Timed Behavior Trees and their Application to Verifying Real-time Systems

Lars Grunske Kirsten Winter Robert Colvin

ARC Centre for Complex Systems
University of Queensland
4072 Brisbane, Australia
{grunske,kirsten,robert}@itee.uq.edu.au

Abstract

Behavior Trees (BTs) are a graphical notation used for formalising functional requirements and have been successfully applied to several case studies. However, the notation currently does not support the concept of time and consequently its application is limited to non-real-time systems. To overcome this limitation we extend the notation to Timed Behavior Trees, which can be semantically defined by timed automata. Based on this extension we are able to include local timing assumptions in a BT model and can verify system-level timing properties with temporal proof methodologies. We validate the use of the new notation by means of a case study. To verify system-level timing properties we translate the model into timed automata and use the tool UPPAAL for timed model checking.

Keywords: Behavior Trees, real time systems, timed automata, model checking, requirements engineering

1. Introduction

For the development of complex dependable systems a good modelling notation is essential. It should allow the development team to capture the requirements in a traceable manner when creating a first model. It should yield models that are easily understood and it should provide tool support for analysis at the early stages of development. The graphical notation of *Behavior Trees* (BTs) provides such a modelling notation [8]. A BT model has a tree-like form that intuitively shows the flow of control of a component-based system. The requirements are to be captured in a stepwise manner, each single requirement leading to an individual tree, which are then integrated into one design tree. This approach provides effective support for the initial modelling phase where requirements need to be reflected in the initial model of the system and allows for collaborative team work. The current tool support for the notation includes a graphical editor, a simulator and an interface to various model checkers enabling safety and liveness checks [15, 18, 10]. Behaviour Trees have been successfully applied to industrial case studies of which some had more than 1500 system requirements. The number of defects in the requirements (i.e., inconsistencies, missing requirements, etc.) that were detected through the BT model was significantly larger than what the companies' own analyses had revealed. The results are unpublished due to non-disclosure agreements.

For many dependable systems the timing of behaviour is critical for the correctness of the overall system model. We therefore want to be able to express timing constraints when modelling and to check if a timed model satisfies given timing requirements. However, the BT notation currently does not support modelling of time and timed behaviour. In this paper we describe an extension of the BT language with a notion of time which allows us to model timing constraints for a component's behaviour. We base our extension on the theory of timed automata [5]. For the extended notation, called *Timed Behavior Trees*, we provide an interface to the UPPAAL tool [4], a model checker for real-time systems. This allows us to check if our model satisfies the given timing requirements.

The paper is organised as follows: Section 2 introduces the BT notation as well as timed automata. In Section 3 we introduce Timed Behavior Trees and define their semantics. Section 4 describes a case study to demonstrate modelling of a timed system with Timed Behavior Trees and verifying its real-time requirements using the UPPAAL tool. Related work is summarised in Section 5 and we conclude with an outlook to future work in Section 6.

2. Preliminaries

As preliminaries we introduce the BT notation and the core concepts of timed automata.

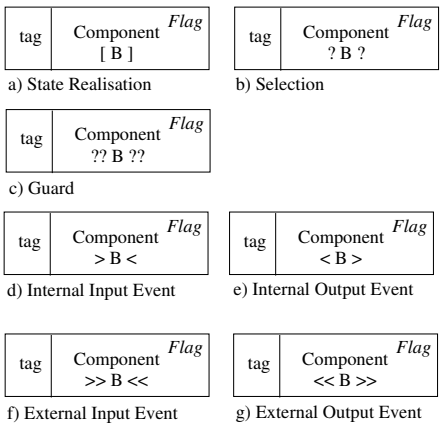

Figure 1. BT node types

2.1. The Behavior Tree Notation

The Behavior Tree (BT) notation [8] 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 done 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. Semantically, this merging step is based on the fact that the matching node provides 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 [8, 17].

The syntax of the BT notation comprises nodes and edges. Each node is one of the types in Figure 1, and refers to a particular *component*, C, and a *behaviour*, B. In addition, each node can be labelled by one or more *flags*.

As shown in Figure 1 a node type can be

(a) a state realisation. If B is a state name, this models C realising (entering) state B. For example, the root node of Figure 2 says initially the $Torch$ component is in state *free*. Alternatively, B can be of the form $attr := e$, where $attr$ is an attribute of component C and e is some expression, modelling C's attribute being updated.

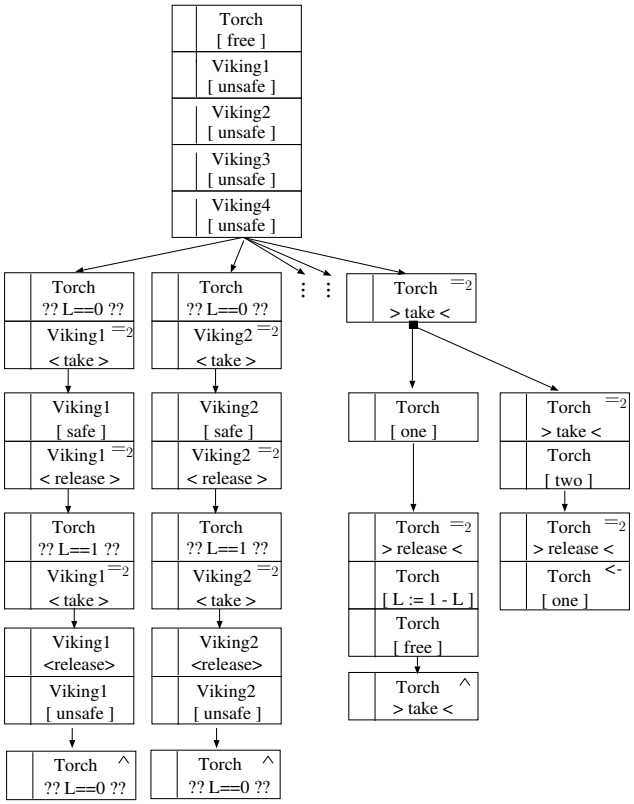

Figure 2. Example: Four vikings over bridge

(b) a selection (or condition) on C's state if B is a state name, or on one of its attributes if B is an expression over the attribute; the control flow terminates if the condition evaluates to false,

(c) a guard, where B may be of the same form as for a selection (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 internal 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 asynchronous,

(f-g) an external event modelling communication and data flow with the environment of the system, where B specifies an event; the control flow can pass the external event node when the event occurs (the message is sent), otherwise it is blocked and waits; the communication is asynchronous.

The tree-like form of Behavior Trees (BTs) allows the user to represent the control flow of the system by either a

normal edge (for sequential flow) or a branching edge (for concurrent or alternative flow). Figure 2 illustrates this by means of an example (taken from the UPPAAL distribution): four vikings are about to cross a damaged bridge at night which can hold only the weight of two people at a time. They have to take a torch before crossing the bridge and after reaching the safe side of the bridge they release the torch. A viking may only claim the torch if they are on the same side of the bridge. A viking might decide to come back from the safe to the unsafe side of the bridge in order to guide other vikings with the torch (to save space the figure leaves out two of the viking threads using "..." as a placeholder; these threads are similar to those given).

In Figure 2 the four viking threads act in parallel with the torch thread. Taking and releasing of the torch is synchronised via internal message passing. We use binary synchronisation in this example to model that only one viking at a time can take or release the torch. The position of the torch is modelled by variable L, for simplicity it is either 0 or 1. The torch might be taken by either one or two vikings. An alternative branching in the torch thread (marked by the black box) indicates this. Initially, the vikings are in an *unsafe* state and the torch is *free*. If nodes are grouped together (i.e., without an edge between them), like in the initialisation step, they model an atomic step (which cannot be interrupted). The model in Figure 2 uses the simple BT notation and thus timing behaviour is not modelled yet. We will extend this model in Section 3.

A flag in BT node can specify (a) a reversion $\hat{}$ in case the node is a leaf node, indicating that the control flow loops back to the matching node (i.e., a node with same component name, type and behaviour), (b) a macro node $\sim$, indicating that the flow continues from the matching node, (c) killing of a thread $--$, which kills the thread that starts with the matching node, or (d) a synchronisation point $=$, where the control flow waits until all other threads with a matching synchronisation point have reached the synchronisation point. We also introduce a flag for *binary synchronisation*, $=_2$, which models that exactly two matching nodes may participate in the synchronisation. This is useful in combination with input and output events when modelling a binary channel. In this case the matching nodes are one input and one corresponding output node. Each node has also a *tag* which allows the user to relate the BT nodes to the original requirement specification. This tag is omitted in the example in Figure 2.

2.2. Timed Automata

Timed automata is a formalism to model and verify real-time systems. Originally, timed automata were defined as finite-state Büchi automata extended with a set of real-timed

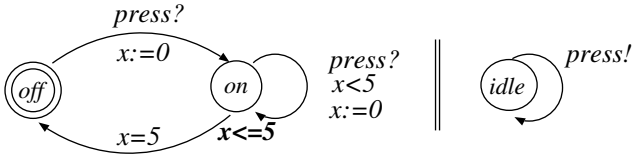

Figure 3. A simple lamp as timed automaton

variables to model clocks [2]. Clock constraints on the transitions enable the user to restrict the behaviour of the automaton. Progress properties are enforced by Büchi acceptance conditions. *Timed Safety Automata* are a simplified version of timed automata in that progress properties are specified using *location invariants*. These simpler automata build the basis of several verification tools, amongst them the model checker UPPAAL [4]. In this paper we refer to these automata as timed automata (as is often done in the literature). For an introduction to timed automata we recommend the work by Bengtsson and Wang [5].

Figure 3 shows a simple example with two timed automata modelling a lamp and a user operating the lamp by pressing a button. The lamp is bound by timing constraints in that it switches itself off after some time. Both automata work in parallel and communicate via a synchronisation channel *press*. To model the timing constraints we introduce a clock x. Initially the lamp is *on* and the user is *idle*. If the user sends a *press* the lamp is switched *on* and the clock is reset to zero. The lamp can stay in location *on* as long as the clock satisfies the given *location invariant*, $x \leq 5$. If the user presses the button again within 5 time units the clock is reset and the light stays on. When 5 time units have elapsed, i.e., $x = 5$, the automaton must leave the location *on* and takes a transition to its initial location *off*.

Formally, guards and location invariants in a timed automaton range over *clock constraints* of the form $x \sim n$ or $x - y \sim n$ where x and y are clocks, $n \in \mathcal{N}$, and $\sim$ can be $<, \leq, =, \geq$, or $>$. In UPPAAL, location invariants are restricted to be "downward closed". That is, the invariant has to be of the form $x < n$ or $x \leq n$. A location can also be marked as *urgent* or *committed* enforcing the next transition to be taken without delay. An urgent location can be interleaved with other states from parallel automata. Whereas a committed location allows interleaving only with other committed locations.

In the following, C denotes a set of clocks and $\mathcal{B}(C)$ a set of clock constraints. The powerset over C, 2^C, models a set of clock resets. Σ is a set of actions comprising variable updates and sending and receiving of synchronisation events.

A timed automaton is defined as a tuple $\langle N, l_0, E, I \rangle$

where

- N is a finite set of nodes
- $l_0 \in N$ is the initial node
- $E \subseteq N \times \mathcal{B}(\mathcal{C}) \times \Sigma \times 2^C \times N$ is the set of edges and
- $I : N \rightarrow \mathcal{B}(\mathcal{C})$ describing location invariants.

The notation $l \overset{g,a,r}{\rightarrow} l'$ is used if $(l, g, a, r, l') \in E$.

An operational semantics of timed automata is defined as a timed transition system. A state is given as a pair $\langle l, u \rangle$ of location l and clock assignment u. The transitions are defined by two rules:

- $\langle l, u \rangle \overset{d}{\rightarrow} \langle l, u + d \rangle$ if the clock assignment before and after the transition (u and $u + d$) both satisfy the location invariant for l, $I(l)$;
- $\langle l, u \rangle \overset{a}{\rightarrow} \langle l', u' \rangle$ if there exists a transition from l to l' with action a such that u satisfies the guard of this transition and u' satisfies its clock resets.

The UPPAAL tool allows the user to label a transition with any number of actions, but at most one of them can be a synchronisation event. That is, if $\Sigma^s \subseteq \Sigma$ is the set of synchronisation events, then in the above definition a set of actions $a \subseteq \Sigma$ must satisfy $\#(a \cap \Sigma^s) \leq 1$.

3. Timed Behavior Trees

Our extension of BTs with a notion of time is based on the concepts used in timed automata [5]. Timed automata provide a well developed theory and also tool support, e.g., model checking [4, 6].

A timed BT model is equipped with a number of clocks which evaluate to a real number. All clocks progress simultaneously. A clock can be reset to zero or can constitute a guard on a transition or an invariant on a location.

Nodes in a Behavior Tree describe transitions from one location to the next as they describe a state change, a guard or message passing. Correspondingly, locations are located between the nodes in a BT. To introduce the notion of timed behaviour we extend ordinary BT nodes with three additional slots: a *guard* **G** over clock values, a *reset* **R** of clocks, an *invariant* **I** over clocks.

The timed BT node in Figure 4 on the left corresponds to a timed automaton fragment of the form shown in Figure 4 on the right. Let S_0 and S_1 be the pre- and post-location of the BT node in the figure. The node's (un-timed) behaviour, *component_behaviour*, relates to action $a \in \Sigma$ (e.g. an update, a guard or a synchronisation) of the transition between location S_0 and S_1 in the timed automaton. Guard **G** and

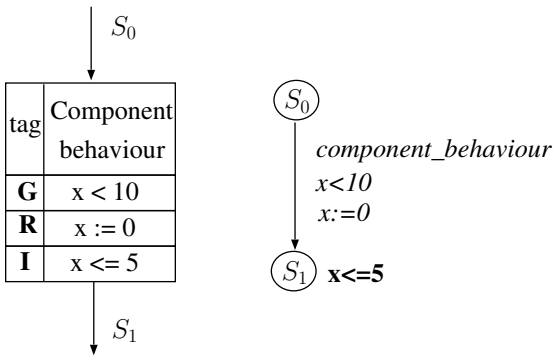

Figure 4. Timed BT node (left) and corresponding timed automaton (right)

reset **R** also label the transition modelling the timing constraints. The invariant **I** provides the location invariant of the location the transition leads to, i.e., S_1.

We use this notation to extend our model of the four vikings crossing a bridge (as given in Figure 2). The vikings take 5, 10, 20 and 25 time units, respectively, to reach the other side. The problem is to find the shortest amount of time required for all vikings to reach the safe side of the bridge. The obvious solution is to use a greedy strategy, where the fastest viking (5 time units) guides all other vikings over the bridge and always brings the torch back. In total, this would sum up to 65 time units (10 +20+25 +2*5 time units). However, it is also possible that the four vikings will cross the bridge in 60 time units as we will prove in section 4.

We introduce the timing constraints into our model utilising timed BTs. Figure 5 shows two threads: the thread of the fastest viking (all other vikings are modelled similarly with different timing constraints) and the thread of the torch process.

The viking behaves as follows: if the torch is on the right side of the river (torch is unsafe, $L = 0$) and the torch synchronises on the event *take* then the process resets its clock y and transits to the next state. After 5 time units (as specified in the guard of the second atomic block of timed BT nodes) the viking transits to a state where he is safe and releases the torch when possible. As a result the variable L which represent the position of the torch will be inverted. From here the viking might move back to the unsafe side taking and releasing the torch within another 5 time units.

The torch process (see Figure 5 on the right) is extended with clock x in order to enforce that no time is spent when the torch is taken (by either one or two vikings). We model a reset on clock variable x in the first timed BT node and enforce an invariant on the location that is reached after the first transition. Otherwise the timing of the torch is unrestricted.

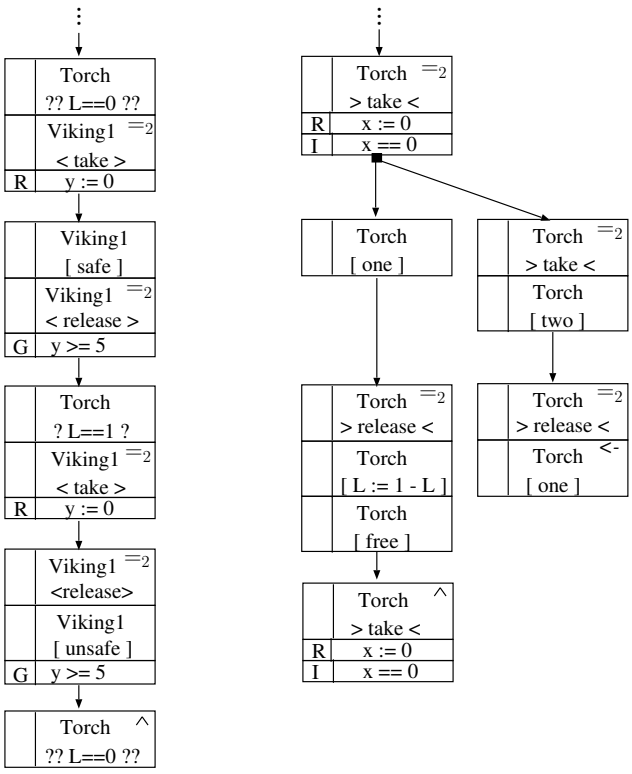

Figure 5. Viking and torch with timing constraints

3.1. Semantics

The semantics of timed BTs can be defined through a mapping to timed automata. We show for each language concept of timed BTs an equivalent automaton or a network of automata. This mapping builds the basis of the interface from timed BTs to the UPPAAL tool which allows us to model check given timing requirements.

Generally, a timed BT without parallel threads can be envisaged as a single timed automaton if we explicitly insert edges for reversions and macros. We add an initial location that we call *disabled*. From this location the initial BT node provides an edge to a *start* location which models the timed BT after initialisation. The leaf nodes in the BT lead from a *last* location back to either the *disabled* state if the tree terminates (i.e., has no reversion flag) or to a location where a reversion points to. If a timed BT has parallel threads, we model each thread as an individual timed automaton, operating in parallel.

Node Types. The different BT node types (as introduced in Section 2) translate into actions $a \in \Sigma$ that label the transitions of a timed automaton. Component name and behaviour constitute the variable name. (Further on we will refer to both simply as *behaviour* of a BT node.)

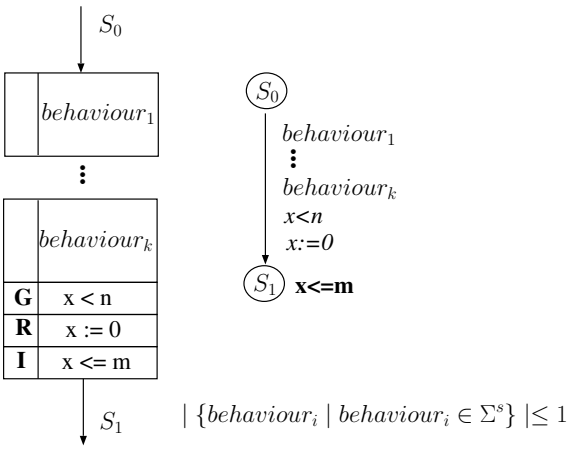

Figure 6. Mapping of atomic block with at most one synchronisation

- a state realisation maps onto an update $a \in \Sigma$ where the component's name becomes a variable name;
- a selection becomes a guard; it also produces an alternative transition (from the same pre-state) labelled with the negation of the selection and leading to a location *disabled* (each process has such a location where it stays until it gets "invoked");
- a guard maps to a timed automaton guard;
- input and output events are mapped to *communication variables* that can be *set* and *unset*; i.e., an output event is mapped to an update that *sets* the corresponding variable and an input event is mapped to a guard that is satisfied if the communication variable is set and an update to *unset* the variable; in the case of external communication the environment is modelled as a parallel process, setting and unsetting the communication variable;
- a binary synchronisation is mapped into a synchronisation event in timed automata; a synchronisation with n threads involved is mapped using boolean variables $synch_1, \ldots, synch_n$. Each synchronisation partner i that reaches the synchronisation point sets its variable $synch_i$ to true; the transition from the next location (i.e., the next step) is labelled with a guard $synch_1 \wedge \ldots \wedge synch_n$ and updates $synch_i$ to false.

Sequential Flow. Figure 4 shows the mapping of a single node into a single transition. A sequence of timed BT nodes maps into a sequence of transitions in similar fashion.

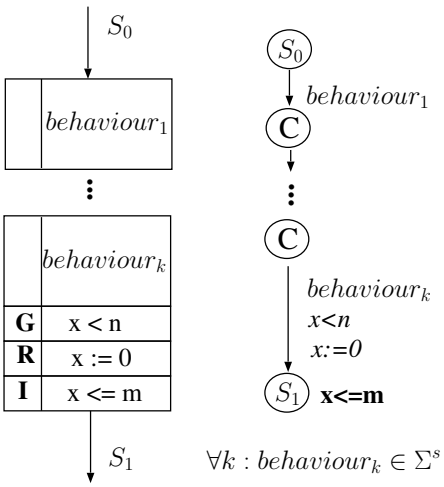

Figure 7. Mapping of atomic block with more than one synchronisation

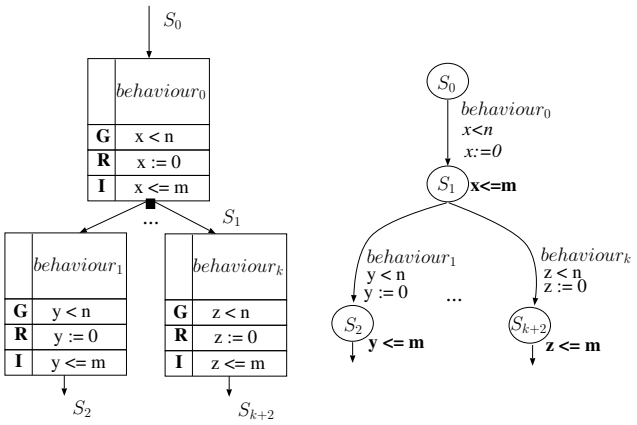

Figure 8. Mapping of alternative branching

Atomic block. If two or more timed BT node are grouped together in an atomic block we have to capture that the block cannot be interrupted. In most cases, we can utilise the fact that transitions in a timed automaton can be labelled with guards and a set of actions containing at most one synchronisation. If the atomic block satisfies this restriction it can be mapped into one transition as shown in Figure 6.

Atomic blocks with more than one synchronisation are excluded from the timed BT notation for the following reason. We could translate these block by introducing *committed* locations for each synchronisation into the timed automaton which enforce a priority to the actions of the atomic block. Figure 7 shows the idea of this translation. Since transitions to committed locations can be interleaved with transitions to other committed locations, however, this model does not provide a faithful translation. The atomicity is violated. Moreover, more than one synchronisation in an atomic block easily leads to deadlocking behaviour of the overall system and is seen as an unclear modelling style that should be avoided.

Alternative Branching. Alternative branching in a timed BT is simply mapped to a location with more than one outgoing edge, with each edge corresponding to the root node of one of the alternatives. Behaviour, guard and reset of the first timed BT node in each branch label one transition. Figure 8 depicts this notion.

Concurrent Branching. A timed BT can be mapped into a number of parallel timed automata each representing a concurrent branch in the timed BT.

At each concurrent branching point in the timed BT we introduce a new process for each branch that follows. Figure 9 depicts this mapping. The last BT node before the branching is translated into a number of transitions (in the parent process $Proc_0$) each invoking one of the new processes. The locations between these transitions are labelled as committed locations. Each (child-) process ($Proc_1, \ldots, Proc_k$) contains an initial state *disabled*, from which a single invocation transition leads to the first enabled state of the process.

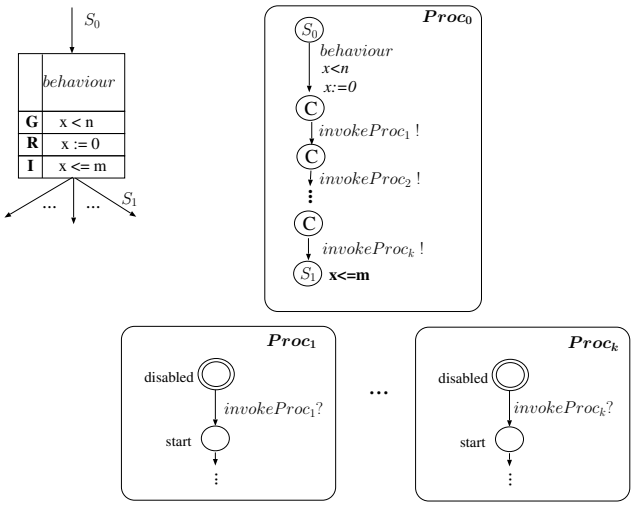

Figure 9. Mapping of branching nodes

Killing of Threads, Reversion and Macros.

Killing of a thread is realised by means of synchronisation. Each process $Proc_i$ has transitions from each location back to the location *disabled* which are labelled with the action $killProc_i?$. Since a kill event can be received at any time we have to introduce transitions from each location. Reversion and macros can be mapped in a similar fashion using transitions from the location *disabled* and the loca-

tion *last* to the location the reversion or macro is directed at.

4. Verification of Timing Properties

Based on the extension of the BT notation we are able to integrate local timing assumptions into a system model. An example of such an assumption is the minimum time in which one of the four vikings is able to cross the bridge. These assumptions can be specified as constraints over local clocks and state invariants. Consequently, they provide an upper or lower bound for the execution of one specific local action. The benefit of integrating these local timing assumptions into the system model is the ability to reason about global timing properties with a timed model checker [1, 3, 4, 6]. Since timed BTs are based on timed automata, we can utilise UPPAAL as our timed model checker. The objective of this section is to describe the basic ideas behind the verification of global timing properties with timed model checking. Consequently, we describe first how to specify these global timing properties with real-time temporal logical formulae and afterwards we illustrate how to enrich the model with the required local timing assumptions. Finally, we present our tool support for the complete verification process.

4.1. Specification of Real-Time Properties

To specify system-level real-time properties, real-time temporal logics such as metric temporal logic (MTL) [12] and timed computational tree logic (TCTL) [1] are used [3]. These real-time temporal logics extend standard temporal logics with operators that allow reasoning over discrete (such as $\mathbb{N}^+$) or continuous/dense (such as $\mathbb{R}^+$) time domains [11]. One interesting property for the viking-bridge problem can be formulated by the following temporal logical formula:

$$\exists \Diamond_{\leq 60}(\mathit{Viking1} = \mathit{safe} \wedge \mathit{Viking2} = \mathit{safe} \wedge \mathit{Viking3} = \mathit{safe} \wedge \mathit{Viking4} = \mathit{safe})$$

This formula queries whether there exists a trace of the system where the system gets into a state where all four vikings are safe and the total time is less than or equal to 60 time units, i.e., can all four vikings cross the bridge in 60 or less time units. The query notation is rather cumbersome, so to make it easier for non-experts to specify TCTL queries we propose using the real-time specification patterns of Konrad and Cheng [11] which extends the pattern system of Dwyer et al. [9] toward the formalisation of critical system properties with real time temporal logics. The idea behind these specification patterns is (a) to create a repository and a classification scheme of commonly specified properties in terms of a specific (real-time) temporal logic and (b) to provide a structured natural language grammar. Consequently, these (real-time) specification patterns provide templates for the specification of system properties and thus guide requirements analysts in expressing (real-time) system requirements directly in a (real-time) temporal logical formula.

4.2. Integration of Local Timing Assumptions

When the requirements or high level design decision require a timing constraint on a particular local behaviour, we introduce a new local clock and place the constraint as a guard on the new clock, using the syntax explained in Section 3. This will typically also require the local clock to be reset at an earlier node in the tree. The exact point at which to reset the clock must be determined from the requirement. For instance, in Figure 5, we know that at least 5 time units must have passed between $\mathit{Viking1}$ taking and releasing the torch, hence we initialise the new local clock y to 0 when the torch is taken, and provide the lower bound on y when it is released. If the requirements specify a timing constraint on the state of the system, this is typically expressed using a location invariant. For example, in Figure 5, we ensure that the action of taking a torch takes no time by constraining the local clock x to satisfy the invariant $x == 0$ when the take action is complete. In real-time systems it is often the case that not every behaviour is given a specified timing constraint. In some cases this can lead to unrealistic behaviour, where certain actions appear to take no time at all, whereas in reality there is some non-zero lower bound on the time the behaviour must take (e.g., the pressing of a button). In such circumstances, assumptions must be made about the length of time required for the relevant behaviours and the constraints must be added to the model as above; of course, such assumptions must be thoroughly documented.

4.3. Tool Support

To validate the feasibility of the presented concepts, the software package Integrare [15] has been extended to allow the specification of timed BTs, i.e., guards, invariants and clock resets may be associated with each node. To enable the verification of global timing properties, the extended version of the tool contains the facility to generate UPPAAL code from a timed BT. The translation follows the mapping between timed BTs and timed automata as described in Section 3. To improve the visual representation of the generated timed automata, the translator also adds layout information. Fig. 10 is a screen-shot of the Integrare software package. The open document represents the timed BT of the viking example. Also shown is the UPPAAL translation dialog, displaying the XML code that can be used as

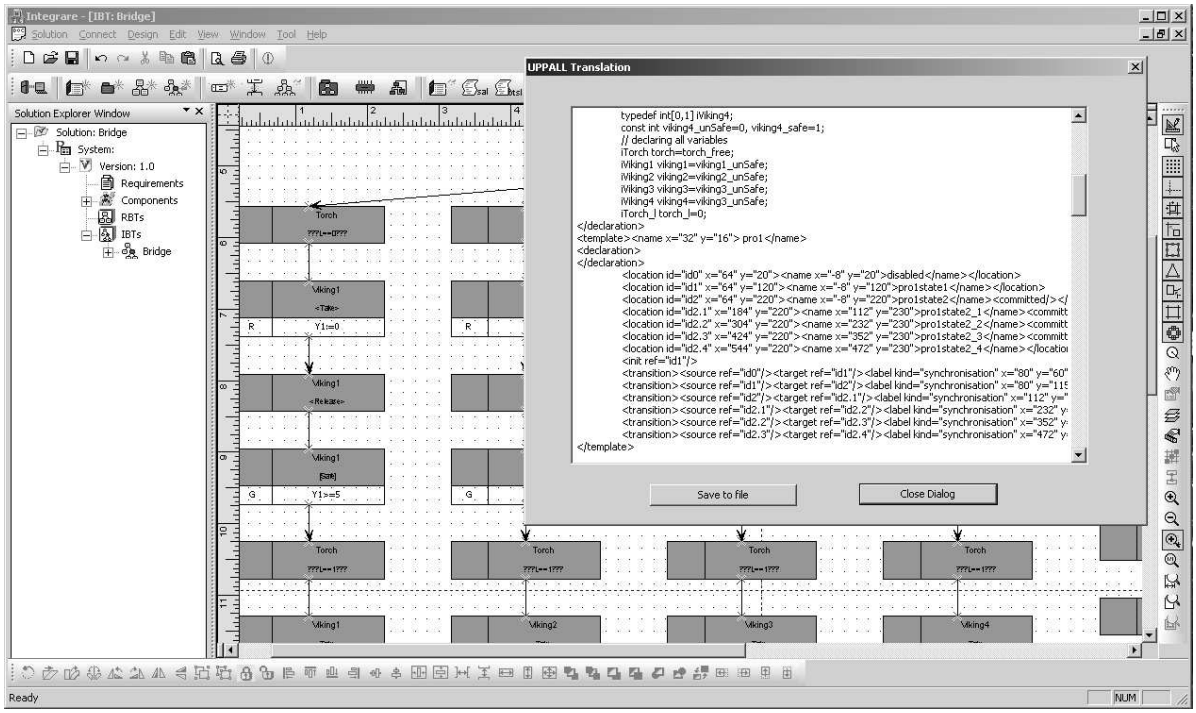

Figure 10. The tool Integrare with a timed BT model of the viking example

an input for the model checker. In Fig. 11 the XML code has been opened with UPPAAL and the timed automaton for the fastest viking is shown.

UPPAAL's query language includes state and path expressions. State expressions cover properties of a single state, e.g., $Viking1 = safe$, or $y <= 5$. Path expressions, of which UPPAAL contains five types, cover properties of execution paths, and are classified as reachability, safety, or liveness properties. For instance, the TCTL formulae given earlier, that there exists a path where all four vikings are safe and the total time is less than or equal to 60 time units, is an example of reachability and is specified as follows.

```
E<> Viking1==safe and Viking2==safe
and Viking3==safe and Viking4==safe
and globaltime<=60
```

Although UPPAAL does not allow nesting of path expressions, the five types allowed give an adequate coverage of TCTL for most real-world problems. UPPAAL can return a yes or no answer automatically for these queries, and provide either a witness (reachability) or counter-example (safety, liveness) as appropriate. By model checking our example, we were able to prove that the model is deadlock free and that the four vikings are able to cross the bridge in 60 time units. The given diagnostic trace, which can also be viewed step by step in UPPAAL's simulator, can be interpreted as follows:

- step 1: the fastest (5 time units) and the fast (10 time units) viking cross the bridge. (total time = 10 time units)
- step 2: the fastest (5 time units) viking brings the torch back to the unsafe side. (total time = 10 +5 time units)
- step 3: the slow (20 time units) and slowest (25 time units) viking cross the bridge. (total time = 10+5+25 time units)
- step 4: the fast (10 time units) viking brings the torch back to the unsafe side. (total time = 10+5+25+10 time units)
- step 5: the fastest (5 time units) and the fast (10 time units) viking cross the bridge. (total time = 10+5+25+10+10)

After executing this trace all four vikings are safe and only 60 time units have been used. Even though this is a very simple example with a relatively small state space, the benefit of timed model checking to verifying timing requirements becomes clear. The initial solution (65 time units), which was describe in Section 3 is suboptimal and by using timed model checking we could find a better solution.

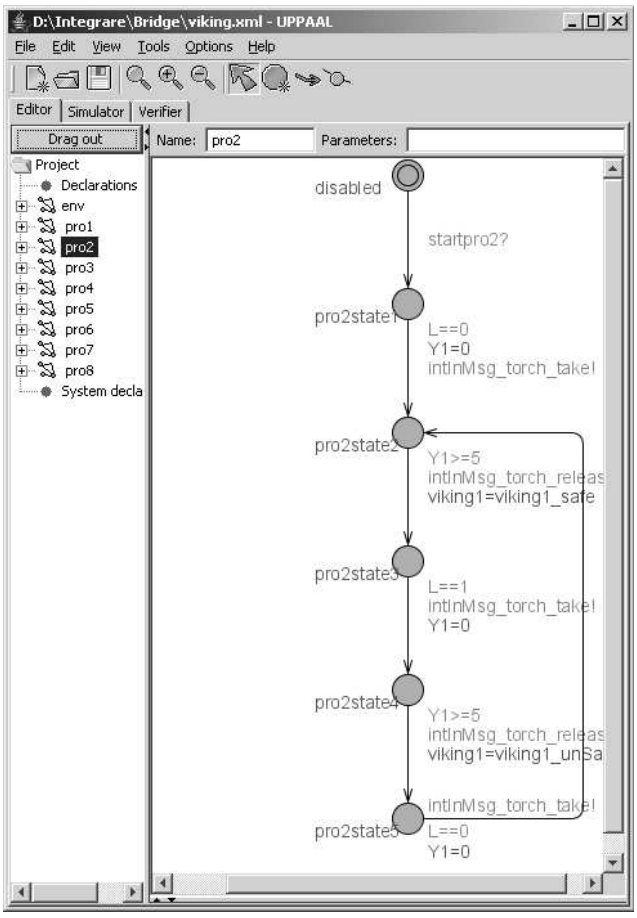

Figure 11. The tool UPPAAL with the timed automaton for one viking process

5. Related Work

Our work can be categorised under the topic of "integrating formal methods" in that we integrate the concepts of timed automata into the BT notation. In related work, other languages have been extended by a timed notation, e.g. Object-Z with the Timed Interval Calculus [16] or Object-Z with timed automata [7].

In general, an integration can be "conservative", maintaining the syntax of both notations and defining a semantical link between them, or "non-conservative", by introducing new syntax into one language to capture the concepts of the other. Since both our notations, the BT notation and timed automata, are graphical, we follow a non-conservative approach which can be done with a simple extension to the BT syntax that can still be easily read by somebody who is familiar with the BT notation. The semantics of the timed notation is mapped onto one of the formalisms, namely the timed automata semantics. We gain the benefit of an interface to the tool support for timed automata (here we use UPPAAL). Examples of other "non-conservative" approaches are timed CSP [14] and timed Petri Nets [13]. An example of a "conservative" approach to extend a languages by a timed notation is the integration of Object-Z and timed automata as given by Dong et al. [7].

6. Conclusion and Future Work

In this paper we have extended the requirements formalisation notation Behaviour Trees with timing constraints, allowing the notation to concisely express real-time systems. The extensions are based on timed automata [2, 5], since this is a well-established and elegant formalism for capturing timing constraints. This also allows us to translate a Behaviour Tree representation of a problem into a timed automata representation, which we can then simulate and model check using the tool UPPAAL [4]. A timed automata model in UPPAAL may be checked against safety or liveness properties expressed in a subset of TCTL [1], incorporating constraints on global or local (process) time.

We have thus provided Behaviour Trees with a straightforward extension of timing constraints, based on an expressive formalism for real-time systems which is supported by a mature verification tool. We have demonstrated the timing extensions and the translation process on a system that incorporates standard features of the Behaviour Tree notation in combination with constraints on both global and local time, and used UPPAAL to automatically check the system satisfies several timing constraints as well as graphically simulate the correct traces.

In future work, we will apply the extended notation to larger real-time examples, in particular a system which controls the raising and lowering of a large metal press. This system contains many potentially hazardous situations; using Timed Behaviour Trees, which provides both lower and upper bounds on global and local clocks, we can develop a comprehensive Failure Mode Effect Analysis (FMEA) of the system, extending earlier work in this area [10].

Additionally, we are working towards a probabilistic extension of BTs which would allow us to use probabilistic model-checking for the above described FMEA process. Based on this probabilistic extension random failures could be modelled and analysed conveniently.

Acknowledgements: This work was produced with the assistance of funding from the Australian Research Council (ARC) under the ARC Centres of Excellence program within the ARC Center of Complex Systems (ACCS). The authors wish to thank their colleagues in the Dependable Complex Computer-based Systems project and also the anonymous reviewers for their constructive suggestions.

References

[1] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 414–425. IEEE Computer Society Press, 4–7 June 1990.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In *Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 74–106. Springer-Verlag, 1992.

[4] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, (SFM-RT 2004)*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[5] J. Bengtsson and Y. Wang. Timed automata: Semantics, algorithms and tools. In W. Reisig and G. Rozenberg, editors, *Lecture Notes on Concurrency and Petri Nets*, volume 3098 of *LNCS*. Springer-Verlag, 2004.

[6] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control*, pages 208–219. Springer-Verlag, 1996.

[7] J. S. Dong, R. Duke, and P. Hao. Integrating Object-Z with timed automata. In *Int. Conference on Engineering of Complex Computer Systems (ICECCS 2005)*, pages 488– 497. IEEE Computer Society, 2005.

[8] R. G. Dromey. From requirements to design: Formalizing the key steps. In *Int. Conference on Software Engineering and Formal Methods (SEFM 2003)*, pages 2–13. IEEE Computer Society, 2003.

[9] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. 21st International Conference on Software Engineering*, pages 411–420. IEEE Computer Society Press, ACM Press, 1999.

[10] L. Grunske, P. Lindsay, N. Yatapanage, and K. Winter. An automated failure mode and effect analysis based on high-level design specification with Behavior Trees. In J. Romijn, G. Smith, and J. van de Pol, editors, *Proc. of Int. Conference on Integrated Formal Methods, (IFM 2005)*, volume 3771 of *LNCS*, pages 129–149. Springer, 2005.

[11] S. Konrad and B. H. C. Cheng. Real-time specification patterns. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 372–381. ACM, 2005.

[12] R. Koymans. Specifying Real-Time Properties with Metric Temporal Logic. *Real-Time Systems*, 2:255–299, 1990.

[13] C. Ramchandani. *Analysis of asynchronous concurrent systems by timed Petri nets*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1974. Project MAC Report MAC-TR-120.

[14] S. Schneider. An operational semantics for timed CSP. *Information and Computation*, 116(2):193–213, 1 Feb. 1995.

[15] C. Smith, K. Winter, I. Hayes, G. Dromey, P. Lindsay, and D. Carrington. An environment for building a system out of its requirements. In *Int. Conference on Automated Software Engineering (ASE 2004)*, pages 398–399. IEEE Computer Society, 2004.

[16] G. Smith and I. Hayes. An introduction to Real-Time Object-Z. *Formal Aspects of Computing*, 13(2):128–141, 2002.

[17] L. Wen and R. G. Dromey. From requirements change to design change: A formal path. In *Int. Conference on Software Engineering and Formal Methods (SEFM 2004)*, pages 104–113. IEEE Computer Society, 2004.

[18] K. Winter. Formalising Behaviour Trees with CSP. In E. Boiten, J. Derrick, and G. Smith, editors, *4th Int. Conf. on Integrated Formal Methods (IFM 2004)*, volume 2999 of *LNCS*, pages 148–167. Springer-Verlag, 2004.