ARC CENTRE FOR
COMPLEX SYSTEMS

Technical Report

ACCS-TR-07-01

A Semantics for Behavior Trees

Robert Colvin and Ian J. Hayes

April 2007

ARC Centre for Complex Systems
School of ITEE, The University of Queensland
St Lucia Qld 4072 Australia
T +61 7 3365 1003
F +61 7 3365 1533
E admin@accs.edu.au
W www.accs.edu.au

A Semantics for Behavior Trees

Robert Colvin and Ian J. Hayes

10:46 A.M., Wednesday 4th April 2007

Abstract

The *Behavior Tree* notation is used as part of a framework for developing complex computer systems. The framework is designed to simplify the process of constructing a formal specification of a system from its informal functional requirements. To give a meaning to Behavior Trees, this paper describes a lower-level language called *Behavior Tree Process Algebra* (BTPA) and its operational semantics, and defines a mechanical translation of Behaviour Trees into BTPA. The process algebra provides several methods by which processes may communicate with each other and interact with the environment: CSP-like synchronisation; send/receive message passing; and shared variables. The meaning of a BTPA process is defined with respect to the current state of the system (value of the components) and the active processes.

1 Introduction

Obtaining a set of requirements which is complete and consistent is one of the crucial steps in implementing a large software system. To help software engineers communicate effectively with their clients about their requirements, a common language is needed which is both easy to understand and work with, and which also has a precise meaning. The *Behavior Tree* program development framework developed by Dromey [Dro06, Dro03] is intended to fit these criteria by allowing rapid translation of informal requirements into individual Behavior Trees, in a way which is traceable and suitable for validation by clients without knowledge of formal languages. In this paper we give a formal semantics for Behavior Trees, providing an unambiguous reference for their meaning and making them amenable to tool support such as simulation and model checking.

A Behavior Tree system is formed from a set of components with state, and the behaviour of the system is described by a Behavior Tree. The notation includes constructs for message passing (events) and synchronisation, as well as testing and updating the state of components. To give a meaning to this core functionality of Behavior Trees, we describe a lower-level language called *Behavior Tree Process Algebra* (BTPA), for which we provide an operational semantics, and define a mechanical translation of Behaviour Trees into BTPA. An advantage of using a lower-level language to describe Behavior Trees is that the graphical notation, which is designed to be user-friendly, can be adapted or extended without directly affecting the underlying semantics – all that is required is a translation of the new notation into BTPA. The challenge is to define a flexible core language which can express the constructs of the Behavior Tree notation. For the purposes of providing quick feedback to the behaviour modeller, we also desire the behaviour of a system to be easily simulated via a tool.

The paper is structured as follows. In Sect. 2 we give a brief introduction to the Behavior Tree notation. In Sect. 3 we present the BTPA language, into which we map the Behavior Tree constructs, before defining a semantics for BTPA in Sect. 4.

1.1 Related work

The most widely known framework for developing a program from its requirements is UML [RJB98]. However the notation lacks a precise semantics and can become cumbersome due to its large variety of diagrams (some of which have been given a semantics, e.g., Activity Diagrams have been given an encoding as petri-nets [ED03]). It has been argued that UML's advantage lies in being graphical and requiring little specialist knowledge to understand. In contrast, a specification language such as Z [Spi92] has a fully formal semantics and mature methods for correct program development, but is harder to validate against user requirements since expert knowledge is required to understand Z specifications. Behavior Trees are intended to provide the benefits of a simple graphical notation, especially user validation, but also have a straightforward and precise semantics that supports simulation and formal verification via model checking.

The meaning of a Behavior Tree is process-based, and a large range of languages for expressing the behaviour of concurrent processes have been developed; some of the better known include CSP [Hoa85], CCS [Mil82], Petri Nets [Pet81], Action Systems [BKS88], Unity [CM88], and State charts [HN96]. The differences between each typically lie in their method or methods of *interprocess communication* (IPC). In his book on network programming for Unix, Stevens [Ste99] identifies three methods of IPC: synchronisation, message passing, and shared variable[1]. Using this implementation-based classification, CSP has synchronisation (via its actions) as well as message passing (using channels). In its original form, CSP does not have shared variable communication, though it has been extended to include a notion of state by being combined with Z (Circus, [WC02]). In contrast, IPC in Action Systems is based on shared variables, with no primitive notion of synchronisation; CSP and Action Systems have, however, been combined by Butler [But92]. Statecharts [Har87] allow both synchronised and shared variable communication, though there is not a single source for their precise semantics; the most authoritative appears to be given in [HN96], which describes the semantics (in natural language) in terms of an execution tool.

There are three features of the Behavior Tree language which distinguish it from most other languages. Firstly, it allows individual processes to have access to the full context in which it is executing, including concurrently executing threads. This allows a process to "kill" another process and all of its subprocesses (similar to CSP's *interrupt* operator). It also allows new processes to be spawned during execution (as is allowed by the π-calculus [Mil99]). Secondly, the language includes atomic composition, which creates a more expressive language, and allows small, simple statements to be built into a single atomic action. And thirdly, the languge includes a type of message passing IPC where the sender of the message is not blocked if there are no receivers. This can be used to model a broadcast message which is received by a dynamically changing number of consumers; this system for IPC is implemented by the Elvin messaging system [SAB$^+$00] produced by Mantara Software [Man]. This approach is also suited to modelling incoming information from the environment, which does not "synchronise" with the system being specified but must be captured at the correct moment: the system must be ready to receive the information when it happens or the event will be missed. The Behavior Tree language also allows synchronised and shared variable IPC: the model for synchronisation is based on that of CSP and its alphabets, and shared variable communication is modelled in the usual way, with processes able to test and update the state.

Each of these three differences stems from the Behavior Tree notation being developed for systematic requirements capture. While concepts such as CSP's interrupts and checkpoints are elegant formalisations, they do not commonly appear in client-produced natural language requirements. Similarly, requirements will not typically assume a synchronisation between the system and its environment, but instead specify that the system must be ready to respond to an event when it occurs.

2 Behavior Trees

In this section we provide a very brief and informal description of Behavior Trees; for more detail, see, e.g., [Dro06, Dro03, SWH$^+$04]. Common Behavior Tree nodes are given in Fig. 2, and the constructors are described in Fig. 3. Some variable-naming conventions are described in Fig. 1. The notation is used for a small example given in Appendix C. For ease of presentation we write Behavior Trees from left-to-right, wrapping across lines and with the root node in the top left, though usually Behavior Trees are written top-down with the root node centre-top. When convenient, to save space we use an equivalent textual notation.

The nodes of a Behavior Tree are like statements in a programming language, and include behaviour such as testing and updating of values (guards and state realisations), receiving and sending messages, and synchronising. An iterative control structure can be written using reversion nodes, and conditionals can be written

[1] Stevens also identifies a fourth method of IPC, remote procedure call. This can be regarded as a special case of message passing – the distinction is not important at our level of abstraction.

N	Behavior Tree nodes
T, T_i	Behavior Trees
π, S, TT	Multisets of Behavior Trees
C, C_i, x	Variables (including components)
s, v	Values
$\vec{x}$, $\vec{v}$	Lists of variables, values
$\mathcal{D}$	Contexts
σ	States (of the system)
e, m	Events, messages

Figure 1: Variable naming conventions

Node	Box notation	Text notation	Description
Basic nodes			
State Realisation	$\boxed{\begin{array}{c}C\\ {[s]}\end{array}}$	$C[s]$	Component C “realises” (is assigned) state (value) s.
Guard	$\boxed{\begin{array}{c}C\\ ???s???\end{array}}$	C ??? s ???	Blocks until component C is in state s.
Output event	$\boxed{\begin{array}{c}C\\ < e(\vec{v}) >\end{array}}$	$C < e(\vec{v}) >$	Component C outputs (generates) event e with values in the list $\vec{v}$.
Input Event	$\boxed{\begin{array}{c}C\\ > e(\vec{x}) <\end{array}}$	$C > e(\vec{x}) <$	Blocks until component C receives event e, storing the values passed into the variables in $\vec{x}$.
Other nodes			
Goto	$\boxed{N^{=}}$	$N^{=}$	Behave like the tree rooted at node N. A goto node will typically be a leaf node.
Process kill	$\boxed{N^{--}}$	N^{--}	Kill any behaviour associated with the tree rooted at node N.
Reversion	$\boxed{N^{\hat{}}}$	$N^{\hat{}}$	This is a way of repeating behaviour. Behave like closest ancestor node N, in addition killing all behaviour begun at or below the destination reversion node.
Synchronise	$\boxed{N^{@_m}}$	$N@_m$	Participate in synchronisation event m, and execute N. Each node $N@_m$ is blocked until all other nodes participating in m are ready. When ambiguity is not possible, the tag m may be omitted.

Figure 2: Common Behavior Tree nodes

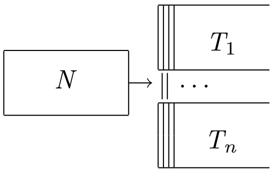
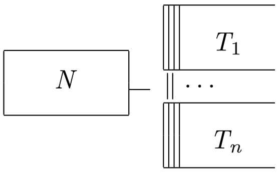
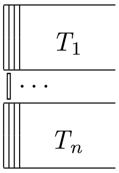
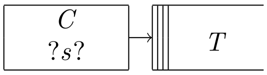
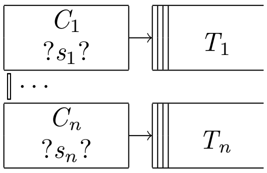
Type	Box notation	Text notation	Description
Sequential composition		$N;\ [\![T_1, .., T_n]\!]$	Execute N, followed by the trees $T_1..T_n$. Other processes operating in parallel may have their behaviour interleaved between N and $T_1..T_n$.
Atomic composition		$N;;[\![T_1, .., T_n]\!]$	Execute N, followed by the trees $T_1..T_n$. Other processes operating in parallel may not have their behaviour interleaved between N and $T_1..T_n$.
Nondeterministic composition		$T_1 [\!] .. [\!] T_n$	Execute one branch from a multiset of possibilities. Execution of a nondeterministic composition blocks until one of the processes can take a step.
Special trees			
Selection		$C\,?\,s\,?;\ T$	Behaves as tree T if component C is in state s. If component C is not in state s, the entire process terminates (never executes).
Multiple selection	 C_1 $?s_1?$ T_1 $[\!]\cdots$ C_n $?s_n?$ T_n	$C_1\,?\,s_1\,?;\ T_1$ $[\!]\cdots$ $C_n\,?\,s_n\,?;\ T_n$	Nondeterministically choose a T_i for which $C_i = s_i$. Those T_i which are not chosen terminate (never execute).

Figure 3: Behavior Tree constructors

by combining non-deterministic choice with guards. The notation also includes a “goto” node, which is used as a shorthand when two trees behave identically.

A Behavior Tree is one of the three forms in Fig. 3:

1. a node sequentially composed with a multiset[2] of trees;
2. a node atomically composed with a multiset of trees; or
3. a nondeterministic composition of trees.

A leaf node is represented by a node sequentially composed with an empty multiset of trees. Typical sequential execution is achieved when a node is sequentially composed with a singleton multiset of processes. Parallelism is introduced when a node is sequentially composed with a multiset of two or more trees; each tree represents a new process which is ready for execution. Atomic composition also involves a multiset of processes, though typically it will have exactly one element in it.

In addition, the special *selection* syntax $C\,?\,s\,?$ may be used to model a tree that is guarded by the condition $C = s$, except that if $C = s$ does not hold the tree immediately terminates (instead of blocking).

The node types goto, kill, and reversion, all assume that there exists a unique destination node N elsewhere in the tree. If this is not the case, the behaviour of these nodes is undefined. Such undefinedness can be easily detected by static analysis.

Summary. The Behavior Tree notation is designed for translating informal requirements, and hence includes nodes for manipulating the state and expressing the different types of communication that may appear in computer systems. In the next section we provide a more general core language which encompasses the Behavior Tree notation, and provide a straightforward translation from Behavior Trees into the core language.

3 The process algebra BTPA

In this section we present a process algebra, BTPA, and describe how it may be used to represent the Behavior Tree notation given in Sect. 2.

3.1 Elements of the process algebra

Variables, values and state. We assume a set *Var*, representing variables (components), and a set *Val*, representing values of variables. The state[3] is given in the usual way as a function from variables to values, $\mathit{State} \mathrel{\widehat=} \mathit{Var} \to \mathit{Val}$.

Processes. The set *Proc*, representing processes, is formed of all possible terms constructed form the operators in Fig. 4. The language constructors include those in the Behavior Tree notation (Fig. 3), and are familiar in the process algebra domain: sequential, atomic, and nondeterministic composition. We do not treat parallelism as a separate operator as with most algebras; instead, parallel behaviour is introduced when there is more than one process on the right-hand side of sequential or atomic composition. (Some consequences of this are discussed later.) The algebra also includes a nonblocking operator, ⧫, which we call *else-skip*. A tree ⧫T behaves as T if T is enabled, or terminates if T is disabled; it is used to model the Behavior Tree selection constructor in Fig. 3.

[2] A multiset, or *bag*, allows more than one instance of an element to be present. We use a multiset of trees rather than a set to allow multiple copies of the same tree to be executed concurrently.

[3] In this paper we will use the term “state” to refer to the state of the system and the term “value” to refer to the state of a single component, except when referring to the node type “state realisation”, which refers to a single component.

Constructors	
$N;\ TT$	Sequential composition
$N;;\,TT$	Atomic composition
$T_1 \talloblong T_2$	Nondeterministic composition of processes T_1 and T_2
$\blacklozenge T$	Execute T if it is enabled, otherwise terminate T.

Figure 4: BTPA constructors

For convenience, when TT is empty, i.e., N is a leaf node, we write just N, and when TT is the singleton bag $\llbracket T \rrbracket$ we write $N;\ T$ or $N;;\,T$. Nondeterministic composition may be generalised to any finite number of processes.

Contexts. A BTPA system, which we call a *context*, is made up of the current state and a multiset of (active) processes, i.e., $\mathit{Ctx} \mathrel{\widehat=} (\operatorname{bag} \mathit{Proc}) \times \mathit{State}$. For a context $\mathcal{D}$, we write $\mathcal{D}.\pi$ to refer to its active processes, and $\mathcal{D}.\sigma$ to refer to its state. In the execution of a system, each step alters the context in some way.

For convenience we define the notation $T{\cdot}S$ as the bag formed by adding element T to bag S. This and other bag operators are described in Appendix A. For compactness we "lift" the operator to contexts, so that $T{\cdot}\mathcal{D}$ is the context formed by adding the process T to the active processes of $\mathcal{D}$, i.e.,

$$T{\cdot}\mathcal{D} \mathrel{\widehat=} (T{\cdot}(\mathcal{D}.\pi), \mathcal{D}.\sigma)$$

For convenience, other multiset operators are lifted to contexts in a similar way.

Environment. In addition to the context, we maintain a static execution environment, which contains two mappings: a labelling system for (sub)processes, and a function giving the synchronisation alphabet of each tree.

Label mapping. Given a process $\mathcal{T}$, the environment includes a function $\rho\colon \mathit{BTLabel} \nrightarrow \mathit{Proc}$, which allows retrieval of the process corresponding to a given *label*. It stays constant throughout the execution of $\mathcal{T}$. The function ρ is required for defining the behaviour of reversion, kill and goto nodes; a method for constructing ρ is given in Appendix B.

Synchronisation alphabet. Synchronisation on message m occurs when all active threads that have m in their alphabet are ready to participate in m. The environment therefore includes a function $\alpha\colon \mathit{Proc} \to \mathbb{P}\,\mathit{Msg}$ which maps each process to the synchronisation events it may participate in. The function α can be populated using static analysis on the tree, and should satisfy the following healthiness conditions as introduced in CSP [Hoa85]:

$$\begin{array}{l} \alpha(N;\ T) = \alpha(T) \ \mathit{if}\ \alpha(N) \in \alpha(T) \\ \alpha(T_1 \talloblong T_2) = \alpha(T_1) \ \mathit{if}\ \alpha(T_1) = \alpha(T_2) \end{array}$$

As with CSP, parallel processes may or may not have disjoint alphabets. For a context $\mathcal{D}$, we define $\alpha(\mathcal{D})$ as the union of alphabets of all the processes in $\mathcal{D}$.

BTPA Nodes. The basic BTPA node types, which allow state tests and updates, synchronisation, and message passing, are given in Fig. 5.

We introduce a general node type *specification command* (*cmd*) that operates on contexts. A *cmd* has a "guard" predicate that must be satisfied for the command to be executed, and an "effect" relation that specifies how the context is updated.

$$[P(\mathcal{D}), Q(\mathcal{D}, \mathcal{D}')]$$

P is a predicate on contexts; Q is a relationship between contexts (pre- and post-contexts). A *cmd* $R \mathrel{\widehat=} [P(\mathcal{D}), Q(\mathcal{D}, \mathcal{D}')]$ will take effect if guard P is satisfied in the current context, and will update the context to satisfy Q. If P does not hold in the current context, R cannot execute (hence, this is a blocking semantics).

Nodes	Type
$[\mathit{Guard}, \mathit{Effect}]$	Specification command (cmd)
$R\,\mathsf{send}\,m(\vec{v})$	Send message m with values $\vec{v}$
$R\,\mathsf{recv}\,m(\vec{x})$	Wait for message m and store values into variables $\vec{x}$
$R\,\mathsf{sync}\,m$	Participate in synchronisation m

Figure 5: BTPA nodes

We allow Q to be arbitrarily complex, though in Behavior Tree notation the possibilities are restricted; we give the translations for the Behavior Tree nodes later. Because the node has access to all other active processes through $\mathcal{D}.\pi$, we can specify complex behaviour, such as, for example, blocking until some combination of other threads becomes active.

We also have three types of nodes for modelling communication, $\mathsf{send}, \mathsf{recv}$ and sync, each of which may be associated with a specification command R. The inclusion of R allows a state test or update to be associated atomically with the communication. A message $m(\vec{v})$ can be sent by $\mathsf{send}\,m(\vec{v})$, which represents a message (or channel) named m, with a possibly empty list of values $\vec{v}$. The process sending the message does not block if there are no receivers – the sender proceeds and the message is lost. We model the reception of a message m by $\mathsf{recv}\,m(\vec{x})$. Such a node blocks until m is sent via a $\mathsf{send}\,m(\vec{v})$ node, and it then stores the values sent, if any, into the variables in $\vec{x}$. A $\mathsf{recv}\,m(\vec{x})$ node will only respond if the length of $\vec{x}$ is the same as the length of $\vec{v}$. A node $\mathsf{sync}\,m$ blocks until all active processes which contain m in their alphabet are at their synchronisation point.

3.2 Translating to the underlying notation

In this section we describe how to translate a Behavior Tree $\mathcal{T}$ into a BTPA process. The translation is formed by the following (straightforward and automatable) steps:

- Each subprocess in $\mathcal{T}$ is mapped directly to its structural equivalent in BTPA, i.e., the constructors in Fig. 3 are mapped to those in Fig. 4, with the exception of selections which are explained below.
- Each node in Fig. 2 is translated to its BTPA equivalent, as given in Fig. 6 and discussed below.
- The label mapping ρ is built (see Appendix B).
- The alphabet function α is built.

3.2.1 Selection translations

Single and multiple selections are translated according to the following definitions.

Definition 1 (Selection)

$$C\,?\,s\,?;\;\; T \;\widehat{=}\; \blacklozenge(C\,???\,s\,???;\;\; T)$$

Definition 2 (Multiple selection)

$$(C_1\,?\,s_1\,?;\;\; T_1) \parallel \cdots \parallel (C_n\,?\,s_n\,?;\;\; T_n) \;\widehat{=}\; \blacklozenge((C_1\,???\,s_1\,???;\;\; T_1) \parallel \cdots \parallel (C_n\,???\,s_n\,???;\;\; T_n))$$

Note that a multiple selection is not just a nondeterministic composition of single selection statements: the else-skip operator must be lifted outside the scope of the nondeterministic choice.

3.2.2 Node translation

A BTPA equivalent for each of the nodes in Fig. 2 is given in Fig. 6.

Node	Behavior Tree	BTPA implementation
Output Event	$C < e(\vec{v}) >$	$\mathsf{send}\ e(\vec{v})$
Input Event	$C > e(\vec{x}) <$	$\mathsf{recv}\ e(\vec{x})$
Synchronisation	$N@_m$	$N\ \mathsf{sync}\ m$
Guard	$C\,???\,s\,???$	$[\mathcal{D}.\sigma(C) = s, \mathcal{D}' = \mathcal{D}]$
State realisation	$C[s]$	$[\mathit{true}, \mathcal{D}' = \mathcal{D} \oplus \{C \mapsto s\}]$
Goto	$N^{=}$	$\mathsf{spawn}\,\ell$ where the root of $\rho(\ell)$ is node N
Kill	N^{--}	$\mathsf{kill}\,\ell$ where the root of $\rho(\ell)$ is node N
Revert	$N\hat{}$	$\mathsf{revert}\,\ell$ where the root of $\rho(\ell)$ is node N

Figure 6: Defined nodes for BT translation

Output/input event nodes are translated directly to send and receive message nodes. We have omitted the name of component C from the BTPA message, though this can easily be added (calling the message, e.g., $C.e$) if important. The translation for synchronisation nodes is straightforward. Guards and state realisations are translated to specification commands. A guard $C\,???\,s\,???$ blocks until the component C is mapped to value s in the current state ($\mathcal{D}.\sigma$). It leaves the context unchanged. A state realisation $C[s]$ does not block (its guard is *true*), but it updates the current context so that the state maps C to the value s and leaves the active processes unchanged (we have lifted function override ($\oplus$) to operator on contexts: the override is applied to the state element of the context pair).

Goto, kill and revert nodes are translated into specification commands as given in Fig. 7.

Node	Definition
$\mathsf{spawn}\,\ell$	$[\mathit{true}, \mathcal{D}' = \rho(\ell)\cdot\mathcal{D}]$
$\mathsf{kill}\,\ell$	$[\mathit{true}, \mathcal{D}' = \mathit{filterk}(\ell, \mathcal{D})]$
$\mathsf{revert}\,\ell$	$[\mathit{true}, \mathcal{D}' = \rho(\ell)\cdot\mathit{filterk}(\ell, \mathcal{D})]$

Figure 7: Label-based node definitions

Each is nonblocking (the guard is *true*), and each leaves the current state unchanged. However the set of active threads is modified in some way. A $\mathsf{spawn}\,\ell$ adds a copy of the process labelled ℓ to the active set. Conversely, a $\mathsf{kill}\,\ell$ removes all processes that are children of the process rooted at ℓ. A $\mathsf{revert}\,\ell$ is a combination of both a spawn and a kill. The function *filterk* is defined below.

Definition 3 (*filterk*) *For label ℓ and context $\mathcal{D}$,*

$$\mathit{filterk}(\ell, \mathcal{D}) \mathrel{\widehat{=}} (\llbracket T : \mathcal{D}.\pi \mid T \not\preceq \rho(\ell) \rrbracket, \mathcal{D}.\sigma)$$

That is, $\mathit{filterk}(\ell, \mathcal{D})$ is the context formed by removing all processes T in $\mathcal{D}.\pi$ that are subprocesses of the process labelled ℓ. The subprocess ordering $\preceq$ is the canonical subterm ordering.

We have now described a process algebra which includes methods for three types of communication, and is based on a bag of processes operating in parallel (hence, we do not give an operator for parallel composition). We can translate all of the constructs of the Behavior Tree notation given in Sect. 2 into equivalents in BTPA, at the same time constructing the static execution environment which includes labelling and synchronisation information. In the next section we provide an operational semantics for BTPA, and hence for Behavior Trees.

4 Semantics

In this section we provide an operational semantics for BTPA.

4.1 Transition relations

The behaviour of a system is defined in terms of several different step relations, given in Fig. 8.

Rel^n	Type	Written	Description
$\Longrightarrow$	$\mathit{Ctx} \leftrightarrow \mathit{Ctx}$	$\mathcal{D} \Longrightarrow \mathcal{D}'$	Top-level behaviour; a single atomic step
$\longrightarrow$	$(\mathit{Action} \times \mathit{Proc} \times \mathit{Ctx}) \leftrightarrow \mathit{Ctx}$	$\langle T, \mathcal{D}\rangle \xrightarrow{a} \mathcal{D}'$	The effect of a process T on the context $\mathcal{D}$
$\twoheadrightarrow$	$(\mathit{Action} \times \mathrm{bag}\,\mathit{Proc} \times \mathit{Ctx}) \leftrightarrow \mathit{Ctx}$	$\langle S, \mathcal{D}\rangle \overset{a}{\twoheadrightarrow} \mathcal{D}'$	The effect of bag S on the context $\mathcal{D}$

Figure 8: Transition relations

Consider a context $\mathcal{D}_0$, formed from the pair $(\llbracket \mathcal{T} \rrbracket, \sigma_0)$, ie, there is exactly one active thread, the tree $\mathcal{T}$, and the initial values of the components are given in σ_0. The execution of this system proceeds in a series of steps in the transition relation $\Longrightarrow$, i.e., $\mathcal{D}_0 \Longrightarrow \mathcal{D}_1 \Longrightarrow \cdots \Longrightarrow \mathcal{D}_n \Longrightarrow \cdots$, where each step is atomic. If no transition is possible from $\mathcal{D}_i$, and there are still active threads (i.e., $\mathcal{D}_i.\pi \neq \llbracket\,\rrbracket$), then we have deadlock; if there are no active threads, then the tree has finished. As long as there are active threads that aren't blocked, the execution can continue, possibly forever.

Steps in $\Longrightarrow$ are constructed by the transition $\xrightarrow{a}$, which gives the effect of a single process on a context. For instance, the transition $\langle T, \mathcal{D}\rangle \xrightarrow{a} \mathcal{D}'$ states that executing process T transforms context $\mathcal{D}$ into $\mathcal{D}'$. The *action* name a indicates the type of transition; it may be a message name (with a list of parameters) if the transition is describing a communication, or the distinguished label δ if no communication is described, i.e., the step is state-based. In the rules, we decorate transitions with a if it is defined for either communication or state-based actions, with $m(\vec{v})$ if it is defined for messages only, and with δ if it is defined for state-based actions only, which we call δ-transitions.

When multiple processes combine their individual steps to form a single atomic step of the whole system, such as when synchronisation occurs, the behaviour is described in terms of the transition relation $\overset{a}{\twoheadrightarrow}$. For instance, the transition $\langle S, \mathcal{D}\rangle \overset{a}{\twoheadrightarrow} \mathcal{D}'$ states that allowing each process in bag S to execute a single atomic step transforms context $\mathcal{D}$ into $\mathcal{D}'$.

In the following sections we provide axioms for the different transition relations. The relations $\Longrightarrow$ and $\twoheadrightarrow$ are defined in terms of $\xrightarrow{a}$, which is the main relation, and each construct of the language is given a transition in $\xrightarrow{a}$.

4.2 Global steps

Formally, an observable step of a BTPA system is made in $\Longrightarrow$ by nondeterministically selecting one of the active processes and making a δ-transition. This is given by the following rule.

Axiom 1 (Global step)

$$\frac{\langle T, \mathcal{D}\rangle \xrightarrow{\delta} \mathcal{D}'}{T \cdot \mathcal{D} \Longrightarrow \mathcal{D}'}$$

We represent the allowable transitions as rules consisting of conditions above a horizontal line and an allowable transition (if the conditions hold) below the line. If there are no conditions then the line is omitted. In this case, a context which contains process T transitions to context $\mathcal{D}'$ if T transforms $\mathcal{D}$ into $\mathcal{D}'$ via the relation $\xrightarrow{\delta}$.

4.3 Combined transitions

Before describing the meaning of a transition in $\overset{a}{\twoheadrightarrow}$, we must define *enabledness*: a process T is *enabled* (not blocked) with respect to some context $\mathcal{D}$ and action a if T can transition in $\xrightarrow{a}$ from $\mathcal{D}$. This is written $\mathit{enabled}^a(T, \mathcal{D})$, and is defined formally below. The bag of enabled processes in a context $\mathcal{D}$ for action a is given by $\mathit{enabled}^a(\mathcal{D})$.

$$\begin{aligned} &\mathit{enabled}^a(T, \mathcal{D}) \mathrel{\widehat{=}} \langle T, \mathcal{D}\rangle \in (\mathrm{dom} \xrightarrow{a}) \\ &\mathit{enabled}^a(\mathcal{D}) \mathrel{\widehat{=}} [\![T \colon \mathcal{D}.\pi \mid \mathit{enabled}^a(T, \mathcal{D} - T)]\!] \end{aligned}$$

We now define transitions for $\overset{a}{\twoheadrightarrow}$. At the top level, a step in $\overset{a}{\twoheadrightarrow}$ occurs by letting each process in S take a single atomic step. This is achieved by selecting an enabled process T from S, taking a step in $\xrightarrow{a}$ with T, then removing T from the bag and "recursively" taking another step in $\overset{a}{\twoheadrightarrow}$ (Axiom 2). The recursion stops when all processes in S have been given a chance to execute a single atomic stop, i.e., when S is empty, or when no members of S are enabled in the current context (Axiom 3).

Axiom 2 (Recurse $\overset{a}{\twoheadrightarrow}$)

$$\frac{(\langle T, \mathcal{D}\rangle \xrightarrow{a} \mathcal{D}') \wedge (\langle S, \mathcal{D}'\rangle \overset{a}{\twoheadrightarrow} \mathcal{D}'')}{\langle T{\cdot}S,\, T{\cdot}\mathcal{D}\rangle \overset{a}{\twoheadrightarrow} \mathcal{D}''}$$

Axiom 3 (Finish $\overset{a}{\twoheadrightarrow}$)

$$\frac{S \sqcap \mathit{enabled}^a(\mathcal{D}) = [\![\,]\!]}{\langle S, \mathcal{D}\rangle \overset{a}{\twoheadrightarrow} \mathcal{D}}$$

Note that in a transition $\langle S, \mathcal{D}\rangle \twoheadrightarrow \mathcal{D}'$, the members of S are typically also members of $\mathcal{D}.\pi$, in contrast to a transition $\langle T, \mathcal{D}\rangle \longrightarrow \mathcal{D}'$ (applied via Axiom 1) where T is not in $\mathcal{D}.\pi$. We do this so that members of S can "see" other members of S, therefore allowing behaviour such as one member of S disabling or even killing another. It is to allow for this possibility that Axiom 3 does not just check whether S is empty, since deadlock could result if there was a process in S which was not enabled in the current context. The way $\overset{a}{\twoheadrightarrow}$ is employed in our rules, at the beginning of a step in $\overset{a}{\twoheadrightarrow}$ all elements in S will be enabled members of $\mathcal{D}$.

4.4 Transitions for single processes

The rules for the operators defined in Fig. 4 are given in Fig. 9.

Sequential composition. The context after a sequential composition $N;\ TT$ is the context after observing the effect of node N on the initial context, and putting all of the processes in TT into the new context (Axiom 4).

Atomic composition. The general rule for atomic composition (Axiom 5) is defined similarly to Axiom 4, except that each process in TT is given a chance to take an atomic δ-transition (via the $\overset{\delta}{\twoheadrightarrow}$ relation) immediately after N is executed, without allowing interleaving from other processes. If not all processes in TT are enabled, the atomic composition can not transition. We thus preclude partial execution of atomic blocks – either the root node N and all the threads in TT may transition, or no step is taken.

We can straightforwardly specialise both Axiom 4 and Axiom 5 for the common case where the bag TT is the singleton $[\![T]\!]$. This gives a much simpler rule for atomic composition.

Rule 9 (Sequential composition (singleton))

$$\frac{\langle N, \mathcal{D}\rangle \xrightarrow{a} \mathcal{D}'}{\langle (N;\ T), \mathcal{D}\rangle \xrightarrow{a} T{\cdot}\mathcal{D}'}$$

Rule 10 (Atomic composition (singleton))

$$\frac{(\langle N, \mathcal{D}\rangle \xrightarrow{a} \mathcal{D}') \wedge (\langle T, \mathcal{D}'\rangle \xrightarrow{\delta} \mathcal{D}'')}{\langle (N;;\, T), \mathcal{D}\rangle \xrightarrow{a} \mathcal{D}''}$$

Axiom 4 (Sequential composition)

$$\frac{\langle N, \mathcal{D}\rangle \xrightarrow{a} \mathcal{D}'}{\langle (N;\ TT), \mathcal{D}\rangle \xrightarrow{a} TT \uplus \mathcal{D}'}$$

Axiom 5 (Atomic composition)

$$\frac{\begin{array}{c}\langle N, \mathcal{D}\rangle \xrightarrow{a} \mathcal{D}' \wedge \\ TT \subseteq \mathit{enabled}^{\delta}(TT \uplus \mathcal{D}') \wedge \langle TT, \mathcal{D}'\rangle \overset{\delta}{\twoheadrightarrow} \mathcal{D}''\end{array}}{\langle (N;;\ TT), \mathcal{D}\rangle \xrightarrow{a} \mathcal{D}''}$$

Axiom 6 (Nondeterministic composition)

$$\frac{\langle T_1, \mathcal{D}\rangle \xrightarrow{a} \mathcal{D}'}{\langle (T_1 \parallel T_2), \mathcal{D}\rangle \xrightarrow{a} \mathcal{D}'} \qquad \frac{\langle T_2, \mathcal{D}\rangle \xrightarrow{a} \mathcal{D}'}{\langle (T_1 \parallel T_2), \mathcal{D}\rangle \xrightarrow{a} \mathcal{D}'}$$

Axiom 7 (Else-skip (i))

$$\frac{\langle T, \mathcal{D}\rangle \xrightarrow{a} \mathcal{D}'}{\langle (\blacklozenge T), \mathcal{D}\rangle \xrightarrow{a} \mathcal{D}'}$$

Axiom 8 (Else-skip (ii))

$$\frac{\neg\ \mathit{enabled}^{a}(T, \mathcal{D})}{\langle (\blacklozenge T), \mathcal{D}\rangle \xrightarrow{a} \mathcal{D}}$$

Figure 9: Axioms for process constructors

Nondeterministic composition. A nondeterministic composition (Axiom 6) proceeds if one of the processes can take a step. If both processes are enabled, either may be selected (rendering the other one obsolete); if neither are enabled, the composition blocks. The rule can be generalised straightforwardly to any finite number of processes.

Axiom 11 (Nondeterministic composition (generalised))

$$\frac{\langle T_j, \mathcal{D}\rangle \xrightarrow{a} \mathcal{D}' \wedge j < n}{\langle (\parallel_{i<n} T_i), \mathcal{D}\rangle \xrightarrow{a} \mathcal{D}'}$$

Our nondeterministic choice operator corresponds to CSP's (angelic) choice operator. It is straightforward to also define rules for a demonic version, e.g., the rule for selecting the left side would be $\langle T_1 \sqcap T_2, \mathcal{D}\rangle \xrightarrow{a} T_1 \cdot \mathcal{D}$.

Else-skip. A tree $\blacklozenge T$ may transition normally if T is enabled (Axiom 7), but if T is not able to transition, it may be terminated (Axiom 8). We use this operator to define the Behavior Tree selection operator. The appropriate rules are given in the next section, after the definition of the guard node.

4.5 Transitions for nodes

In this section we give rules for the nodes in Fig. 5.

4.5.1 Specification command

Axiom 12 states that a specification command $[P(\mathcal{D}), Q(\mathcal{D}, \mathcal{D}')]$ can transition when its guard is enabled in the current context. It updates the state so that the relation Q is maintained.

Axiom 12 (Specification command)

$$\frac{P(\mathcal{D}) \wedge Q(\mathcal{D}, \mathcal{D}')}{\langle [P, Q], \mathcal{D}\rangle \xrightarrow{\delta} \mathcal{D}'}$$

Thus, assuming that the predicate P holds in the current context, and the effect of Q is to update $\mathcal{D}$ to $\mathcal{D}'$, the context after executing the specification command is $\mathcal{D}'$. Note that this is a δ-transition, since it is not a communication node. We can use Axiom 12 to give a meaning to the Behavior Tree nodes in Fig. 2, using the definitions in Figs. 6 and 7.

Rule 13 (Guard)

$$\frac{\mathcal{D}.\sigma(C) = s}{\langle (C\,???\,s\,???), \mathcal{D}\rangle \xrightarrow{\delta} \mathcal{D}}$$

Rule 14 (State realisation)

$$\langle (C[s]), \mathcal{D}\rangle \xrightarrow{\delta} \mathcal{D} \oplus \{C \mapsto s\}$$

Rule 15 (Spawn)

$$\langle (\mathsf{spawn}\,\ell), \mathcal{D}\rangle \xrightarrow{\delta} \rho(\ell) \cdot \mathcal{D}$$

Rule 16 (Kill)

$$\langle (\mathsf{kill}\,\ell), \mathcal{D}\rangle \xrightarrow{\delta} \mathit{filterk}(\ell, \mathcal{D})$$

Rule 17 (Reversion)

$$\langle (\mathsf{revert}\,\ell), \mathcal{D}\rangle \xrightarrow{\delta} \rho(\ell) \cdot \mathit{filterk}(\ell, \mathcal{D})$$

We can define transitions for selections (bottom of Fig. 4) using the else-skip operator.

Rule 18 (Single selection)

$$\frac{\mathcal{D}.\sigma(C) = s}{\langle (C\,?\,s\,?;\ T), \mathcal{D}\rangle \xrightarrow{\delta} T \cdot \mathcal{D}}$$

Proof. From Definition 1, Axiom 7, Axiom 9 and Rule 13. □

Rule 19 (Single selection fail)

$$\frac{\mathcal{D}.\sigma(C) \neq s}{\langle (C\,?\,s\,?;\ T), \mathcal{D}\rangle \xrightarrow{\delta} \mathcal{D}}$$

Proof. From Definition 1 and Axiom 8. □

As with nondeterminism, the rules generalise straightforwardly to any finite number of processes.

Example. We can derive the following rule which summarises the effect a state realisation when it is the root node of a sequential composition. Consider the process $(C[s];\ T)$ operating in parallel with the context formed from the bag of threads π, in state σ. After executing $C[s]$ the bag of active processes will be π with T, and σ will be updated to map C to s.

Rule 20 (State realisation in sequential composition)

$$(C[s];\ T) \cdot (\pi, \sigma) \Longrightarrow (T \cdot \pi, \sigma \oplus \{C \mapsto s\})$$

Proof.

$$\begin{array}{l} \quad (C[s];\ T) \cdot (\pi, \sigma) \Longrightarrow (T \cdot \pi, \sigma \oplus \{C \mapsto s\}) \\ \Leftarrow \text{Axiom 1} \end{array}$$

Axiom 21 (Send message)

$$\frac{\begin{array}{c}\langle R, \mathcal{D}\rangle \xrightarrow{\delta} \mathcal{D}' \wedge \\ \langle \mathit{enabled}^{m(\vec{v})}(\mathcal{D}'), \mathcal{D}'\rangle \xrightarrow{m(\vec{v})} \mathcal{D}''\end{array}}{\langle (R\, \mathsf{send}\; m(\vec{v})), \mathcal{D}\rangle \xrightarrow{\delta} \mathcal{D}''}$$

Axiom 22 (Receive message)

$$\frac{\begin{array}{c}\langle R, \mathcal{D}\rangle \xrightarrow{\delta} \mathcal{D}' \wedge \\ \#\vec{x} = \#\vec{v}\end{array}}{\langle (R\, \mathsf{recv}\; m(\vec{x})), \mathcal{D}\rangle \xrightarrow{m(\vec{v})} \mathcal{D}' \oplus (\vec{x} \mapsto \vec{v})}$$

Axiom 23 (Synchronise (participate))

$$\frac{\langle R, \mathcal{D}\rangle \xrightarrow{\delta} \mathcal{D}'}{\langle (R\, \mathsf{sync}\; m), \mathcal{D}\rangle \xrightarrow{m} \mathcal{D}'}$$

Axiom 24 (Synchronise (initiate))

$$\frac{\begin{array}{c}\mathit{enabled}^{m}(\mathcal{D}) = \{T \colon C.\pi \mid m \in \alpha(T)\} \wedge \\ \langle (R\, \mathsf{send}\; m), \mathcal{D}\rangle \xrightarrow{\delta} \mathcal{D}'\end{array}}{\langle (R\, \mathsf{sync}\; m), \mathcal{D}\rangle \xrightarrow{\delta} \mathcal{D}'}$$

Figure 10: Axioms for communication nodes

$\quad \langle (C[s];\ T), (\pi, \sigma)\rangle \xrightarrow{\delta} (T{\cdot}\pi, \sigma \oplus \{C \mapsto s\})$
$\Leftarrow$ Axiom 9
$\quad \langle (C[s]), (\pi, \sigma)\rangle \xrightarrow{\delta} (\pi, \sigma \oplus \{C \mapsto s\})$
$\Leftarrow$ Rule 14

□

This rule models the basic execution of a Behavior Tree system: sequential processes operating in parallel which modify the state.

4.5.2 Sending a message

A message m is sent via a $R\, \mathsf{send}\; m(\vec{v})$ node. It allows each process that is waiting to receive m with the right number of parameters (i.e., that can transition in $\xrightarrow{m(\vec{v})}$) to make an atomic step. Axiom 21 in Fig. 10 first executes the state-based step associated with specification command R (transitioning to intermediate context $\mathcal{D}'$), then triggers each process waiting for message m to take a step, resulting in final context $\mathcal{D}''$. A simple case of this rule is where no parameters are associated with the message, and where no state-based behaviour is required.

Rule 25 (Send message)

$$\frac{\langle \mathit{enabled}^{m}(\mathcal{D}), \mathcal{D}\rangle \overset{m}{\twoheadrightarrow} \mathcal{D}'}{\langle (\mathsf{send}\; m), \mathcal{D}\rangle \xrightarrow{\delta} \mathcal{D}'}$$

In this case, each process which is enabled to transition in m is given a chance to take a step.

4.5.3 Receive message

The basic process that can participate in transition relation $\xrightarrow{m(\vec{v})}$ is a $R\, \mathsf{recv}\; m(\vec{v})$ node. Such a node is triggered by the execution of the corresponding $\mathsf{send}\; m(\vec{v})$ node – see Axiom 22 in Fig. 10. Given that the process is waiting for the correct number of parameters to message name m, the rule executes the state-based step associated with specification command R (transitioning to intermediate context $\mathcal{D}'$), then updates the resulting context so that the variables listed in $\vec{x}$ are mapped to the corresponding values in $\vec{v}$. The notation $\vec{x} \mapsto \vec{v}$ is defined below.

$$\vec{x} \mapsto \vec{v} = \{i \colon 0..\#\vec{x} - 1 \bullet (\vec{x}(i), \vec{v}(i))\}$$

As with sending a message, a much simpler rule can be derived when no parameters or specification command is associated with the reception of a message.

Rule 26 (Receive message)

$$\langle(\mathsf{recv}\, m), \mathcal{D}\rangle \xrightarrow{m} \mathcal{D}$$

Example communication. Consider a context containing a process which initially sends message m then behaves as T_1, and a process which waits for m then behaves as T_2. The final context is the original context with T_1 and T_2 in place of the sender and receiver. The proof obligation is that no other process is listening for m, otherwise those processes would also be triggered and modify the final context.

Rule 27 (Send to one)

$$\frac{\mathit{enabled}^m(\mathcal{D}) = [\![\,]\!]}{(\mathsf{send}\, m;\ T_1)\cdot(\mathsf{recv}\, m;\ T_2)\cdot\mathcal{D} \Longrightarrow T_1\cdot T_2\cdot\mathcal{D}}$$

Proof. First we note the following property, which follows from the antecedent and that $(\mathsf{recv}\, m;\ T_2)$ is enabled (not blocked) under transition $\xrightarrow{m}$ (see Rule 26).

$$\mathit{enabled}^m((\mathsf{recv}\, m;\ T_2)\cdot\mathcal{D}) = [\![(\mathsf{recv}\, m;\ T_2)]\!] \qquad (1)$$

We now complete the derivation.

$$\begin{aligned}
&(\mathsf{send}\, m;\ T_1)\cdot(\mathsf{recv}\, m;\ T_2)\cdot\mathcal{D} \Longrightarrow T_1\cdot T_2\cdot\mathcal{D}\\
\Leftarrow\ &\text{Axiom 1}\\
&\langle(\mathsf{send}\, m;\ T_1), (\mathsf{recv}\, m;\ T_2)\cdot\mathcal{D}\rangle \xrightarrow{\delta} T_1\cdot T_2\cdot\mathcal{D}\\
\Leftarrow\ &\text{Axiom 9}\\
&\langle(\mathsf{send}\, m), (\mathsf{recv}\, m;\ T_2)\cdot\mathcal{D}\rangle \xrightarrow{\delta} T_2\cdot\mathcal{D}\\
\Leftarrow\ &\text{Rule 25, and (1)}\\
&\langle[\![\mathsf{recv}\, m;\ T_2]\!], (\mathsf{recv}\, m;\ T_2)\cdot\mathcal{D}\rangle \overset{m}{\twoheadrightarrow} T_2\cdot\mathcal{D}\\
\Leftarrow\ &\text{Axiom 2}\\
&\langle(\mathsf{recv}\, m;\ T_2), \mathcal{D}\rangle \xrightarrow{m} \mathcal{D}' \wedge \langle[\![\,]\!], \mathcal{D}'\rangle \overset{m}{\twoheadrightarrow} T_2\cdot\mathcal{D}\\
\Leftarrow\ &\text{Simplify from Axiom 3}\\
&\langle(\mathsf{recv}\, m;\ T_2), \mathcal{D}\rangle \xrightarrow{m} T_2\cdot\mathcal{D}\\
\Leftarrow\ &\text{Axiom 4}\\
&\langle(\mathsf{recv}\, m), \mathcal{D}\rangle \xrightarrow{m} \mathcal{D}\\
\Leftarrow\ &\text{Rule 26}
\end{aligned}$$

□

4.5.4 Synchronisation

The intuition behind synchronisation is that when all threads that wish to synchronise on m are enabled, they are all given a chance to take an atomic step (where that step will typically be just to pass the synchronisation node). We model this using the message passing system introduced in the previous section, in conjunction with the synchronisation alphabet. Associating message passing with synchronisation is discussed at the end of this section.

In Axiom 23 in Fig. 10, we firstly allow $R\,\mathsf{sync}\, m$ nodes to participate in $\xrightarrow{m}$. Similarly to receive nodes (Axiom 22) we execute the specification command R in conjunction with the transition for message m. The simple case where there is no associated specification command is given below.

Rule 28 (Synchronise)

$$\langle(\mathsf{sync}\, m), \mathcal{D}\rangle \xrightarrow{m} \mathcal{D}$$

In addition to passively participating in a synchronisation, a sync node may also initiate a synchronisation if all listening threads are enabled (Axiom 24). A synchronisation is modelled by one of the nodes sending the message m to all of its fellow synchronisation nodes, as long as the processes currently enabled for m ($\mathit{enabled}^m(\mathcal{D})$) are exactly those that participate in m ($\{T\colon C.\pi \mid m \in \alpha(T)\}$); otherwise all synchronising nodes are blocked. Axiom 23 and Axiom 24 are intentionally similar to the axioms for receiving and sending messages; one of the nodes acts as the initiator of the synchronisation message (Axiom 24), and the others respond (Axiom 23). We can abstract away from this model by using the following transition in $\Longrightarrow$, which allows synchronisation to "spontaneously" occur, once the conditions have been met.

Rule 29 (Synchronisation ($\Longrightarrow$)) *Assuming $\mathcal{D}$ contains at least one thread of the form $(R\,\mathsf{sync}\,m;\ T)$,*

$$\frac{\begin{array}{c}\mathit{enabled}^m(\mathcal{D}) = \{T\colon C.\pi \mid m \in \alpha(T)\} \wedge \\ \langle(\mathsf{send}\,m), \mathcal{D}\rangle \xrightarrow{\delta} \mathcal{D}'\end{array}}{\mathcal{D} \Longrightarrow \mathcal{D}'}$$

Proof. From Axiom 24 (and Rule 9). □

Example. Consider the case where exactly two process are waiting to synchronise on m.

Rule 30 (Synchronise twins)

$$\frac{m \notin \alpha(\mathcal{D})}{(\mathsf{sync}\,m;\ T_1)\cdot(\mathsf{sync}\,m;\ T_2)\cdot\mathcal{D} \Longrightarrow T_1\cdot T_2\cdot\mathcal{D}}$$

Proof. From Rule 29, Axiom 2, Axiom 3, and from the antecedent:

$$\mathit{enabled}^m((\mathsf{sync}\,m;\ T_1)\cdot(\mathsf{sync}\,m;\ T_2)\cdot\mathcal{D}) = [\![(\mathsf{sync}\,m;\ T_1), (\mathsf{sync}\,m;\ T_2)]\!]$$

□

Similarly to Rule 27, after the synchronisation the processes have passed their synchronisation point and T_1 and T_2 are now active.

4.6 Discussion

Below we briefly discuss some reasons for, and issues rising from, modelling behaviour tree systems using the semantics given here.

No parallel operator. We do not have an explicit parallel composition operator. In effect the bag of active process is a more primitive method for expressing parallelism; we could alternatively consider the bag as a generalised parallel composition. Axiom 1 corresponds to the rule for parallel composition in other process algebras. The difference is that we always have parallel composition at the top level of the system. This allows us to more easily express process manipulation behaviour, such as killing threads and broadcast messages, than would be allowed if each process could not "see" all other processes. In CSP-like algebras, killing of other threads is handled less generally by mechanisms such as interrupts. However, our approach is restrictive in other ways since we do not allow parallelism at lower levels of nesting, e.g., in this paper we do not allow one choice in a nondeterministic composition to be a bag of processes operating in parallel (though the generalisation is straightforward).

Atomicity and parallelism. Axiom 5 for atomic composition is not intuitive, mainly stemming from the unintuitive nature of combining atomicity with concurrency. We have taken the approach that a bag of threads can take an "atomic" step by allowing each member of that bag to individually take an atomic step (in some nondeterministically chosen order). While this does not appear useful in practice since it is difficult to implement, interestingly, the concept is useful when describing the behaviour of message passing and synchronisation. However, when atomic composition is used with a singleton bag of processes on the right-hand side, the rule specialises to our usual notion of atomicity (Rule 10).

Buffered communication. It is common for communication to occur on a buffer, which can vary in length from one to being unbounded, and vary according to whether it is blocking or nonblocking. Communication along a buffer is a case of shared variable communication, where the buffer is treated as a member of the state. It is easy to define a set of commands that manipulate the buffer in the desired fashion using specification commands.

Synchronised message passing. In CSP, *channels* model a flow of information between synchronised processes (where the sender is blocked until there is a receiver). We may straightforwardly extend the definition of send to model the sending of a message which is blocked until all receivers are ready (and the extension to "at least one" receiver is ready is similarly straightforward). Following the style of CSP, we decorate a blocking send message node along a channel with '!'. This node will only send the message when all receivers (which may be optionally decorated with '?') are ready.

Axiom 31 (Synchronised send)

$$\frac{\begin{array}{c} \mathit{enabled}^m(\mathcal{D}) = \{\, T \colon C.\pi \mid m \in \alpha(T)\} \wedge \\ \langle (R\,\mathsf{send}\; m(\vec{v})), \mathcal{D}\rangle \xrightarrow{\delta} \mathcal{D}' \end{array}}{\langle (R\,\mathsf{send!}\; m(\vec{v})), \mathcal{D}\rangle \xrightarrow{\delta} \mathcal{D}'}$$

Thus a blocking send node (a CSP channel) behaves as a non blocking send node, with the additional constraint that all receivers are enabled (the message name must be added to the synchronisation alphabets of the receiving processes).

4.7 Simulation

The process algebra BTPA and its operational semantics have been implemented as a simulation tool in the Mercury logic programming language [SHC95, Mer]. The simulator, BTsim, takes as input a Behavior Tree using the constructs in Sect. 2, the initial values for the components in the system, and a list of safety properties for it to check. It converts the Behavior Tree into BTPA, automatically labelling the tree and determining the alphabet of each process. For a terminating system, or a non-terminating system that only interacts finitely often with the environment, BTsim can be used to nondeterministically generate a single run of the BT system, or can generate all possible runs (subject to hardware constraints), and can check if the safety properties are maintained after each atomic step. The translation of the operational rules into Mercury was quite straightforward, and also fed back into the development of the semantics.

Though it is possible to check some simple safety properties and produce counter examples if they are violated, it is not intended to develop the simulator into a fully functional model checking tool. In other work, a model checking tool has been developed for Behavior Trees using SAL [GLWY05]. It can check invariants as well as LTL formulas.

5 Conclusions

In this paper we have given a process algebra and operational semantics that can be used to model the Behavior Tree notation. The semantics handles synchronisation, message passing and testing and updating the state. The rules and constructs are intended to form a small but powerful set of primitives on which more complex behaviour can be built. The subset of the semantics corresponding to Behavior Trees has been implemented as a simulation tool.

The development of a new process algebra, rather than using an existing one, was motivated by the Behavior Tree concepts of process killing, atomic composition, and non-blocked message passing. Some of the operational rules appear less elegant than in other formal systems, particularly the rule for sending a message, but this appears to be a by-product of having a language which combines state with atomicity and synchronisation. Despite this, the rules are relatively compact, and were easily implemented in the simulation tool (Sect. 4.7).

For brevity we have assumed simple underlying definitions. In particular, we have not allowed components to have *attributes*, though this is common in Behavior Tree models. The extensions required to allow this are straightforward and hence have been omitted from this document. We could also extend the underlying type *Val*, partitioning it into subtypes and associating a type for each component. This extension has been successfully applied in many other systems, and hence, to keep the presentation uncluttered, we have maintained a simple untyped model.

A Multiset operators

The multiset (or bag) operators are defined below, treating bags with elements of type T as partial functions $T \mathrel{⇸} \mathbb{N}_1$.

$b\#$	$(\lambda\, e: T \bullet 0) \oplus b$
$b_1 \uplus b_2$	$(\lambda\, e: T \bullet b_1\#e + b_2\#e) \rhd \mathbb{N}_1$
$b_1 \sqcap b_2$	$(\lambda\, e: T \bullet min(b_1\#e, b_2\#e)) \rhd \mathbb{N}_1$
$b_1 - b_2$	$(\lambda\, e: T \bullet b_1\#e - b_2\#e) \rhd \mathbb{N}_1$
$e \cdot b$	$[\![e]\!] \uplus b$
$b - e$	$b - [\![e]\!]$
$[\![e: b \mid P(e)]\!]$	$\{e: \mathrm{dom}\, b \mid P(e)\} \lhd b$

B Labelling processes

In Sect. 3.1 a function $\rho: BTLabel \mathrel{⇸} Proc$ was introduced as part of the execution environment, which maps labels, as used by spawn, kill and revert nodes, to processes. The mapping is used to filter out processes in Definition 3, *filterk*. However, from the point of view of simulation, the subterm ordering is not an efficient manner of defining the subprocess ordering $\preceq$. In this section we introduce a system of defining the mapping ρ such that the subprocess ordering may be retrieved by examining the labels of the subprocesses. For these purposes we will define the type $BTLabel$ as $\mathrm{seq}\,\mathbb{N}$.

For a given process $\mathcal{T}$, let the set of subprocesses (subterms) of $\mathcal{T}$ be given by $subprocs(\mathcal{T})$. Then we define the set of order-preserving mappings on $\mathcal{T}$ as

$$label_{\mathcal{T}} \mathrel{\widehat{=}} \{f: subprocs(\mathcal{T}) \rightarrow BTLabel \mid (\forall\, T_1, T_2: \mathrm{dom}\, f \bullet T_1 \preceq T_2 \Leftrightarrow f(T_2)\,\mathsf{prefix}\,f(T_1))\}$$

That is, each subtree of $\mathcal{T}$ contains $\mathcal{T}$'s label as a prefix. Intuitively, an element $f \in label_{\mathcal{T}}$ can be constructed by recursively descending depth-first from the root node of $\mathcal{T}$, with each subprocess extending the label of its parent. Hence, the full tree $\mathcal{T}$ is mapped to, say, $\langle 0 \rangle$, while the leaf nodes will be of length n, depending on how deeply they occur within $\mathcal{T}$.

The set of inverses of $label_{\mathcal{T}}$ is defined by

$$\rho_{\mathcal{T}} \mathrel{\widehat{=}} \{\rho: BTLabel \mathrel{⤔} subprocs(\mathcal{T}) \mid (\forall\, \ell_1, \ell_2: \mathrm{dom}\, \rho \bullet \rho(\ell_2) \preceq \rho(\ell_1) \Leftrightarrow \ell_1\,\mathsf{prefix}\,\ell_2)\}$$

An element of $\rho_{\mathcal{T}}$ suffices for ρ in the execution environment, while an element of $label_{\mathcal{T}}$ effectively augments each process in $\mathcal{T}$ with a label.

We define a function for constructing ρ below.

$$labelTree: (Proc \times BTlabel) \rightarrow (BTlabel \rightarrow Proc)$$

Where ℓ is a label, $\ell 0$ is the label $\ell \frown \langle 0 \rangle$. Similarly for ℓi, where i is a number.

$$
\begin{array}{l}
\mathit{labelTree}(N;\ T, \ell) = \mathit{labelTree}(T, \ell 0) \oplus \{\ell \mapsto N;\ T\} \\
\mathit{labelTree}(N;;\ T, \ell) = \mathit{labelTree}(T, \ell 0) \oplus \{\ell \mapsto N;;\ T\} \\
\mathit{labelTree}(\parallel_{i<n} T_i, \ell) = (\bigcup_{i<n} \mathit{labelTree}(T_i, \ell i)) \oplus \{\ell \mapsto \parallel_{i<n} T_i\} \\
\mathit{labelTree}(\blacklozenge T, \ell) = \mathit{labelTree}(T, \ell 0)) \oplus \{\ell \mapsto \blacklozenge T\}
\end{array}
$$

As an example, given a process

$$\mathcal{T} \mathrel{\widehat{=}} a;\ ((b;\ [\![c, d, g]\!]) \parallel (e;\ f))$$

we construct the following labelling scheme for each of the subprocesses of $\mathcal{T}$. We choose the label of $\mathcal{T}$ to be the singleton sequence "0"[4].

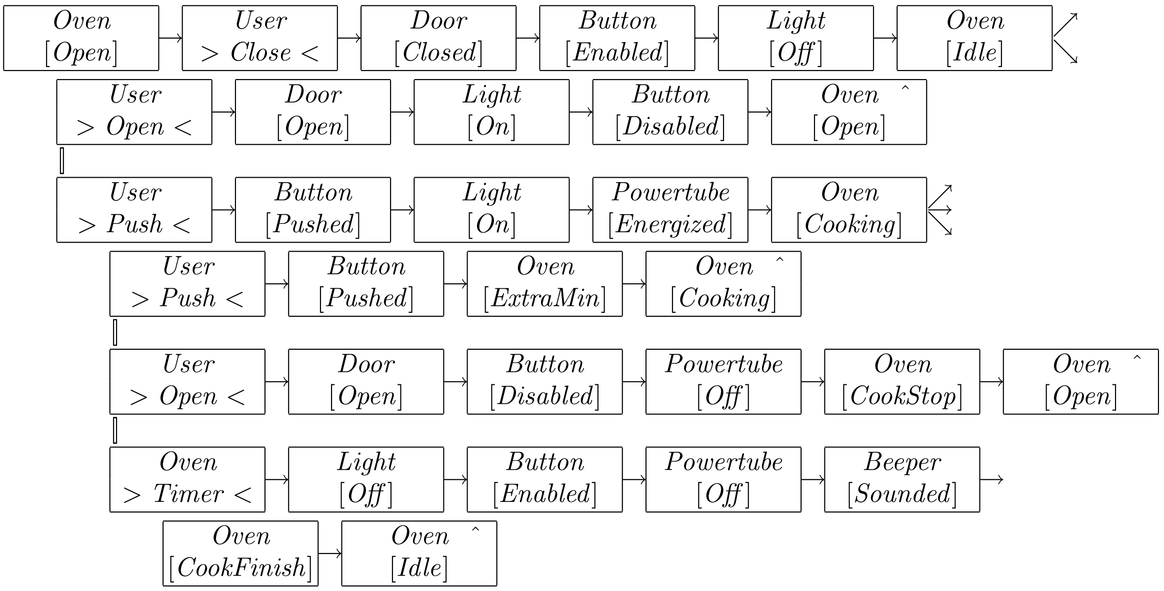
$$
\begin{array}{lcl}
0 & \mapsto & a;\ ((b;\ [\![c, d, g]\!]) \parallel (e;\ f)) \\
00 & \mapsto & (b;\ [\![c, d, g]\!]) \parallel (e;\ f) \\
000 & \mapsto & (b;\ [\![c, d, g]\!]) \\
0000 & \mapsto & c \\
0001 & \mapsto & d \\
0002 & \mapsto & g \\
001 & \mapsto & (e;\ f) \\
0010 & \mapsto & f
\end{array}
$$

It is easy to check that each subprocess's label contains $\mathcal{T}$'s label (0) as a prefix. Similarly, we can see that d (label 0001) is a subprocess of $(b;\ [\![c, d, g]\!])$ (label 000). There is no relationship between, for instance, d and f, since neither is a prefix of the other (though we can determine that their closest common ancestor must be the process labelled 00).

C Example Behavior Trees

In this section we present several versions of the microwave example presented in [Win04, Figure 6]. There are some slight differences because the notation and approach have been developed since publication.

C.1 Microwave Behavior Tree in box notation



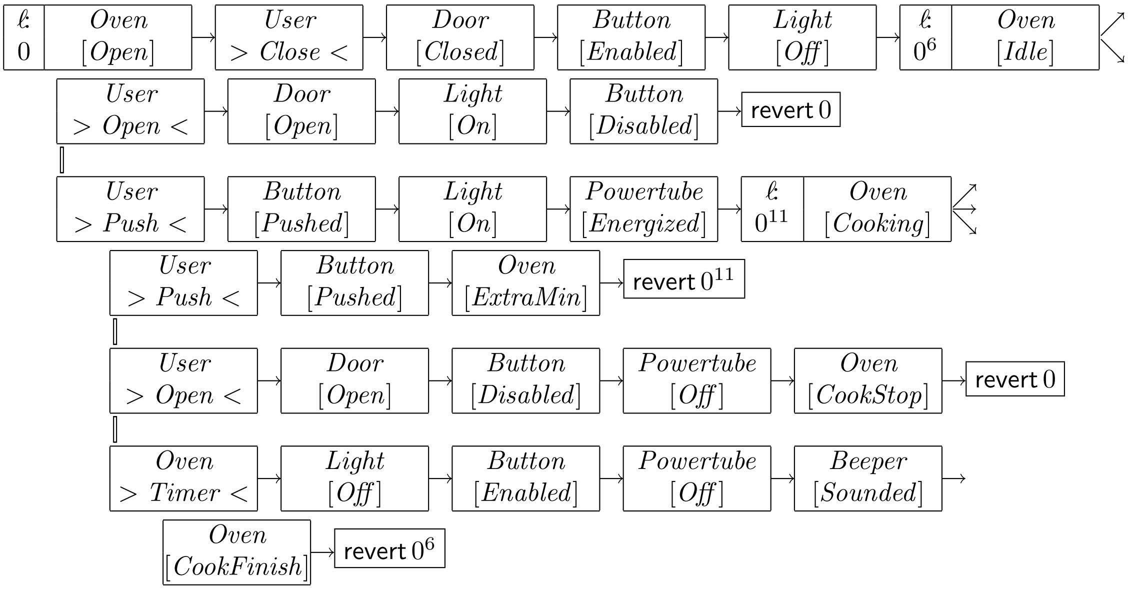
[4] We will omit the usual sequence bracketing notation and instead write the list of numbers as a string; this is acceptable in the context of this example because we do not require two-digit numbers, i.e., no process has more than ten direct subprocesses.

C.2 Microwave Behavior Tree translated to BTPA

In this section we show how the microwave example is expressed in BTPA. Firstly we construct the label mapping ρ using function *labelTree* defined in Appendix B. The main tree, rooted at $Oven[Open]$, is labelled 0. Its subtree is labelled 00, rooted at $User > Close <$. We label the subsequent nodes similarly until we reach the first branch point. The subtrees rooted at $User > Open <$ and $User > Push <$ are labelled 0000001 and 0000000, respectively. In actuality, the only interesting labels in ρ are the reversion points (since we do not have any spawns or kills). We write 0^i to indicate a label formed from i 0s. We have

$\rho(0) = Oven[Open]; \ ...$
$\rho(00) = User > Close <; \ ...$
$\rho(0^6) = Oven[Idle]; \ ...$
$\rho(0^{11}) = Oven[Cooking]; \ ...$

Now we translate the tree itself. There are no selections, spawns, kills or synchronisations, so we need only worry about the reversions; the rest of the tree remains the same. We have added labels to the three destination reversion nodes, and translated the reversion (source) nodes to use the revert keyword.



References

[BKS88] R.J.R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Trans. Program. Lang. Syst.*, 10(4):513 – 554, 1988.

[But92] M.J. Butler. *A CSP Approach to Action Systems.* PhD thesis, Computing Laboratory, Oxford University, 1992.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley Longman Publishing Co., Inc., 1988.

[Dro03] R. Geoff Dromey. From Requirements to Design: Formalizing the Key Steps, Keynote Address. In *SEFM*, pages 2–11. IEEE Computer Society, 2003.

[Dro06] R.G. Dromey. Formalizing the transition from requirements to design. *Mathematical Frameworks for Component Software: Models for Analysis and Synthesis*, page In press, 2006.

[ED03] R. Eshuis and J. Dehnert. Reactive Petri nets for Workflow Modeling. In W.M.P. van der Aalst and E. Best, editors, *Application and Theory of Petri Nets*, volume 2679 of *Lecture Notes in Computer Science*, pages 295–314. Springer-Verlag, Berlin, 2003.

[GLWY05] L. Grunske, P. Lindsay, K. Winter, and N. Yatapanage. 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. Conf. on Integrated Formal Methods (IFM 2005)*, volume 3771 of *LNCS*, pages 129–149. Springer-Verlag, 2005.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HN96] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.

[Hoa85] C. A. R. Hoare. *Communicating sequential processes.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[Man] Mantara Software. `http://www.mantara.com`.

[Mer] Mercury home page. `http://www.cs.mu.oz.au/research/mercury/index.html`.

[Mil82] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[Mil99] R. Milner. *Communicating and mobile systems: the pi-calculus.* Cambridge University Press, 1999.

[Pet81] J.L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.

[RJB98] Jim Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1998.

[SAB$^+$00] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content based routing with Elvin. In *Proceedings of AUUG2K*, June 2000.

[SHC95] Z. Somogyi, F.J. Henderson, and T.C. Conway. Mercury, an efficient purely declarative logic programming language. In R. Kotagiri, editor, *Proceedings of the Eighteenth Australasian Computer Science Conference*, pages 499–512, Glenelg, South Australia, 1995. Australian Computer Science Communications.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, second edition, 1992.

[Ste99] W. Richard Stevens. *UNIX Network Programming, Volume 2 (2nd ed.): Interprocess Communications.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[SWH$^+$04] C. Smith, K. Winter, I. J. Hayes, R. G. Dromey, P. A. Lindsay, and D. A. Carrington. An environment for building a system out of its requirements. In *Automated Software Engineering (ASE)*, pages 398–399. IEEE Computer Society, 2004.

[WC02] J. C. P. Woodcock and A. L. C. Cavalcanti. The semantics of *circus*. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184—203. Springer-Verlag, 2002.

[Win04] K. Winter. Formalising behaviour trees with CSP. In *Integrated Formal Methods*, volume 2999 of *LNCS*, pages 148–167. Springer Verlag, April 2004.