that the model checker will generate counter examples that are relatively improbable. With the results presented in this paper, we propose to assign to each of these injected faults an occurrence rate, and perform an analysis of the resulting behaviour with probabilistic model checking. We will then be able to analyse hazard conditions together with their tolerable hazard probabilities.

As it stands, current probabilistic model checking approaches work with exponential distributions only, but in practice, many faults are distributed differently; in particular, many faults are Weibull distributed [20], and follow the common "bathtub" curve which models a burn-in and wear-out phase. Consequently, we will investigate how to include arbitrary distributions in the probabilistic timed BT notation, and in what way these distributions can be supported by model checking tools.

References

1. Dromey, R.G.: From requirements to design: Formalizing the key steps. In: Int. Conference on Software Engineering and Formal Methods (SEFM 2003), IEEE Computer Society (2003) 2–13
2. Wen, L., Dromey, R.G.: From requirements change to design change: A formal path. In: Int. Conference on Software Engineering and Formal Methods (SEFM 2004), IEEE Computer Society (2004) 104–113
3. Winter, K.: Formalising Behaviour Trees with CSP. In Boiten, E., Derrick, J., Smith, G., eds.: 4th Int. Conf. on Integrated Formal Methods (IFM 2004). Volume 2999 of LNCS., Springer-Verlag (2004) 148–167
4. Grunske, L., Lindsay, P., Winter, K., Yatapanage, N.: An automated failure mode and effect analysis based on high-level design specification with Behavior Trees. In Romijn, J., Smith, G., van de Pol, J., eds.: Proc. of Int. Conf. on Integrated Formal Methods (IFM 2005). Volume 3771 of LNCS., Springer-Verlag (2005) 129–149
5. Grunske, L., Winter, K., Colvin, R.: Timed Behavior Trees and their Application to Verifying Real-time Systems. In: Proc. of 18th Australian Conference on Software Engineering (ASWEC 2007). (2007) accepted for publication.
6. Bengtsson, J., Wang, Y.: Timed automata: Semantics, algorithms and tools. In Reisig, W., Rozenberg, G., eds.: Lecture Notes on Concurrency and Petri Nets. Volume 3098 of LNCS. Springer-Verlag (2004)
7. Beauquier, D.: On probabilistic timed automata. Theoretical Computer Science **292** (2003) 65–84
8. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In Hermanns, H., Palsberg, J., eds.: Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06). Volume 3920 of LNCS., Springer (2006) 441–444

9. Kwiatkowska, M.: Model checking for probability and time: From theory to practice. In: Proc. 18th Annual IEEE Symposium on Logic in Computer Science (LICS'03), IEEE Computer Society Press (2003) 351–360 Invited Paper.
10. A. Aziz, K. Sanwal, V. Singhal, R. K. Brayton: Verifying continuous-time markov chains. In Rajeev Alur, Thomas A. Henzinger, eds.: Eighth International Conference on Computer Aided Verification CAV. Volume 1102., New Brunswick, NJ, USA, Springer Verlag (1996) 269–276
11. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing **6** (1994) 512–535
12. Kwiatkowska, M., Norman, G., Segala, R., Sproston, J.: Automatic verification of real-time systems with discrete probability distributions. Theoretical Computer Science **282** (2002) 101–150
13. Kwiatkowska, M., Norman, G., Parker, D., Sproston, J.: Performance analysis of probabilistic timed automata using digital clocks. Formal Methods in System Design **29** (2006) 33–78
14. Kwiatkowska, M., Norman, G., Segala, R., Sproston, J.: Verifying quantitative properties of continuous probabilistic timed automata. In Palamidessi, C., ed.: CONCUR. Volume 1877 of LNCS., Springer-Verlag (2000) 123–137
15. Marsan, M.A., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: Modelling with Generalized Stochastic Petri Nets. Wiley Series in Parallel Computing (1995)
16. Lowe, G.: Probabilistic and prioritized models of timed CSP. Theoretical Computer Science **138** (1995) 315–352
17. Priami, C.: Stochastic π calculus. The Computer Journal **38** (1995) 578–589
18. Troubitsyna, E.: Reliability assessment through probabilistic refinement. Nordic Journal of Computing **6** (1999) 320–342
19. Jansen, D.N., Hermanns, H., Katoen, J.P.: A probabilistic extension of UML Statecharts. In: Proceedings of Int. Conf. on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 2002). Volume 2469 of LNCS., Springer-Verlag (2002) 355–374
20. Birolini, A.: Reliability Engineering: Theory and Practice. Third edn. Springer (1999)