Science of Computer Programming 76 (2011) 891–914

Contents lists available at ScienceDirect

Science of Computer Programming

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

A semantics for Behavior Trees using CSP with specification commands

Robert J. Colvin [a,b,*], Ian J. Hayes [a]

[a] The University of Queensland, Australia
[b] The Queensland Brain Institute, Australia

ARTICLE INFO

Article history:
Available online 9 December 2010

Keywords:
Structural operational semantics
Communicating Sequential Processes (CSP)
Hierarchical state
Specification commands
Process algebras
Behavior Trees
Requirements modelling

ABSTRACT

In this paper we give a formal definition of the requirements translation language *Behavior Trees*. This language has been used with success in industry to systematically translate large, complex, and often erroneous requirements documents into a structured model of the system. It contains a mixture of state-based manipulations, synchronisation, message passing, and parallel, conditional, and iterative control structures. The formal semantics of a Behavior Tree is given via a translation to a version of Hoare's process algebra CSP, extended with state-based constructs such as guards and updates, and a message passing facility similar to that used in publish/subscribe protocols. We first provide the extension of CSP and its operational semantics, which preserves the meaning of the original CSP operators, and then the Behavior Tree notation and its translation into the extended version of CSP.

1. Introduction

A system developer is often faced with a system requirements document containing hundreds, or even thousands, of requirements, written in a natural language, and by a varied group of people, each with specialised domain knowledge. Unsurprisingly, such documents may be filled with problems, such as ambiguity, inconsistency, redundancy, and incompleteness. The process of transforming such documents into a working system must therefore be able to identify issues with the natural language requirements in a way that is easy for the client to understand, and the model must be structured such that it can be cross referenced with the original document.

The Behavior Tree notation was developed by Dromey to address this problem [1–3]. Because it is designed for use by both client and expert modeller, it is a graphical notation, and contains a range of constructs that cover state-based manipulations, as well as more abstract concepts such as synchronisation and message passing, along with typical concurrency, choice and iteration control structures familiar to specification and programming languages. The notation is designed to be easy for a non-expert to understand in a relatively short amount of time.

Each requirement is translated into its own, small Behavior Tree, and each node in the tree is tagged with the number of the requirement from which it was translated, allowing traceability back to the original informal requirements. The requirements may then be progressively *integrated* into a whole-system tree, by finding syntactically matching constructs. This process will reveal inconsistencies, redundancies, incompleteness, and ambiguities. The constructed tree can then serve as the basis for discussion between developer and client for validation purposes, using the traceability tags on each node to cross reference to the original document. Once a validated tree is defined, the developer has a systematically structured representation of the system, which can serve as the basis for further development work.

Experience with industrial trials indicate that the modelling process is better at detecting errors in requirements than other techniques [4,5]. The Behavior Tree process has been adopted for industrial use, in particular by Raytheon Australia [6],

* Corresponding author at: The University of Queensland, Australia.
E-mail address: robert@itee.uq.edu.au (R.J. Colvin).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.11.007

who have invested resources to developing a Behavior Tree editor [7]. Existing tools [8,9] include facilities for ensuring well-formedness and model checking; due to its syntax-driven nature, another aspect that may be suitable for tool support includes the integration of Behavior Trees, and as a result the identification of errors in the requirements.

In this paper we present a formal semantics for the Behavior Tree notation. As its base we use Hoare's process algebra Communication Sequential Processes (CSP) [10,11], a well established and elegant formal notation for describing interactions between concurrent processes. We extend this language to include state-based constructs such as tests and updates, which are common within requirements documents, and a message passing facility similar to publish/subscribe models of communication [12]. We call this new language CSP_σ. The extensions and operational semantics of CSP_σ are defined so that the original laws of CSP are preserved.

The most immediate motivation for providing a formal semantics for the Behavior Tree notation is to add precision to Behavior Tree models. As a result, the consequences of modelling decisions are easier to understand, and ambiguities and inconsistencies are removed from the models themselves. In the longer term, the semantics may be used as the basis for developing automated analysis of system behaviour, in particular, simulation and model checking. It is for these longer-term goals that the semantics is defined as an extension of CSP, with the intention that tools and techniques for Behavior Trees may extend existing tools and techniques for CSP [13,14].

The paper is structured as follows. In Section 2 we present CSP extended with state. In Section 3 we present a further extension which includes message passing. In Section 4 we present the Behavior Tree notation, and in Section 5 we describe how to translate Behavior Trees into the extended version of CSP. For the remainder of this section we consider related work.

1.1. Related work: requirements modelling

The Behavior Tree notation shares much in common with other formal (and informal) specification languages, but is targeted at mapping typical requirements in a straightforward manner, rather than as a vehicle for abstract specification. That is, the notation is designed so that a client can understand the models, and the models can be mapped back to their original statement of requirements.

The *Unified Modelling Language* (UML) [15] is also used for constructing a model from requirements. The main point of difference with Behavior Trees is that a UML model is formed from several different diagram types, many of which do not easily support traceability back to the original requirements. In comparison, the *Behavior Engineering* development framework comprises only two diagram types, Behavior Trees and *Composition Trees*, both of which support traceability.[1] Furthermore, the semantics of UML has not been fully formalised [15, Section 8].

1.2. Related work: Behavior Tree semantics

There are several previous definitions of the semantics of Behavior Trees. In particular a technical report by the authors of this paper [16], which defined a process algebra for capturing the constructs of Behavior Trees directly. While comprehensive, the operational semantics were overly complicated, and did not exhibit desirable properties such as compositionality of parallel Behavior Trees. In contrast, in this paper we use an existing process algebra, CSP, for the underlying definitions, and this provides a more elegant specification of interactions between processes, and is inherently compositional.

Earlier definitions of the semantics of Behavior Trees include a translation into CSP but without the extension of state [17], and translations to automata-based languages such as action systems [8] and timed/probabilistic automata [18,19]. The translation in [17] is complicated when complex state is involved, as CSP does not naturally handle mutable state (this is explored in more detail in [20]). The work in this paper uses a similar translation technique, but with a version of CSP extended with state, which makes many of the translations simpler. The translations to state-based notations [8,18,19] resulted in complex configurations required to represent concurrency and the control structures of Behavior Trees. They were also targeted specifically at model checking, and hence were written more for efficiency than elegance. In this paper, we present the semantics using an established and elegant process algebra as its core, with a straightforward translation process, which preserves the structure of the original Behavior Tree. Compared with the semantics mentioned above, this gives further confidence in validating the formal semantics that we present here against the informal semantics described for Behavior Trees by Dromey [1–3].

1.3. Related work: process algebras with state

CSP has been integrated with state-based languages, for instance, with *Z* by Woodcock and Cavalcanti (Circus) [21], with Object-*Z* by Smith [22] and Fischer and Wehrheim (CSP-OZ) [23], with Action Systems by Butler [24], and with *B* by Butler and Leuschel [25] and by Schneider and Treharne [26]. In comparison with these approaches to combining state-based specification with CSP, we have taken a "lightweight" approach, introducing only a single construct for defining state

[1] Composition Trees give the static declarations of the system, such as the components, states and events which occur within the system, in a hierarchical manner similar to the static declarations of other languages. We do not consider them in detail in this paper.

manipulation, and with little change to the underlying syntax and semantics of CSP. In the languages mentioned above, there is a notational and informational overhead associated with combining two pairs of syntax and semantics. Of course, the addition of state tests and updates does not provide the same richness of specification as afforded by a full combination of CSP with, for example, *B*, but does provide a useful stepping stone between event-based and state-based specifications. We have taken this approach for defining the semantics of Behavior Trees in an attempt to keep the translation as simple as possible and to preserve the structure of the original Behavior Tree in the resulting CSP_σ process.

Baeten and Bergstra [27,28] define a process algebra with state, which can be tested and updated through *propositional signals*. In their framework the state is anonymous, and each action is defined to test or modify the state in some way. The use of local state by Baeten and Bergstra, as with local state in CSP_σ, are examples of *contexts* as explored generically by Larsen and Xinxin [29]. A recent extension of CSP by Sun et al., CSP# [30], introduces shared variables and sequential programs, and is supported by the model checker PAT [31]. That language is roughly equivalent in expressiveness to CSP_σ, except that CSP_σ includes a general specification command, which abstractly represents any atomic update of the state, whereas CSP# allows sequential code blocks using typical imperative constructs. This difference is because we define a specification language, while CSP# is designed for efficient model checking. The main point of difference with CSP_σ is in the style of operational semantics and handling of variables. Our transition rules define a relation on *Processes*, while the transition rules of [30] define a relation on *Process* $\times$ *State* pairs. This means that our rules collapse to the standard CSP rules when state is not involved, and that the majority of rules are relatively concise. This style also admits concurrent processes to use the same variable name without conflict, whereas this must be explicitly disallowed in CSP#, as with any language in which the *State* is kept globally rather than hierarchically. Future work is to reconcile the differences between CSP_σ and CSP#, with the intention of using the PAT tool for model checking and animating CSP_σ processes, and therefore Behavior Trees.

2. CSP_σ

In this section we describe the language CSP_σ, which is CSP extended with state-based constructs. CSP_σ was originally introduced in [20], although the presentation given here differs in that it includes definitions for interrupts, restarts, explicit recursion, and sending expressions on channels. We also use an interface parallel composition operator rather than Hoare's original alphabetised parallel composition operator [10], as interface parallel is more flexible and corresponds with more recent presentations of CSP [11,32].

The language CSP_σ is a process algebra which allows concurrent processes to communicate synchronously via shared events, and to manipulate and check the value of state variables. Synchronisation and variable manipulations may be combined atomically, subject to certain restrictions described below.

2.1. Syntax

Basic types. The basic unit for synchronisation are *events*, given by the set Σ. Elements of the set Σ are either an event name or the pairing of a channel name with an associated value in the set *Val*. The set $\Sigma^{\tau,\checkmark}$ is Σ extended to include the special events τ, representing a (hidden) *internal* event, and $\checkmark$, representing termination.

We assume a set of variable names *Var*, and define a *State* (also sometimes called a *valuation* or *store*) as a finite partial mapping from variables to values.

$$State \mathrel{\widehat{=}} Var \twoheadrightarrow Val$$

We assume *Val* contains the booleans and integers, and whatever other values that are required for a particular application. A state in which the variable i has value 0 and j has value 1 is represented by the mapping $\{i \mapsto 0, j \mapsto 1\}$.

Expressions and predicates. Single-state expressions are given by the type $Expr_1$, and are terms which may contain elements of *Var*. We assume an expression syntax which contains the standard operators of logic, arithmetic and set theory. An expression, E, may be instantiated with a state, σ, to form a new expression, $E[\sigma]$; it is the expression obtained by replacing all of the free variables in E that are also in the domain of σ with their value in σ. This may return a "ground" expression (containing no free variables) that can be evaluated to an element of *Val*, or another expression which has fewer free variables. For instance, an instantiation $(i > 0)[\{i \mapsto 1\}]$ is $(1 > 0)$ which evaluates to the *true*. Similarly, an instantiation $(i > j)[\{i \mapsto 1\}]$ is the (boolean) expression $1 > j$. We refer to boolean-valued expressions as predicates.

Two-state expressions are given by the type $Expr_2$, and contain free variables in *Var* as with $Expr_1$, but may also contain *primed* versions of *Var*. The primed versions indicate the post-state, while the unprimed versions indicate the pre-state. An instantiation of $E \in Expr_2$ requires two states, i.e., $E[\sigma, \sigma']$, where the variables in the domain of σ are replaced in E by their values in σ, and the primed variables in the domain of σ' are replaced by their values in σ'. For instance, $(i' = i + 1)[\{i \mapsto 0\}, \{i \mapsto 1\}]$ is $1 = 0 + 1$, which evaluates to *true*. When an expression may be either one or two state, we just use the type *Expr*. We say a predicate E is satisfiable, written $\mathbf{sat}(E)$, when there exist pre- and post-states σ and σ', defined for all free variables in E, such that $E[\sigma, \sigma']$ evaluates to true.

State-based constructs. Following Morgan [33], we introduce *specification commands* (*SCmd*) as the basic state-manipulation construct in the language. A specification command $x_1, \ldots, x_n$: [R] contains a two-state predicate R and a *frame* $x_1, \ldots, x_n$, which is the (possibly empty) set of variables which the command may alter. Therefore, the primed variables in the predicate must be a subset of the *frame*. An example is i: [$i' = i + 1$], which modifies the frame variable i so that in the post-state it has a value one greater than in the pre-state. When the frame of a specification command is empty (and hence its predicate does not refer to any post-state (primed) variables), we call it a *guard*, and write it as [g]. We also allow an *update*, $x := E$, where $x \in Var$ and E is a single-state expression, to abbreviate the specification command x: [$x' = E$].

A specification command x: [R] may be interpreted as a relation, SR, on total states, that satisfies R and modifies only variables in x. That is, given total states $S, S' \in (Var \to Val)$, the relation corresponding to x: [R], given by $[\![x\text{: } [R]]\!]$, contains (S, S') if S and S' make R true and only variables in x may differ.

$$(S, S') \in [\![x\text{: } [R]]\!] \Leftrightarrow (R[S, S'] \land x \lhd S = x \lhd S')$$

(The function $x \lhd S$ is the function S with its domain restricted to elements not in x.) We define equivalence of specification commands as follows.

$$c_1 \equiv c_2 \mathrel{\widehat{=}} [\![c_1]\!] = [\![c_2]\!]$$

A special specification command is id, which we define as [*true*]. It does not depend on nor change any variables. Note that there are many other commands which are equivalent to id by the definition above, such as [$5 > 1$], [$x = x$], x: [$x' = x$].

Events and processes. The syntax of *Events* and *Processes* are given below, where $A \subseteq \Sigma$ is a set of events, and *ch* and *rec* are identifiers.

$$\begin{array}{lcl}
Event & ::= & \Sigma^\tau \mid ch!E \mid ch?Var \\
P & ::= & (SCmd, Event) \to P \\
& \mid & P \,;\, P \\
& \mid & P \,[]\, P \\
& \mid & P \sqcap P \\
& \mid & P \underset{A}{\|} P \\
& \mid & (\mu\, rec \bullet P) \\
& \mid & P \backslash A \\
& \mid & (\textbf{state}\ State \bullet P) \\
& \mid & P \triangle P \\
& \mid & P\ \mathsf{restart}(\Sigma)\ P \\
& \mid & SKIP \\
& \mid & STOP
\end{array}$$

An event is either an identifier (the event name), or $ch!E$, indicating output of the value of expression E on channel *ch*, or $ch?y$, indicating receiving a value on channel *ch* and storing it in variable y.[2]

An *action prefix* process $(c, e) \to P$, where c is a specification command and e is an *Event*, is a process that tests and/or updates the state such that c is satisfied and simultaneously performs event e, before behaving as process P. In the case where a state test or update only is specified, the event e is τ. A sequential composition $P \,;\, Q$ behaves as P until P terminates, after which it behaves as Q. An *external choice* between processes P and Q is given by $P \,[]\, Q$. The choice is external because the environment selects P or Q through synchronisation. In contrast, an *internal choice* between P and Q, written $P \sqcap Q$, nondeterministically chooses between P and Q, without reference to the environment. Concurrency is written as $P \underset{A}{\|} Q$, which states that the two processes operate in parallel, synchronising on events in the *interface* A, and interleaving other events. A recursive process is defined using the fix-point operator μ as $(\mu\, rec \bullet P)$. Free occurrences of *rec* within P represent a new instance of $(\mu\, rec \bullet P)$. A set of events, A, may be "hidden" within a process P, written $P \backslash A$, so that any events in A are not visible externally to P (these become *internal* steps of $P \backslash A$). A state $\sigma \in State$ may be declared local to P via (**state** $\sigma \bullet P$). An interrupt $P \triangle Q$ behaves as P until process Q takes some externally observable action, at which point it "interrupts" P and becomes the active process. A restart process $P\ \mathsf{restart}(a)\ Q$ behaves as P until the restart event a is generated by P, at which point it behaves as $Q\ \mathsf{restart}(a)\ Q$. The restart operator is similar to the *exception* operator of Roscoe [34]. The process *SKIP* has only one possible behaviour, which is to terminate successfully and take no further action. The process *STOP* has no behaviour—it may never synchronise or take any other action. In general, CSP processes may also be parameterised by values, but we do not consider parameters in this paper: a comparison of parameterised values and mutable state is given in [20].

As an example, consider the following specification of a queue process, which makes use of a state, q, which is a sequence of values, where $\langle\rangle$ represents the empty sequence, $\langle v \rangle$ represents the singleton sequence containing v, and $\frown$ represents

[2] In [20] channels could include expressions, but the rules covered only the case where the value of E could be determined locally to the sending process. This paper contains a full treatment.

$P, Q : Process$	$a : \Sigma$	$y : Var$	$v : Val$
$c : SCmd$	$e : \Sigma^{\tau,\checkmark}$	$x, x_1, x_2 : \mathbb{P}\, Var$	$E, R : Expr$
$\sigma : Var \pfun Val$	$\mathsf{id} = [true]$	$A \subseteq \Sigma$	$g : Expr_1$

Fig. 1. Naming conventions.

Rule 1 (*Prefix*).

$$((c, e) \to P) \xrightarrow{c,e} P$$

Rule 2 (*Channel output*).

$$\frac{\mathbf{sat}(R \wedge v = E)}{((x : [R], ch!E) \to P) \xrightarrow{x:[R \wedge v = E], ch.v} P}$$

Rule 3 (*Channel input*).

$$\frac{\mathbf{sat}(R \wedge y' = v)}{((x : [R], ch?y) \to P) \xrightarrow{x,y:[R \wedge y' = v], ch.v} P}$$

Rule 4 (*External choice*).

$$(a)\ \frac{P \xrightarrow{\tau} P'}{P \,[]\, Q \xrightarrow{\tau} P' \,[]\, Q} \qquad (b)\ \frac{P \xrightarrow{c,e} P' \quad (c \not\equiv \mathsf{id} \vee e \neq \tau)}{P \,[]\, Q \xrightarrow{c,e} P'}$$

and similarly for Q.

Rule 5 (*Internal choice*).

$$P \sqcap Q \xrightarrow{\tau} P$$

and similarly for Q.

Rule 6 (*Sequential composition*).

$$(a)\ \frac{P \xrightarrow{c,e} P' \quad e \neq \checkmark}{P \,;\, Q \xrightarrow{c,e} P' \,;\, Q} \qquad (b)\ \frac{P \xrightarrow{\checkmark} P'}{P \,;\, Q \xrightarrow{\tau} Q}$$

Rule 7 (*Skip*).

$$SKIP \xrightarrow{\checkmark} STOP$$

Rule 8 (*Recursion*).

$$(\mu\, rec \bullet P) \xrightarrow{\tau} \left(P \left[\frac{(\mu\, rec \bullet P)}{rec}\right]\right)$$

Rule 9 (*Hiding*).

$$(a)\ \frac{P \xrightarrow{c,a} P' \quad a \in A}{P \backslash A \xrightarrow{c} P' \backslash A} \qquad (b)\ \frac{P \xrightarrow{c,e} P' \quad e \notin A}{P \backslash A \xrightarrow{c,e} P' \backslash A}$$

Rule 10 (*Interrupt*).

$$(a)\ \frac{P \xrightarrow{c,e} P'}{P \triangle Q \xrightarrow{c,e} P' \triangle Q} \qquad (b)\ \frac{Q \xrightarrow{\tau} Q'}{P \triangle Q \xrightarrow{\tau} P \triangle Q'} \qquad (c)\ \frac{Q \xrightarrow{c,e} Q' \quad (c \not\equiv \mathsf{id} \vee e \neq \tau)}{P \triangle Q \xrightarrow{c,e} Q'}$$

Rule 11 (*Restart*).

$$(a)\ \frac{P \xrightarrow{c,e} P' \quad e \neq a}{(P\ \mathsf{restart}(a)\ Q) \xrightarrow{c,e} (P'\ \mathsf{restart}(a)\ Q)} \qquad (b)\ \frac{P \xrightarrow{c,a} P'}{(P\ \mathsf{restart}(a)\ Q) \xrightarrow{c} (Q\ \mathsf{restart}(a)\ Q)}$$

Fig. 2. Rules' summary for CSP_σ.

sequence concatenation. When the event in an action pair is τ, we omit it, and write just the specification command; similarly, when the command is equivalent to id, we omit it.

$$\begin{aligned} &Queue \mathrel{\widehat{=}} (\mathbf{state}\ \{q \mapsto \langle\rangle\} \bullet Qrec) \\ &Qrec \mathrel{\widehat{=}} \mu\, Q \bullet \begin{array}{ll} & (enq?x \to (q := q \frown \langle x \rangle) \to Q) \\ [] & (([q \neq \langle\rangle], deq!head(q)) \to (q := tail(q)) \to Q) \end{array} \end{aligned} \tag{1}$$

After an *enq*?*x* event, *q* is extended by *x* and the process repeats. If *q* is nonempty, *Queue* may participate in a *deq* event, which returns the head of the queue, and then removes it from *q*.

2.2. Semantics

We formally define the meaning of CSP_σ in a structural operational semantics style [35] in Fig. 2, using the variable naming conventions in Fig. 1. The label on each step is a pair containing a command and an event.

2.2.1. Prefixing

A *prefix* is the basic building block of a process. A process may be prefixed by the pairing of a specification command with an event. Rule 1 is straightforward—the process transitions to P, and the command and event (if any) are shown in the transition label. We abbreviate a label in which the command is equivalent to id as just the event, and if the event is τ we abbreviate the label to just the command. This means, for instance, that the rules for a guard and an update with no event are as follows.

$$([g] \to P) \xrightarrow{[g]} P \qquad (x := s \to P) \xrightarrow{x := s} P$$

It is also the case that the usual CSP event prefix rule holds, i.e.,

$$(a \to P) \xrightarrow{a} P$$

2.2.2. Channels

Rules 2 and 3 for channels are more complex, because of the possible interplay of variables in the channel expression with the command. An output $ch!E$ must choose some value for E, say v, and output that value on the channel (within labels, channels must be paired only with values, not expressions). That E does indeed have the value v in context is established by adding a (satisfiable) guard to the command. In the simple case, where a value u is sent on the channel and there is no associated command, we have the rule

$$(ch!u \to P) \xrightarrow{ch.u} P$$

If an expression is sent with no associated command, we have

$$ch!(x + 1) \to P \xrightarrow{[v=x+1],ch.v} P$$

for all possible values v. When the context of a particular state is added (see Section 2.3) this will define the value of x and hence restrict the choice of v to the value of $x + 1$ in that state.

The expression E is of type $Expr_2$, and hence it may reference primed variables (for instance, this allows a command which updates x and outputs its new value on ch). To be consistent, we restrict E to reference primed variables in the frame of the associated command only.

Now consider inputting a value from a channel and storing it in a variable. We do not constrain the input variable to be in or out of the frame of the associated command—either is possible. This allows behaviour such as receiving a new value for a variable only when that value is in a desired set.

When there is no associated command, the effect of a channel input is an update of the receiver variable.

$$(ch?y \to P) \xrightarrow{y := v,ch.v} P$$

Below we demonstrate communication via channels using a simple example.

$$(ch!(x + 1) \to P) \parallel (ch?x \to Q)$$

This command has the effect of incrementing x, with the new value of x sent along channel ch.

For the sending process we have the following possible transition (amongst many).

$$(ch!(x + 1) \to P) \xrightarrow{[1=x+1],ch.1} P$$

The specification command in the label simplifies to $[x = 0]$. This indicates that a visible behaviour of this process is to output the value 1 on channel ch, provided $x = 0$.

For the receiving process we have the following possible transition (amongst many).

$$(ch?x \to Q) \xrightarrow{x:[x'=1],ch.1} Q$$

Combining these transitions with Rule 18 (described later), we have the full transition:

$$(ch!(x + 1) \to P) \parallel (ch?x \to Q) \xrightarrow{c,ch.1} P \parallel Q$$

where $c = x{:}\ [x = 0 \land x' = 1]$.

More generally, we have the following possible transition for any $v \in Val$,

$$(ch!(x + 1) \to P) \parallel (ch?x \to Q) \xrightarrow{c,ch.(v+1)} P \parallel Q$$

where $c = x{:}\ [x = v \land x' = v + 1]$. The context will determine the initial value for x, as described in Section 2.3.

Rule 12 (*State*).

$$\frac{P \xrightarrow{x,y:[R],e} P' \quad x = dom(\sigma_x) \subseteq dom(\sigma) \quad y \cap dom(\sigma) = \varnothing \quad \sigma' = \sigma \oplus \sigma_x \quad \mathbf{sat}(R[\sigma, \sigma'])}{(\mathbf{state}\ \sigma \bullet P) \xrightarrow{y:[R[\sigma,\sigma']],e} (\mathbf{state}\ \sigma' \bullet P')}$$

Rule 13 (*Event in state*).

$$\frac{P \xrightarrow{e} P'}{(\mathbf{state}\ \sigma \bullet P) \xrightarrow{e} (\mathbf{state}\ \sigma \bullet P')}$$

Rule 14 (*Guard*).

$$\frac{P \xrightarrow{[g]} P' \quad \mathbf{sat}(g[\sigma])}{(\mathbf{state}\ \sigma \bullet P) \xrightarrow{[g[\sigma]]} (\mathbf{state}\ \sigma \bullet P')}$$

Rule 15 (*Update - nonlocal*).

$$\frac{P \xrightarrow{x:=E} P' \quad x \notin dom(\sigma)}{(\mathbf{state}\ \sigma \bullet P) \xrightarrow{x:=E[\sigma]} (\mathbf{state}\ \sigma \bullet P')}$$

Rule 16 (*Update - local*).

$$\frac{P \xrightarrow{x:=E} P' \quad x \in dom(\sigma) \quad \sigma' = \sigma \oplus \{x \mapsto v\} \quad \mathbf{sat}(v = E[\sigma])}{(\mathbf{state}\ \sigma \bullet P) \xrightarrow{[v=E[\sigma]]} (\mathbf{state}\ \sigma' \bullet P')}$$

Fig. 3. Rules for local state.

2.2.3. External and internal choice

An external choice between two processes is resolved when one of them makes an observable step, that is, engaging in an event and/or accessing a nonlocal variable. Rule 4(a) allows either process to take an internal step without resolving the choice, while in Rule 4(b) an observable step of either process resolves the choice in that process' favour.

In contrast, an internal choice (Rule 5) is resolved nondeterministically at any time, regardless of the environment.

2.2.4. Sequential composition

The transitions for sequential composition in CSP_σ (Rule 6) are similar to those of CSP. The first process executes its steps (Rule 6(a)), until it terminates (Rule 6(b)), at which point the second process becomes active. The *SKIP* process can do nothing but generate the termination event $\checkmark$ and then take no further action (Rule 7).

2.2.5. Recursion

The transition for a recursive process is to simply unfold the recursion (Rule 8). For instance, recall the definition of *Qrec* (1). An unfolding of *Qrec* results in eliminating the outer μ operator and replacing the recursion variables Q with *Qrec* itself.

$$Qrec \xrightarrow{\tau} \begin{array}{l} (enq?x \rightarrow (q := q \frown \langle x \rangle) \rightarrow Qrec) \\ [] \ (([q \neq \langle\rangle], deq!head(q)) \rightarrow (q := tail(q)) \rightarrow Qrec) \end{array}$$

2.2.6. Hiding

Rule 9(a) removes the event a from the label, that is, it is hidden from the environment. The command c remains observable. If the event part of the label is not hidden, the label does not change (Rule 9(b)).

2.2.7. Interrupt

Rule 10(a) is a typical step of the main process P, while Rule 10(b) is the case where the interrupting process makes an internal step; it may evolve separately to P. Rule 10(c) handles the case where Q makes an externally observable transition to Q': the execution of P is halted, and Q' becomes the active process.

2.2.8. Restart

A *restart process*, $(P\ \mathsf{restart}(a)\ Q)$, acts similarly to an interrupt process $P \triangle Q$, except that the interrupt event, a, is generated *internally* by P. That is, a restart process $(P\ \mathsf{restart}(a)\ Q)$ behaves has P until it generates the (restart) event a, at which time it will halt execution and restart, behaving as $(Q\ \mathsf{restart}(a)\ Q)$. Rule 11($a$) states that P may behave normally as long as it does not generate the event a, while Rule 11(b) states that P terminates and restarts as $Q\ \mathsf{restart}(a)\ Q$ when the event a is generated. The restart operator is similar to the *exception* operator given by Roscoe [34], except that rather than terminating the process we restart it. For brevity, we make the following definition.

$$\mathsf{restart}(a, Q) \mathrel{\widehat{=}} Q\ \mathsf{restart}(a)\ Q \tag{2}$$

2.3. State-based rules

The rules involving the local state construct are given in Fig. 3. Rule 12 covers the general case where, for a process $(\mathbf{state}\ \sigma \bullet P)$, process P transitions with a pair $(x, y{:}\,[R], e)$, where x and y partition the frame into those variables local

to σ and nonlocal to σ, respectively. A new mapping of values in x, σ_x, is nondeterministically chosen so that $R[\sigma, \sigma']$ is satisfiable, where $\sigma' = \sigma \oplus \sigma_x$ is the updated local state containing the new values for x.[3] The visible label on the transition is $(y{:}\,[R[\sigma, \sigma']], e)$, that is, since x is now local, x has been removed from the frame, and references to x and x' in R have been replaced by their values in σ and σ', respectively. The event e is unaffected by the local state.

For example, consider a process P that transitions with a label containing a specification command that updates variables i and j to 0.

$$P \xrightarrow{i,j:[i'=0 \land j'=0]} P'$$

Inside a local state that maps i to the initial value 5, in Rule 12 we instantiate x to $\{i\}$, y to $\{j\}$, σ_x to $\{i \mapsto 0\}$ (which gives $\sigma' = \sigma_x$), and hence

$$\begin{aligned} & R[\sigma, \sigma'] \\ &= (i' = 0 \land j' = 0)[\{i \mapsto 5\}, \{i \mapsto 0\}] \\ &= (0 = 0 \land j' = 0) \\ &= j' = 0 \end{aligned}$$

The substitution serves to eliminate the parts of R that refer to the local state, while the **sat**(..) constraint restricts the post-state σ' to only valid choices of new values. Since **sat**$(j' = 0)$ holds, we may derive the following transition.

$$(\textbf{state}\ \{i \mapsto 5\} \bullet P) \xrightarrow{j:[j'=0]} (\textbf{state}\ \{i \mapsto 0\} \bullet P')$$

Note that any choice for the post-state σ' other than $\{i \mapsto 0\}$ will result in an unsatisfiable predicate, and hence prevent the rule from being applied.

In the case where the local state contains mapping for both i and j, e.g., $\sigma = \{i \mapsto 5, j \mapsto 5\}$, in Rule 12 we instantiate x to $\{i, j\}$, y to $\varnothing$, $\sigma_x (= \sigma')$ to $\{i \mapsto 0, j \mapsto 0\}$, and hence $R[\sigma, \sigma']$ simplifies to *true* and the resulting label is, as expected, id.

In a prefixed process, if the command is equivalent to id, Rule 12 reduces to the simple case of an event, as given in Rule 13.

We now consider specialisations of Rule 12 for guards and updates with no associated events. Rule 14 states that if P transitions with guard g, the observable transition of (**state** $\sigma \bullet P$) is the guard $g[\sigma]$, i.e., the guard g with variables local to σ instantiated with their local values. Below are some examples:

$$(\textbf{state}\ \{i \mapsto 1\} \bullet [i \leqslant 5] \to P) \xrightarrow{\mathsf{id}} (\textbf{state}\ \{i \mapsto 1\} \bullet P) \tag{3}$$

$$(\textbf{state}\ \{i \mapsto 1\} \bullet [i \leqslant x] \to P) \xrightarrow{[1 \leqslant x]} (\textbf{state}\ \{i \mapsto 1\} \bullet P) \tag{4}$$

$$(\textbf{state}\ \{i \mapsto 1\} \bullet [y \leqslant x] \to P) \xrightarrow{[y \leqslant x]} (\textbf{state}\ \{i \mapsto 1\} \bullet P) \tag{5}$$

In (3) the transition label is id, which plays a similar role to τ. The guard trivially evaluates to true in the local state, so to an external observer some internal step is taken. In (4) the guard accesses nonlocal variable x. The externally observable behaviour of this process is that it can evolve to P if $x \geq 1$. The predicate has been partially instantiated according to the local state. In (5) the local state has no effect on the guard: its progress is independent of local variables and hence is externally visible (via the transition label). A process (**state** $\{i \mapsto 1\} \bullet [i > 5] \to P$) cannot transition at all since the guard does not hold in the local context.

Rule 15 states that, for a process (**state** $\sigma \bullet P$), if P makes a transition which updates a nonlocal variable x to E, then the observable transition is an update of x to $E[\sigma]$, that is, the local variables in E are instantiated with their value in σ. For example:

$$(\textbf{state}\ \{i \mapsto 1\} \bullet s := 0 \to P) \xrightarrow{s := 0} (\textbf{state}\ \{i \mapsto 1\} \bullet P) \tag{6}$$

$$(\textbf{state}\ \{i \mapsto 1\} \bullet s := i \to P) \xrightarrow{s := 1} (\textbf{state}\ \{i \mapsto 1\} \bullet P) \tag{7}$$

Transition (6) describes an update to a nonlocal variable, in which the update expression is independent of the local state. In (7) the local state does not include s, but does include a variable in the update expression. Since i is mapped to 1 locally, to an external observer the process appears as an update of s to 1.

Rule 16 states that, for a process (**state** $\sigma \bullet P$), if P makes a transition which updates local variable x, then x is locally updated to a new value v, and the observable transition is a *guard* that ensures v is the value of expression E in context. As such, there are many possible transitions for each local update, one for each value v. However, once placed in a context

[3] The operator '$\oplus$' is function override, that is, given functions f and g, the function $f \oplus g$ maps inputs according to g for elements in the domain of g, and all others according to f, that is, $(f \oplus g)(x) = g(x)$ if $x \in \mathrm{dom}(g)$, and $f(x)$ otherwise.

Rule 17 (*Parallel independent*).

$$\frac{P \xrightarrow{c,e} P' \quad e \notin A \cup \{\checkmark\}}{P \underset{A}{\|} Q \xrightarrow{c,e} P' \underset{A}{\|} Q}$$

and similarly for Q.

Rule 18 (*Parallel synchronise*).

$$\frac{P \xrightarrow{x_1:[R_1],a} P' \quad Q \xrightarrow{x_2:[R_2],a} Q' \quad \mathbf{sat}(R_1 \wedge R_2) \quad a \in A}{P \underset{A}{\|} Q \xrightarrow{x_1,x_2:[R_1 \wedge R_2],a} P' \underset{A}{\|} Q'}$$

Rule 19 (*Parallel – terminate one*).

$$\frac{P \xrightarrow{\checkmark} P'}{P \underset{A}{\|} Q \xrightarrow{\tau} STOP \underset{A}{\|} Q}$$

and similarly for Q.

Rule 20 (*Parallel – terminate both*).

$$STOP \underset{A}{\|} STOP \xrightarrow{\checkmark} STOP$$

Fig. 4. Rules for parallel composition.

which defines all the free variables in E, only the transition in which v has the value for E in that state will be valid. For example:

$$(\mathbf{state}\ \{s \mapsto 1\} \bullet s := 0 \to P) \xrightarrow{\mathrm{id}} (\mathbf{state}\ \{s \mapsto 0\} \bullet P) \tag{8}$$

$$(\mathbf{state}\ \{s \mapsto 1\} \bullet s := s + i \to P) \xrightarrow{[i=0]} (\mathbf{state}\ \{s \mapsto 1\} \bullet P) \tag{9}$$

$$(\mathbf{state}\ \{s \mapsto 1\} \bullet s := s + i \to P) \xrightarrow{[i=1]} (\mathbf{state}\ \{s \mapsto 2\} \bullet P) \tag{10}$$

$$(\mathbf{state}\ \{s \mapsto 1\} \bullet s := s + i \to P) \xrightarrow{[i=2]} (\mathbf{state}\ \{s \mapsto 3\} \bullet P) \tag{11}$$

Transition (8) is a simple example of the application of Rule 16 where we make the obvious choice of 0 for v, since $E[\sigma]$ evaluates to 0, and therefore $[E[\sigma] = v] = [0 = 0] \equiv \mathrm{id}$. The remaining transitions deal with the more complex case where the updated variable is local but the expression E is not. In these cases we cannot locally determine the value to which s must be updated, since the update expression accesses the nonlocal variable i. Locally, therefore, there are many possible transitions, one for each $v \in Val$ to which s can be updated (we have shown only the transitions for $v = 1$, $v = 2$, $v = 3$). However, in practice, only one transition will be possible for a given context. In this case, that will be the transition in which v has the value of $1 + i$ in that context.

2.4. Interface parallel

The parallel operator used in this paper is based on the interface parallel operator described by Roscoe [11], rather than Hoare's original alphabetised parallel [10] (which was used in [20]). The rules for interface parallel are given in Fig. 4. For interface parallel, the interface A defines the set of events on which the two processes must synchronise. Note that A cannot include the special events τ or $\checkmark$.

Rule 17 states that process P may evolve to P' independently of Q provided that the event that P is engaging in is not a member of the interface A, and that P is not terminating.

Rule 18 handles the more interesting case where both P and Q are ready to engage in a shared event a which is in the interface A. In this case the associated specification commands are conjoined, provided that the conjunction is satisfiable. Note that this rule allows x_1 and x_2 to overlap, and hence finds a mapping for variables in the intersection that satisfies both R_1 and R_2, should such a mapping exist. However, this admits rather subtle behaviour and semantics; generally, it is safer to prevent synchronised specification commands from modifying the same variable, and this constraint may be enforced statically.

Termination of a parallel composition of processes requires all processes to have terminated, i.e., distributed termination. Rule 19 handles the case where one of the processes terminates: the terminated process is replaced by *STOP*, but this appears as an internal step of the parallel composition. In Rule 20 both processes have terminated, in which case the parallel composition itself visibly terminates.

2.5. Examples

In Fig. 5 we present two executions side-by-side. For space reasons we abbreviate the **state** keyword to **st**, and use $\overset{l}{\Rightarrow}$ to indicate a sequence of two or more transitions that contain exactly one non-internal step, l. Often, the omitted τ steps include the initial unfolding of a recursion.

$$
\begin{array}{ll}
 & \mathit{Queue} \\
\widehat{=} & \text{from (1)} \\
 & (\textbf{state}\ \{q \mapsto \langle\rangle\} \bullet \mathit{Qrec}) \\
\xRightarrow{enq.v} & \text{Rule 8, Rule 1, Rule 4, Rule 13} \\
 & (\textbf{state}\ \{q \mapsto \langle\rangle\} \bullet q := q \frown \langle v\rangle \to \mathit{Qrec}) \\
\xrightarrow{\mathsf{id}} & \text{Rule 1, Rule 16} \\
 & (\textbf{state}\ \{q \mapsto \langle v\rangle\} \bullet \mathit{Qrec}) \\
\xRightarrow{enq.w} & \text{Rule 8, Rule 1, Rule 4, Rule 13} \\
 & (\textbf{state}\ \{q \mapsto \langle v\rangle\} \bullet q := q \frown \langle w\rangle \to \mathit{Qrec}) \\
\xrightarrow{\mathsf{id}} & \text{Rule 1, Rule 16} \\
 & (\textbf{state}\ \{q \mapsto \langle v, w\rangle\} \bullet \mathit{Qrec}) \\
\xRightarrow{deq.v} & \text{Rule 8, Rule 1, Rule 4, Rule 12 } (q \neq \langle\rangle) \\
 & (\textbf{state}\ \{q \mapsto \langle v, w\rangle\} \bullet q := \mathit{tail}(q) \to \mathit{Qrec}) \\
\xrightarrow{\mathsf{id}} & \text{Rule 1, Rule 16} \\
 & (\textbf{state}\ \{q \mapsto \langle w\rangle\} \bullet \mathit{Qrec}) \\
\xRightarrow{deq.w} & \text{Rule 8, Rule 1, Rule 4, Rule 12 } (q \neq \langle\rangle) \\
 & (\textbf{state}\ \{q \mapsto \langle w\rangle\} \bullet q := \mathit{tail}(q) \to \mathit{Qrec}) \\
\xrightarrow{\mathsf{id}} & \text{Rule 1, Rule 16} \\
 & (\textbf{state}\ \{q \mapsto \langle\rangle\} \bullet \mathit{Qrec})
\end{array}
$$

(a) Execution of *Queue*

$$
\begin{array}{ll}
 & \mathit{Sum} \\
\widehat{=} & \text{from (12)} \\
 & (\textbf{state}\ \{i \mapsto 1, s \mapsto 0\} \bullet S) \\
\xRightarrow{[1 \leqslant N]} & \text{Rule 8, Rule 1, Rule 4, Rule 14} \\
 & (\textbf{state}\ \{i \mapsto 1, s \mapsto 0\} \bullet \\
 & \quad (s := s + i) \to (i := i + 1) \to S) \\
\xrightarrow{\mathsf{id}} & \text{Rule 1, Rule 16} \\
 & (\textbf{state}\ \{i \mapsto 1, s \mapsto 1\} \bullet (i := i + 1) \to S) \\
\xrightarrow{\mathsf{id}} & \text{Rule 1, Rule 16} \\
 & (\textbf{state}\ \{i \mapsto 2, s \mapsto 1\} \bullet S) \\
\xRightarrow{[2 \leqslant N]} & \text{Rule 8, Rule 1, Rule 4, Rule 14 (as above)} \\
 & (\textbf{state}\ \{i \mapsto 2, s \mapsto 1\} \bullet \\
 & \quad (s := s + i) \to (i := i + 1) \to S) \\
\xRightarrow{\mathsf{id}} & \text{Rule 1, Rule 16 } (\times 2 \text{ as above}) \\
 & (\textbf{state}\ \{i \mapsto 3, s \mapsto 3\} \bullet S) \\
\xRightarrow{[3 > N]} & \text{Rule 8, Rule 1, Rule 4, Rule 14} \\
 & (\textbf{state}\ \{i \mapsto 3, s \mapsto 3\} \bullet x := s \to \mathit{SKIP}) \\
\xrightarrow{x := 3} & \text{Rule 1, Rule 15} \\
 & (\textbf{state}\ \{i \mapsto 3, s \mapsto 3\} \bullet \mathit{SKIP}) \\
\xrightarrow{\checkmark} & \text{Rule 7, Rule 13} \\
 & (\textbf{state}\ \{i \mapsto 3, s \mapsto 3\} \bullet \mathit{STOP})
\end{array}
$$

(b) Execution of *Sum* where $N = 2$ in context

Fig. 5. Example executions.

Fig. 5(a) contains an execution of the process Q from (1) when q is initially empty. Fig. 5(b) contains the execution of program *Sum*, which is an example of how computation sequences may be specified in CSP_σ.

$$
\begin{array}{l}
\mathit{Sum} \mathrel{\widehat{=}} (\textbf{state}\ \{i \mapsto 1, s \mapsto 0\} \bullet S) \\
S \mathrel{\widehat{=}} \mu\, \mathit{sum} \bullet \begin{array}{l} [i \leqslant N] \to (s := s + i) \to (i := i + 1) \to \mathit{sum} \\ []\ [i > N] \to (x := s) \to \mathit{SKIP} \end{array}
\end{array}
\tag{12}
$$

Sum calculates the sum to the value of (nonlocal) variable N, and writes the final value to (nonlocal) variable x. Variable N is set prior to the invocation of *Sum*, and x is read after *Sum* terminates to retrieve the result. An alternative specification would be to parameterise *Sum* by N, and to output the result on a channel, however, for illustrative purposes we have chosen the former approach. The relative merits of shared-variable communication and channel-based communication are explored in more detail in [20]. The definition of *Sum* uses local variables s and i to accumulate progressive values.

The trace of the execution of *Sum* is an interleaving of internal steps (id) with accesses of nonlocal variable N, which in the figure is assumed to have the value 2 in context, until the final observable transition which updates x to 3. No more transitions are possible.

3. Message passing

In this section we introduce a new message passing construct for CSP in which the sender does not need to block until there is a receiver. Such a construct more naturally represents some communications in certain systems; for example, a taxi company headquarters sending notification of a new job. In addition, it allows the number of potential listeners to change dynamically, as is common in systems of many interacting autonomous agents. It follows the publish/subscribe model of communication [12]. Another strength of the new construct is that it is easy to specify multiple sending processes, which do not interact (need to synchronise) with each other.

We are interested in the semantics of this construct because, as outlined above, some systems more naturally use this form of communication over the more abstract, and harder to implement, synchronisation construct of CSP. Therefore when translating from natural language requirements this type of communication model will be easier to apply in some situations.

The syntax for sending and receiving is given below.

$$\mathsf{send}\ m.E \qquad \mathsf{recv}\ m.y$$

Messages, as with channels, may send expressions and be paired with specification commands. The send action communicates message m, with optional expression E, while the recv action receives message m, storing any associated value in variable y.

Rule 21 (*Message send*).

$$\frac{\mathbf{sat}(R \wedge v = E)}{((x{:}\,[R],\, \mathsf{send}\, m.E) \to P) \xrightarrow{x:[R \wedge v = E],\, \mathsf{send}\, m.v} P}$$

Rule 22 (*Message receive*).

$$\frac{\mathbf{sat}(R \wedge y' = v)}{((x{:}\,[R],\, \mathsf{recv}\, m.y) \to P) \xrightarrow{x,y:[R \wedge y' = v],\, \mathsf{recv}\, m.v} P}$$

Rule 23 (*Message sent*).

$$\frac{P \xrightarrow{x_1:[R_1],\, \mathsf{send}\, m.v} P' \quad Q \xrightarrow{x_2:[R_2],\, \mathsf{recv}\, m.v} Q' \qquad \mathbf{sat}(R_1 \wedge R_2)}{P \underset{A}{\|} Q \xrightarrow{x_1, x_2:[R_1 \wedge R_2],\, \mathsf{send}\, m.v} P' \underset{A}{\|} Q'}$$

and similarly with P and Q swapped.

Rule 24 (*Message ignored*).

$$\frac{P \xrightarrow{c,\, \mathsf{send}\, m.v} P' \quad Q \xrightarrow{\mathsf{recv}\, m.v} \!\!\!\!\!/}{P \underset{A}{\|} Q \xrightarrow{c,\, \mathsf{send}\, m.v} P' \underset{A}{\|} Q}$$

and similarly for Q.

Rule 25 (*Multiple listeners*).

$$\frac{P \xrightarrow{x_1:[R_1],\, \mathsf{recv}\, m.v} P' \quad Q \xrightarrow{x_2:[R_2],\, \mathsf{recv}\, m.v} Q' \qquad \mathbf{sat}(R_1 \wedge R_2)}{P \underset{A}{\|} Q \xrightarrow{x_1, x_2:[R_1 \wedge R_2],\, \mathsf{recv}\, m.v} P' \underset{A}{\|} Q'}$$

Rule 26 (*Single listener*).

$$\frac{P \xrightarrow{c,\, \mathsf{recv}\, m.v} P' \quad Q \xrightarrow{\mathsf{recv}\, m.v} \!\!\!\!\!/}{P \underset{A}{\|} Q \xrightarrow{c,\, \mathsf{recv}\, m.v} P' \underset{A}{\|} Q}$$

and similarly for Q.

Rule 27 (*Hide messages*).

$$(a)\ \frac{P \xrightarrow{c,a} P' \quad a \in A \quad (\forall m, v \bullet a \neq \mathsf{recv}\, m.v)}{P \backslash A \xrightarrow{c} P' \backslash A} \qquad (b)\ \frac{P \xrightarrow{c,e} P' \quad e \notin A}{P \backslash A \xrightarrow{c,e} P' \backslash A}$$

Fig. 6. Rules for messages.

The rules for messages and parallel composition are given in Fig. 6; they follow a similar pattern to the synchronisation event rules. We write messages as message/value, message/variable, or message/expression pairs ($m.v$, $m.y$ or $m.E$), although the rules equally apply to basic messages (m).

Rules 21 and 22 correspond to sending the value of an expression via a channel in Rules 2 and 3. Rule 23 captures P sending a message to Q. The visible behaviour is the conjunction of their respective specification commands, and that $m.v$ is sent. Rule 24 states that P can still send m even if Q is not waiting: this is the nonblocking nature of sending a message. We use the notation $P \xrightarrow{\ell} \!\!\!\!\!/$ to indicate that there exists no process P' such that $P \xrightarrow{\ell} P'$. Rule 25 states that two receiving processes can receive the same message, while Rule 26 allows a single process to receive a message if the other is not listening.

Note that a listening process can respond directly to a send, acting as process Q in Rule 23, or ignore the send and propagate its recv, acting as process P in Rule 26. This nondeterminism arises because there may be more than one producer sending a message, and a listener is free to react to either of them. To prevent traces where a receiver responds to an 'external' message, the scope over which the message is listened for must be limited (hidden) in the usual way. For instance, the smallest common ancestor of both sender and receiver will typically hide the receive message.

The rules in Fig. 6 apply only to messages, while the rules in Fig. 4 apply only to events and channels.[4] However, the majority of rules from Figs. 2 and 3 hold for both events and messages, where we allow e to range over messages as well as events. The only exception is Rule 9 for hiding, which requires special treatment for receiving messages, and is now replaced by Rule 27. Rule 27(a) is similar to Rule 9(a), except that it applies to synchronisation events and sent messages, but not to receiving messages. To allow receive actions to become internal events through hiding would be to allow them to transition without a corresponding send action. Hence, instead of becoming an internal step, a hidden receive message is prevented from transitioning at all. Rule 27(b) is identical to Rule 9(b) (it applies to synchronisation, send, and recv actions).

3.1. Example 1

Consider the following simple process S that sends message m, and process L that contains two concurrent processes listening for that same message.

[4] Note that we do not examine the interface of the parallel composition for messages, that is, we assume that all messages of the same name within a system are designed to interact. The rules may be rewritten so that the interfaces are consulted.

$$S \mathrel{\widehat{=}} (\mathsf{send}\, m \to P) \qquad L \mathrel{\widehat{=}} (\mathsf{recv}\, m \to Q \parallel \mathsf{recv}\, m \to R)$$

Before progressing we first note the following specialisations of Rules 21 and 22 if there is no associated specification command or expression.

Rule 28 (*Message-only Send/Receive*)**.**

$$(\mathsf{send}\, m \to P) \xrightarrow{\mathsf{send}\, m} P \qquad (\mathsf{recv}\, m \to P) \xrightarrow{\mathsf{recv}\, m} P$$

Through Rules 28 and 25 we have the following transitions

$$S \xrightarrow{\mathsf{send}\, m} P \qquad L \xrightarrow{\mathsf{recv}\, m} (Q \parallel R)$$

Then through Rule 23 we have

$$S \parallel L \xrightarrow{\mathsf{send}\, m} P \parallel (Q \parallel R)$$

Now consider two competing senders, S_1 and S_2, operating in parallel.

$$S_1 \mathrel{\widehat{=}} \mathsf{send}\, m \to P_1 \qquad S_2 \mathrel{\widehat{=}} \mathsf{send}\, m \to P_2$$

Through Rule 24 only one of these process will send a message—they do not synchronise. Either of the following two transitions are allowed by the rules.

$$(S_1 \parallel S_2) \parallel L \xrightarrow{\mathsf{send}\, m} (P_1 \parallel S_2) \parallel (Q \parallel R)$$

$$(S_1 \parallel S_2) \parallel L \xrightarrow{\mathsf{send}\, m} (S_1 \parallel P_2) \parallel (Q \parallel R)$$

As a final example, consider a variant of L in which one of the processes, T, is not yet ready to receive the message.

$$L \mathrel{\widehat{=}} (\mathsf{recv}\, m \to Q) \parallel T \quad \text{where } T \xrightarrow{\mathsf{recv}\, m} \!\!\!\!/$$

We have the following transition.

$$S \parallel L \xrightarrow{\mathsf{send}\, m} P \parallel (Q \parallel T)$$

Although T is a process that may eventually listen for message m, the fact it is not yet ready does not block S from sending the message to process Q.

3.2. Example 2

We now give a more complex example that combines state tests and updates with message passing. Consider a network which consists of producers and consumers. Producers periodically send information, which is conditionally received by all ready consumers. We treat the sent information abstractly, although it may be, for instance, (cumulative) security or database updates, etc. The conditions under which consumers receive the information are also treated abstractly, as are their general tasks. Consumers may also be turned on and off at any time.

For simplicity we write a recursive procedure

$$P \mathrel{\widehat{=}} (\mu\, c \bullet .. \to c) \quad \text{as } P \mathrel{\widehat{=}} .. \to P$$

We use this abbreviation as it simplifies the syntax and hides the unfolding steps (Rule 8) in the executions. However, the derivations we give may be easily transformed to use the least-fixpoint syntax.

$$\begin{aligned} P_i &\mathrel{\widehat{=}} in_i?d \to \mathsf{send}\, update.f(d) \to P_i \\ C_i &\mathrel{\widehat{=}} (\mathbf{state}\ \{y \mapsto 0\} \bullet Wk_i \parallel Upd_i) \triangle Rbt_i \\ Wk_i &\mathrel{\widehat{=}} .. \\ Upd_i &\mathrel{\widehat{=}} (y\colon [Test], \mathsf{recv}\, update.y) \to Upd_i \\ Rbt_i &\mathrel{\widehat{=}} off_i \to on_i \to C_i \end{aligned}$$

A producer P_i receives input data d from the environment along input channel in_i, then sends some calculated value, $f(d)$, to every consumer on the network. This is done repeatedly.

A consumer C_i has a local state, which is treated abstractly as variable y. Consumers are assumed to be initially active, and performing some work tasks given by the process, Wk_i, which we leave unspecified. In parallel, the consumer is always ready to receive updates to y, given by the recursive process Upd_i, which receives updates from the producers provided two-state predicate *Test* holds (defined below), and stores the received value in y. At any time the consumer may be switched off by the event off_i, which interrupts the working behaviour of the consumer. When the consumer is switched back on (on_i), the consumer restarts. This "rebooting" behaviour is given by Rbt_i.

The relevant point is that this specification would be cumbersome to define using CSP-like synchronisation only. The sending of updates by the producers is not held up by the transient consumers, which may come online and offline at any time. Note also that C_i only conditionally receives messages, i.e., those that satisfy *Test*. Since *Test* may refer to both pre- and post-values of y, complex relationships between the current value of y and the value received via *update* messages may be specified. For instance, for the purposes of the example, let us assume that there are three possible values for y, u_1, u_2 and u_3, and define *Test* such that if $y = u_1$ it may be updated to u_2 only, if $y = u_2$ it may be updated to u_1 or u_3, and if $y = u_3$ it may be updated arbitrarily.

$$(y = u_1 \land y' = u_2) \lor (y = u_2 \land y' \neq u_2) \lor y = u_3 \tag{13}$$

In general there may be M producers and N consumers, but for presentation purposes we demonstrate the execution of the send/receive behaviour assuming that there is a single producer, P_1, and three consumers, C_1, C_2 and C_3. Let us assume that C_1 and C_2 are active, and that C_3 has been switched off, and let C_i' represent C_i after an unspecified number of steps.

$$C_1' \mathrel{\widehat{=}} (\textbf{state}\ \{y \mapsto u_1\} \bullet Wk_1' \parallel Upd_1) \triangle Rbt_1$$
$$C_2' \mathrel{\widehat{=}} (\textbf{state}\ \{y \mapsto u_2\} \bullet Wk_2' \parallel Upd_2) \triangle Rbt_2$$
$$C_3' \mathrel{\widehat{=}} on_3 \rightarrow C_3$$

The values u_i are the local values of y, and the processes Wk_1' and Wk_2' represent the current stage of execution of the working tasks for C_1 and C_2. Process C_3 is suspended until the event on_3 occurs.

Let P_1' be the producer process after it has received the event $in_1.v$ and as a result is about to send the value u_2 $(= f(v))$. Then we have the following transition by Rule 21.

$$P_1' \xrightarrow{\text{send } update.u_2} P_1 \tag{14}$$

The system process *Sys* is defined as the parallel composition of the producer and the three consumers.

$$Sys \mathrel{\widehat{=}} P_1' \parallel ((C_1' \parallel C_2') \parallel C_3')$$

We leave the alphabets on the parallel composition implicit: they do not directly affect the messages.

The consumers C_1 and C_2 are actively listening for update messages through the Upd_i process. For instance, for Upd_1, we have the following transition by Rule 22.

$$Upd_1 \xrightarrow{y:[Test \land y' = u_2],\ \text{recv } update.u_2} Upd_1$$

By Rule 12 we have

$$C_1' \xrightarrow{update.u_2} \textbf{state}\ \{y \mapsto u_2\} \bullet ..$$

provided $\textbf{sat}(Test \land y' = u_2)$. By (13) this is implied by $y = u_1$, which holds in this case. However, for process C_2', where $y = u_2$, the satisfiability test does not hold, and hence the transition is not possible. Also note that C_3' is not ready to receive the update message as it is switched off. Then we have:

$$C_1' \xrightarrow{\text{recv } update.u_2} C_1'' \tag{15}$$

$$C_2' \xrightarrow{\text{recv } update.u_2} \not\quad \tag{16}$$

$$C_3' \xrightarrow{\text{recv } update.u_2} \not\quad \tag{17}$$

By (15), (16) and Rule 24 we have

$$C_1' \parallel C_2' \xrightarrow{\text{recv } update.u_2} C_1'' \parallel C_2'$$

Hence by this, (17) and Rule 24,

$$(C_1' \parallel C_2') \parallel C_3' \xrightarrow{\text{recv } update.u_2} (C_1'' \parallel C_2') \parallel C_3'$$

Finally, by this, (14) and Rule 23, we have

$$Sys \xrightarrow{\text{send } update.u_2} P_1 \parallel (C_1'' \parallel C_2') \parallel C_3'$$

The net transition is that the producer P_1 has sent the message to C_1, while C_2 and C_3 have ignored the message, for different reasons. This type of selective communication is more difficult to specify using only CSP-like synchronisation.

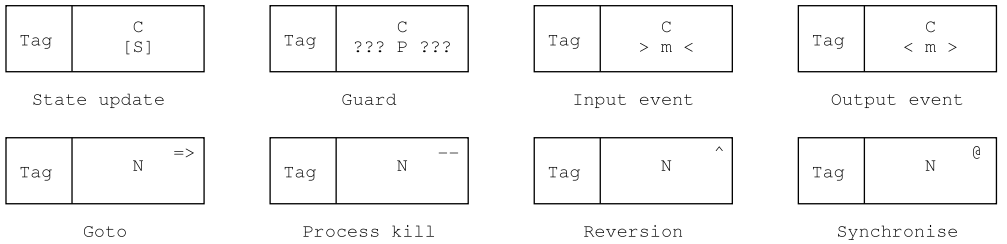


Fig. 7. Behavior Tree nodes.

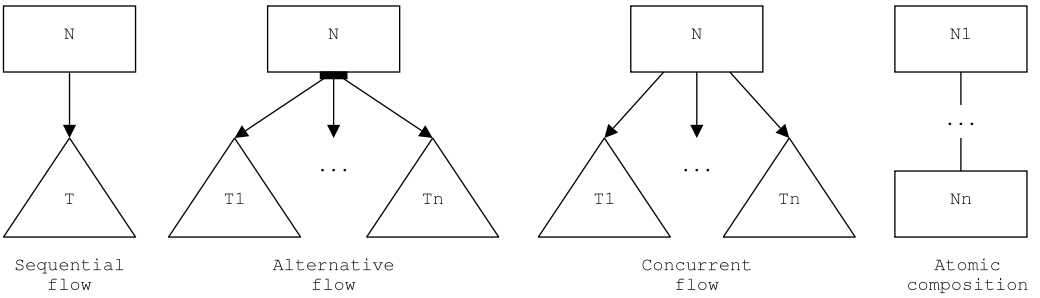


Fig. 8. Behavior Tree constructors.

3.3. Related work: message passing

The motivation for the message passing scheme defined in this section is in modelling networks with transient agents, where other communication schemes such as shared variable and synchronisation are also required. The main point of technical difference with CSP synchronisation is that the sender is never blocked waiting for a receiver.

A related communication mechanism is the *barrier synchronisation* of Occam [36], where processes can dynamically register an interest in a barrier, and all such processes are blocked until every other registered process is ready. In that scheme there is no explicit sender, and as such is closer to CSP synchronisation. Other situations in which message passing frameworks must be combined with state-based constructs include security protocols, as explored, for instance, by Chevalier et al. [37,38]. Cardelli and Gordon explore the concept of ambient processes more abstractly in [39].

4. Behavior Trees

In this section we provide a brief and informal description of Behavior Trees and the method for developing specifications from requirements; more detail on Behavior Trees, and the motivation for them, is available elsewhere [1–3]. The Behavior Tree notation as presented in [1] also includes other constructs, which may be mapped into the basic primitives we give here.

4.1. Notation and informal description

Nodes. The Behavior Tree node types are given in Fig. 7. Each node refers to a specific component (C), and describes some operation involving that component. A *state update* node updates the state of C to some expression S, while a *guard* node blocks until predicate P is satisfied by C's state. The full Behavior Tree notation includes many types of nodes and node combinators for expressing predicates on the state and for updates, but in this paper we have used generalised "state updates" and "guard" nodes. It is straightforward to map the original Behavior Tree nodes and node combinators into our more general node.

In addition to these state-based nodes, the notation includes message-based communication. An *output event* node indicates that C generates message m (possibly with a list of values). The reciprocal *input event* node blocks until C receives message m (storing the passed values (if any) into a list of variables).

The bottom line of Fig. 7 gives four node modifiers, which operate on some node N: a well-formed tree will therefore contain a node N at some other place.[5] A *goto* node indicates that the subsequent behaviour should be that of the subtree rooted at node N. Any such tree must appear in an alternative branch. Typically *goto* nodes (and *reversion* nodes, see below) are leaf nodes. *Goto* nodes are used as a shorthand if the same behaviour occurs in different parts of the tree. A *process kill* node terminates any behaviour associated with the tree rooted at node N. The target node must appear in a concurrent branch. A *reversion node* allows iteration. A well-formed Behavior Tree will have a node N as an ancestor of the reversion

[5] This well-formedness condition, and others described later, can be checked syntactically [8].

node, and any behaviour associated with the tree rooted at N is restarted from that point. A *synchronisation* node indicates participation in a synchronisation event. Each process synchronising on N blocks until all other such processes are also at the synchronisation node, at which time they may all progress and N is executed.

Each node has an associated *tag*, which is used for traceability. The tag records from which of the original informal requirement(s) the node originates. This allows tracking of requirements and facilitates requirements change. In addition, nodes can be colour coded, to indicate where the developer has introduced assumptions/behaviour, or modified/removed behaviour with respect to the original requirements.

Constructors. A Behavior Tree has one of the four forms in Fig. 8: *sequential flow*, *alternative flow*, *concurrent flow* or *atomic composition*. A sequential flow of a node N with a tree T indicates simple ordering on node execution: node N is executed, after which T is ready for execution. Because several trees may be executing in parallel, it is possible that the behaviour of other nodes will be interleaved before and after N.

Alternative flow indicates that one of $T_1, \ldots, T_n$ will be executed after N, depending on which are enabled. If exactly one is enabled, that *tree* is executed, and if none are enabled, execution blocks until at least one of them is ready to execute (e.g., by the reception of a message). If more than one are enabled, a nondeterministic choice is made as to which is executed.

A concurrent flow from node N to a set of trees, $T_1, \ldots, T_n$, indicates that after N is executed, all of the trees are ready for execution.

An atomic composition of nodes $N_1, \ldots, N_n$ indicates that there is no opportunity for processes operating in parallel to interleave their actions during the execution of the N_is. Therefore the nodes operate together in a single atomic action, with the order of execution being sequentially from N_1 to N_n. Atomic composition is distinguished graphically from sequential flow by omitting the arrowhead on the connecting line. To be well-formed, an atomic composition of nodes must contain at most one node with a non-internal event, i.e., at most one node of type input/output event, process kill, reversion, and synchronisation. An atomic composition of nodes may take the place of a single node in the other three constructors.

4.2. Application

The Behavior Tree method is designed for translating a requirements document, in which each requirement is numbered, into a structured model. The first step is to systematically translate each individual requirement into a Behavior Tree and record the requirement number in the tag. In addition, nodes are coloured if the developer believes them to contain some sort of defect, e.g., redundancy, incompleteness, ambiguity. Problems commonly arise with the use of an inconsistent vocabulary, as can be introduced in documents where multiple authors use different terms to represent the same concept, or, more insidious, use the same term to refer to different concepts.

The process of developing a Behavior Tree can be divided amongst a group of people who work in parallel. The trees are then *integrated* by identifying syntactically matching nodes, and joining them appropriately. The tags are also merged in the joining nodes, serving to highlight the overlapping nature of the requirements. The resulting structure helps to identify errors in the requirements, and the result is a single Behavior Tree which describes the system as a whole.

The process has the benefit that it can be initially split amongst developers working largely independently. The combination of tagging and colour coding means that clients can use the Behavior Tree model to quickly find problems with the requirements and compare them to the original document. We demonstrate the approach more fully with an example.

4.3. Example

In this section we show an example of how the Behavior Tree notation is used to construct a specification from natural language requirements. For presentation purposes we give a partial specification of a controller system, with the intention of showing the type of systems and requirements for which the Behavior Tree notation is designed.

4.3.1. The system and its requirements

Consider an abstract system, which is comprised of a *Control* component with two buttons, and a *Sensor* component. The behaviour of the system is given by the following requirements:

R1. After performing tasks required for initialisation, the *Control* component becomes ready and the *Sensor* can detect errors.

R2. When the *Sensor* detects an error, it tells the *Control* to halt.

R3. After telling the *Control* to halt, the *Sensor* waits until the *Control* is ready before trying to detect further errors.

R4. After the *Control* component is ready, if button 1 is pressed the *Control* component becomes active.

R5. While the *Control* component is active, if button 1 is pressed it enters mode A, or if button 2 is pressed it enters mode B.

R6. At any time after the *Control* component has become ready, if a halt message is received, the *Control* goes into shutdown mode before returning to the ready state.

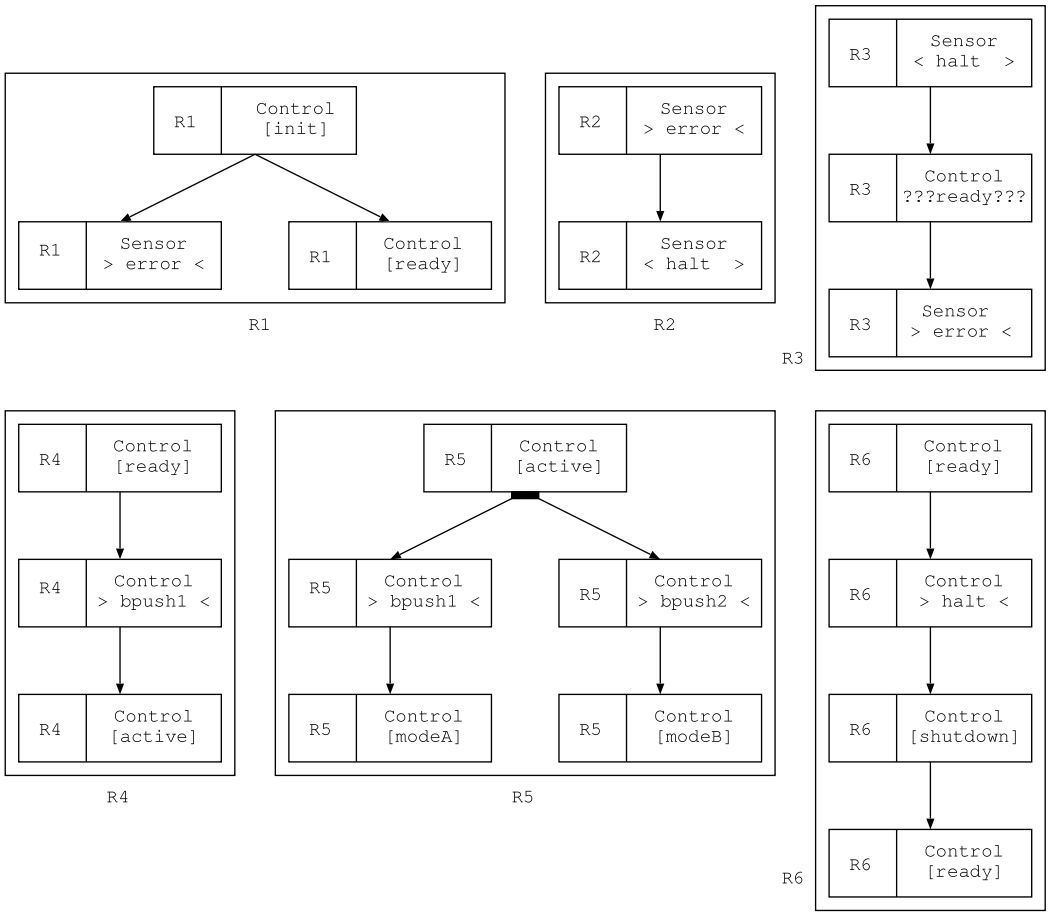


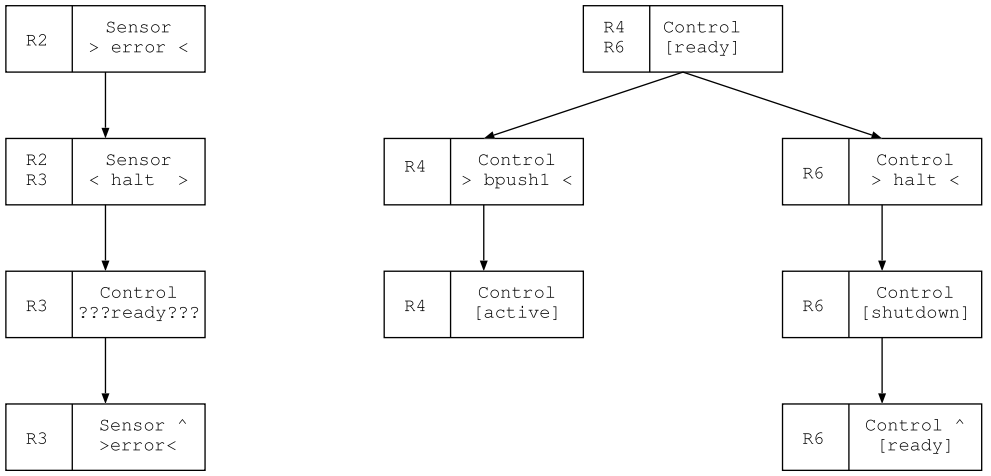
Fig. 9. Individual requirements.

4.3.2. Individual requirements translation

The translation of the individual requirements are given in Fig. 9. The translation process serves to compile a vocabulary of component names, component states, messages, and events, which are collected in the *Composition Tree*.[6] In translating Requirement R1, we have abstracted the initialisation tasks as the state *init*, and used *ready* for the subsequent state. The *Sensor* and *Control* are given as parallel subtrees, as it appears they are intended to operate concurrently. We use *error* as the message name for detecting error events from the environment. Requirement R2 is translated as a sequential flow, such that after the *error* event is detected, the sensor sends the *halt* message, which is a message used exclusively for communication between the *Sensor* and the *Control*. Requirement R3 is also translated using sequential flow, making use of a guard node to test the state of *Control*. An alternative to using a guard node to model the "wait" is to *synchronise* with the *Control* process on its readiness, and thus move from shared-variable communication to synchronised communication. We explore this alternative further in Section 5.4.2. Once the *Control* is known to be in the *ready* state, the *Sensor* returns to its earlier behaviour of waiting for an error event. This will likely become a reversion node, since it is repeat behaviour, but this will be resolved during the integration phase. Requirement R4 and Requirement R6 are translated similarly. Requirement R5 is translated using alternative choice between the two input events. Once one of the buttons is pressed, the *Control* enters the corresponding mode and will not leave it (unless the system is restarted).

Several issues are raised during the translation, for instance, in Requirement R6, is the *halt* message only of relevance while the *Control* is in the state *ready*, or should the same behaviour follow even if it has progressed to state *active*? For this example, we have assumed the *Control* will be shutdown anytime it receives a *halt* message. Another issue is what happens if the second button is pressed while the *Control* is ready? We have assumed that the second button is ignored unless *Control* is active. These are exactly the types of issues that Behavior Tree modelling is intended to highlight; while such issues will be raised whichever modelling language or approach is used, using Behavior Trees the task is systematic. If the modeller notes choices and assumptions using the colour coding, they can be traced back to the original requirements document using the tags, facilitating discussion with the originator of the requirements as the model develops.

[6] As mentioned earlier, we do not consider the static information contained in Composition Trees in this paper.



(a) Integration of Requirements R2 and R3. (b) Integration of Requirements R4 and R6.

Fig. 10. Partial integration of requirements.

4.3.3. Requirements integration

We now integrate the individual trees to construct a full view of the system. The integration process is based on finding syntactically matching nodes, and completing other aspects of the tree such as reversion. Although this process is systematic, and therefore potentially amenable to tool support, there are still choices for the modeller to make. For instance, if the integration results in a branch, a decision must be made as to whether the branch should be a parallel or alternative choice; the order in which trees are integrated will also effect the shape of the final tree [40]. The contribution of the Behavior Tree modelling approach is not to eliminate such choices, but to make the choice explicit and traceable in the model.

Requirements R2 and R3 integrate end-to-end, since they share a common (joining) node. Furthermore, because the leaf node `Sensor > error <` matches one of its ancestors, it is flagged as a reversion node. The resulting tree, which captures the behaviour of the *Sensor* component, is shown in Fig. 10(a). Note that the joining node references the tags for both requirements. Fig. 10(b) shows the result of integrating Requirements R4 and R6. In this case the integration point (node `Control [ready]`) is the root node of both trees. We must decide whether the resulting branches are composed in parallel or as alternatives. As mentioned above, we assume that the halt message is always of relevance, and hence is not affected by the subsequent behaviour of *Control*; hence we integrate the trees using parallel composition. The `Control [ready]` leaf node is also flagged as a reversion.

It remains to integrate Requirement R5 with Fig. 10(b), and to integrate both trees with Requirement R1, both of which tasks are straightforward. The resulting Behavior Tree, giving the complete behaviour of the system, is shown in Fig. 11.

The system has been built out of its natural language requirements in a straightforward manner. Even though the requirements we have given are contrived for simplicity, they demonstrate that even in simple systems there is considerable room for (mis)interpretation of natural language. Using the Behavior Tree approach, inconsistencies and modelling assumptions can be highlighted and communicated to the client, and through keeping (multiple) tags in the tree, traceability from the model back to the original requirements is maintained.

For example, consider a variation on Requirement R3 in which the first phrase is omitted: *The Sensor waits until the Control is ready before trying to detect further errors*. The resulting translation would appear as in Fig. 9, without the first `Sensor < halt >` node. This means that Requirement R3 no longer integrates with Requirement R2. One may be tempted to treat the guard as a state realisation instead, and integrate it with the root node of Requirement R6, but this leads to an inconsistent tree where it is unclear that the *Sensor* process needs to be restarted. The error of omission may be resolved by adding the missing node to Requirement R3 and flagging it as a missing requirement using the colour coding, and later confirming the decision with the client using the tags. Errors of ambiguity may be discovered if there are multiple integration points; redundancy may be discovered if there are identical subtrees; and inconsistency may be uncovered if integration leads to contradictory behaviour.

5. Translating Behavior Trees to CSP_σ

In this section we describe how Behavior Trees may be translated into CSP_σ processes. The translation process is defined so that the structure of the tree is preserved, and is summarised in Fig. 12. The translation occurs in three phases. Phase 1 identifies and marks subtrees that are the target of process kill and reversion. The second phase is the bulk of the translation, where the marked trees are recursively transformed into CSP_σ processes. The final phase collects state and event information. We describe these phases in more detail below. Throughout the translation process we assume that given a Behavior Tree node `N`, a canonical representation of `N` may be generated and used as events and messages. For instance, from

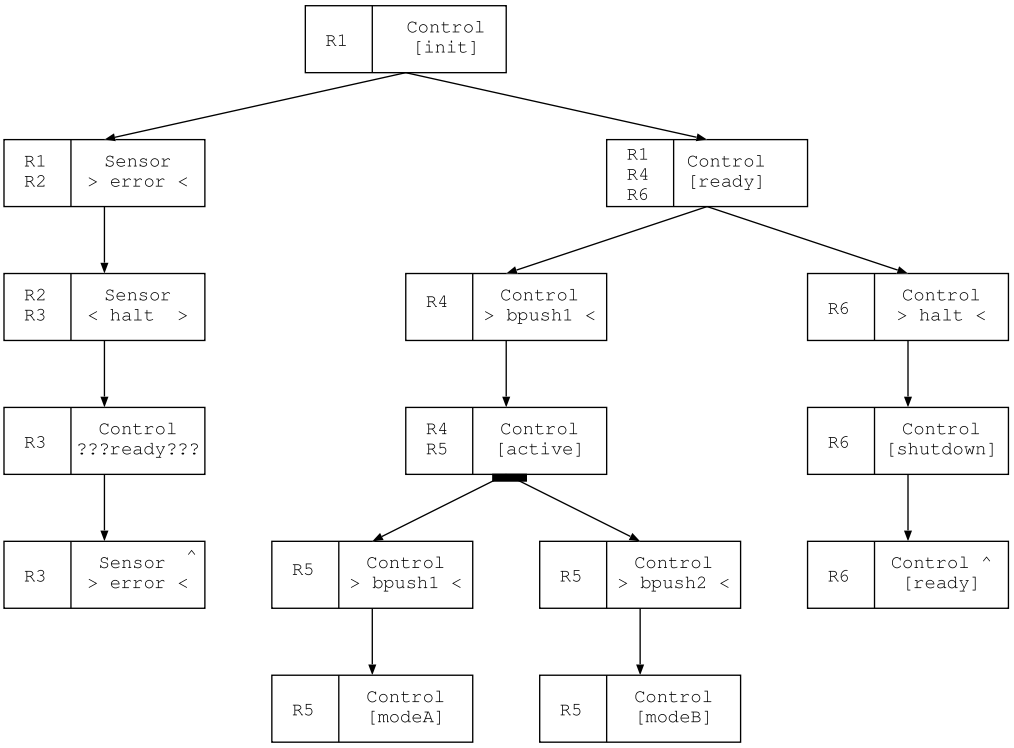


Fig. 11. Fully integrated requirements.

the node `Control [ready]` we may generate a string such as *control.ready*. We write *rev*(`N`) for the reversion (restart) event name generated from node `N`, and *kill*(`N`) for the kill message generated from node `N`.

5.1. Phase 1

Phase 1 is an initial traversal of the Behavior Tree to handle the node modifiers that refer to subtrees found elsewhere in the structure. The first task in Phase 1 is to replace goto nodes $\mathtt{N}^{=>}$ with the target tree, which we write as *tree*(`N`), i.e., the subtree with root node `N`. For this replacement to be well-formed, the scope of the node and its target must be the same. That is, they must reference the same set of variables, events, and messages. This is implied by the well-formedness constraints on Behavior Trees, and that source and target nodes of a goto must appear in sibling branches in an alternative flow.

The remaining tasks of Phase 1 are to mark trees that are the target of a process kill or reversion. Given a tree `T` we let `kill:T` be the tree marked as the target of a process kill, `rev:T` be the tree marked as a target of a reversion node, and `kill:(rev:T)` be the tree marked as the target of both a process kill and a reversion. Trees may of course be unmarked. The marking is a temporary syntactic construct used only as an intermediate step in the translation. The marking may be achieved through a simple traversal of the tree, since any Behavior Tree has a finite number of subtrees.

Given the existence of a node $\mathtt{N}^{--}$ in the tree, the target tree of that node, say `T`, is replaced by `kill:T`. Tree `T` must be the subtree which has root node `N`. A similar translation occurs for trees that are the target of a reversion, or of both a process kill and a reversion.

5.2. Phase 2

Having identified the targets of process kills and reversion, and having eliminated goto nodes, the translation to CSP_σ may begin. We use the $\rightsquigarrow$ relation to transform a Behavior Tree to a CSP_σ process, or a node to a CSP_σ action.

The left-hand column of Fig. 12 for Phase 2 is a left-to-right textual representation of the constructs which were depicted graphically in Figs. 7 and 8, as well as the marked trees described above. The translated versions are given in the right-hand column.

5.2.1. Targets of process kill

A tree with root node `N` that is marked as the target of a process kill node is translated to an interrupt process, where the interrupt is triggered by the canonical message *kill*(`N`). As described below, this is the message that is sent by the corresponding process kill node, $\mathtt{N}^{--}$. After the kill message is received, the process terminates.

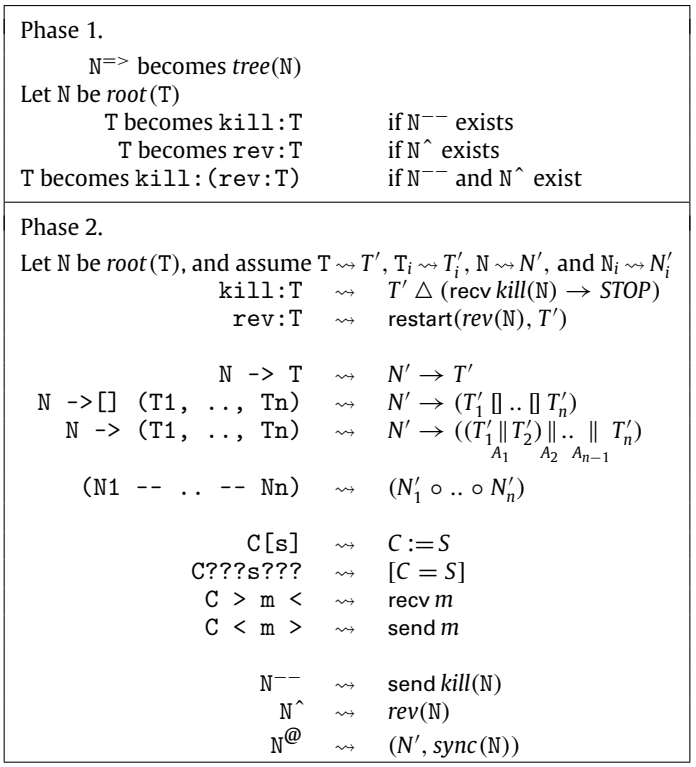


Fig. 12. Translation summary.

5.2.2. Targets of reversion

Before giving the translation for the target of a reversion we first briefly discuss the difference between reversion and recursion: in a reversion the behaviour of relevant sibling threads must terminate. In this sense, a reversion behaves as a restart. When a thread with a reversion does not contain any parallelism, then the behaviour is identical to that of recursion. Consider the following example:

$$\mu\, r \bullet (\mu\, s \bullet .. \rightarrow s) \parallel (.. \rightarrow r)$$

There are two threads operating in parallel. The inner thread scoped by the recursion name s may be thought of as a sensor, while the other thread may be some controller thread. Under certain conditions, the controller thread will need to restart the entire system. As the program above is written, every time the controller restarts the system through unfolding r, a new copy of the sensor thread will be generated. However, this does not model the typical restart of a system, where each subthread will be restarted individually. This is the meaning of reversion, and is part of the Behavior Tree notation because within requirements documents it is more common to find "restart" behaviour than the behaviour associated with pure recursion.

The target of a reversion node becomes a *restart* process, where the restart is triggered by the canonical event name $rev(\mathtt{N})$. This is an event that occurs internally to T', and is generated by a corresponding source reversion node, $\mathtt{N}^\wedge$. When the event $rev(\mathtt{N})$ is generated by the execution of T', the subprocesses associated with the tree T' are terminated and a new copy of T' begins (recall abbreviation (2)).

5.2.3. Targets of process kill and reversion

In the case where a node is the target of both a process kill and a reversion, the resulting process becomes a restart inside an interrupt. The process will execute, and restart, as usual, until the kill message is received, and all behaviour is terminated.

5.2.4. Translation of constructors

A sequential flow in Behavior Trees, represented textually as `N -> T`, is straightforwardly translated to prefixing in CSP_σ. Alternative flow, represented textually as `N ->[] (T1, .., Tn)`, is straightforwardly translated to external choice.

Concurrent branching, represented textually as `N -> (T1, .., Tn)`, is translated to an alphabetised parallel composition. Given a composition $T_1 \underset{A}{\|} T_2$, the interface A is the intersection of the common synchronisation events in T_1 and T_2.[7]

[7] This is a similar process to determining the *alphabet* of an individual process following the ideas of Hoare [10].

An atomic composition of nodes, represented textually as `(N1 -- .. -- Nn)`, are *relationally composed* into a single command/event pair. The individual nodes are translated independently, and composed in order. Since there can be at most one event or message in a CSP_σ action, there can be at most one event-based node in the chain. This means there may be at most one node of type input/output event, process kill, reversion, or synchronisation in the chain. The remaining nodes must be guards, state updates, with potentially a goto node as the leaf (assuming the chain does not also contain a reversion).

The relational composition of two specification commands which have the same frame is given below, and is described in detail in [20].

$$x\colon [R_1] \circ x\colon [R_2] = x\colon \left[\exists x'' \bullet R_1 \left[\frac{x''}{x'}\right] \wedge R_2 \left[\frac{x''}{x}\right]\right]$$

The expression $R_1[\frac{x''}{x'}]$ is R_1 with a syntactic replacement of variables x' with x''. (Note that this is a different type of substitution to that involving states.) For the purposes of defining relational composition when the frames do not match, their frames may be widened according to the rule below.

$$x\colon [R] = x, y\colon [R \wedge y' = y] \quad \text{for } x \cap y = \varnothing \tag{18}$$

We lift relational composition to command/event pairs, as defined below.

$$(c_1, \tau) \circ (c_2, \tau) = (c_1 \circ c_2, \tau)$$
$$(c_1, a) \circ (c_2, \tau) = (c_1 \circ c_2, a) = (c_1, \tau) \circ (c_2, a)$$

The composition is undefined if more than one of the pairs has a non-internal event. An example of translating atomic composition is given in Section 5.4.

5.2.5. Translation of state- and event-based nodes

A state update `C[s]` is straightforwardly translated to an update of C to the value s, $C := s$. A guard `C ???s???` is translated simply to $[C = s]$. The full syntax for Behavior Trees contains constructs for tests other than equality; these may be translated straightforwardly to guards as well.

Input nodes, `C > m <`, are translated to receiving messages, and outputs, `C < m >`, to sending messages.

5.2.6. Kill nodes

A kill node is translated to the sending of message *kill*(`N`), which is the interrupt message in the target process. We use a message, rather than an event, so that the killing thread is not blocked in the case where the target thread is not active.

5.2.7. Reversion nodes

A reversion node, represented textually as `N^`, is translated to the canonical event name *rev*(`N`).

By default we do not place the event *rev*(`N`) into the interfaces of parallel composition. This means that any single reversion node will trigger a restart. However, this may result in undesirable race conditions in some cases. An alternative is for all related reversion nodes (or a selection of them) to synchronise before the restart can take place. In Behavior Trees, this behaviour can be expressed by coupling the reversion symbol with the synchronisation symbol. To translate this behaviour into CSP_σ, the *rev*(`N`) event name must be added to the interfaces of the relevant parallel composition operators.

5.2.8. Synchronisations

A synchronisation, represented textually as `N@`, is mapped to a CSP_σ event name *sync*(`N`), which, as above, we assume may be constructed canonically from N. In addition, the node N itself must be translated, and this is paired with the event *sync*(`N`). This implies that the node N must be a guard or state update, as it is not possible to combine more than one event or message. This is typical of process algebras, where it is not possible to atomically combine the actions of one event with another.

For instance, the node `Control[ready]@` is translated to

(*Control* := *ready*, *control.ready*)

where the event name *control.ready* has been constructed from the node itself. Since all synchronisation nodes by definition encode the same test or update of a component, only one of the synchronisation nodes requires the translated node to be paired with the event; the remaining nodes are translated to the singular event *sync*(`N`).

5.3. Phase 3

In the final phase, all component names are collected and added as local variables, and event and message names used to communicate between local threads are hidden. This limits the effect of the Behavior Tree to interactions with the environment. The scope of a local variable (component) x may be restricted to the smallest subtree that contains all references to x, and similarly an event e, used for internal communication, may be hidden at the level of the smallest subtree that contains all uses of e. The initial value of components are nondeterministically chosen.

$$
\begin{array}{rcl}
Sys & \widehat{=} & (\textbf{state}\ \{Control \mapsto \_\} \bullet Main) \backslash \{\mathsf{send}\ halt, \mathsf{recv}\ halt\} \\
Main & \widehat{=} & (Control := init) \rightarrow \\
 & & \quad (\mathsf{restart}(rev(s), Sensor)) \\
 & & \quad \parallel \\
 & & \quad (\mathsf{restart}(rev(c), ControlReady)) \\
Sensor & \widehat{=} & \mathsf{recv}\ error \rightarrow \mathsf{send}\ halt \rightarrow [Control = ready] \rightarrow rev(s) \\
ControlReady & \widehat{=} & (Control := ready) \rightarrow \\
 & & \quad \mathsf{recv}\ halt \rightarrow (Control := shutdown) \rightarrow rev(c) \\
 & & \quad \parallel \\
 & & \quad \mathsf{recv}\ bpush1 \rightarrow (Control := active) \rightarrow \\
 & & \qquad (\mathsf{recv}\ bpush1 \rightarrow (Control := modeA) \\
 & & \qquad [] \ \mathsf{recv}\ bpush2 \rightarrow (Control := modeB))
\end{array}
$$

Fig. 13. Translated version of Fig. 11.

5.4. Example translation

The translation of the Behavior Tree in Fig. 11 is given in Fig. 13. To ease the presentation we break the definition into several (named) subprocesses. Because there are no synchronisations, the interfaces of the parallel composition operators are empty and hence omitted. All revert events e are written $rev(e)$, and for brevity we allow leaf nodes N to abbreviate the process $N \rightarrow SKIP$. At the top level we define the process *Sys*, which gives the context of the system. It includes the variable *Control* to represent the state of the *Control* component, which has some unknown initial value (represented by an underscore). To keep the tree relatively concise, we hide the *halt* message, which is used for communication between the *Sensor* and *ControlReady* processes, at the topmost level, although it could be hidden with a smaller scope on the parallel composition within the definition of *Main*. The *error* and *bpush* messages are not hidden, since they are received from processes outside of the scope of *Sys*.

This example shows the translation of the control structures parallel, alternative, and sequential flow, and the node types state update, guard, input/output event and reversion. We now provide examples of the remaining constructs.

5.4.1. Process kill

Consider extending the controller system to add Requirement R7, that states that "The *Sensor* terminates if the *level* exceeds 50" (we do not elaborate further on the meaning of *level*). The corresponding Behavior Tree for R7 is given in Fig. 14(a)—the root node of the *Sensor* process is the target of a process kill node after the value of *level* exceeds 50. The translation of this behaviour into CSP_σ introduces a new message name, $kill(s)$, that acts as an interrupt to the *Sensor* process.

$$(Sensor \ \triangle\ \mathsf{recv}\ kill(s)) \quad \parallel \quad (.. \rightarrow [level > 50] \rightarrow \mathsf{send}\ kill(s) \rightarrow ..)$$

The introduction of the process kill node means that the node `Sensor >error<` is now the target of both a kill and a reversion.

5.4.2. Synchronisation

Consider using synchronisation nodes to communicate between the *Control* and the *Sensor*, as in Fig. 14(b). Instead of a guard to check whether the *Control* is ready, the *Sensor* synchronises on the change to the ready state. Note the use of the synchronisation flag "@".

The synchronisation nodes are translated as described earlier: we arbitrarily choose one of the nodes (in the *Control* process) to contain the action of updating the *Control*, while both nodes synchronise on the event *control.ready*, which is placed into the interface of the parallel operator.

$$Control := init \rightarrow \quad (.. \rightarrow control.ready \rightarrow ..) \quad \parallel_{\{control.ready\}} \quad (Control := ready, control.ready) \rightarrow ..$$

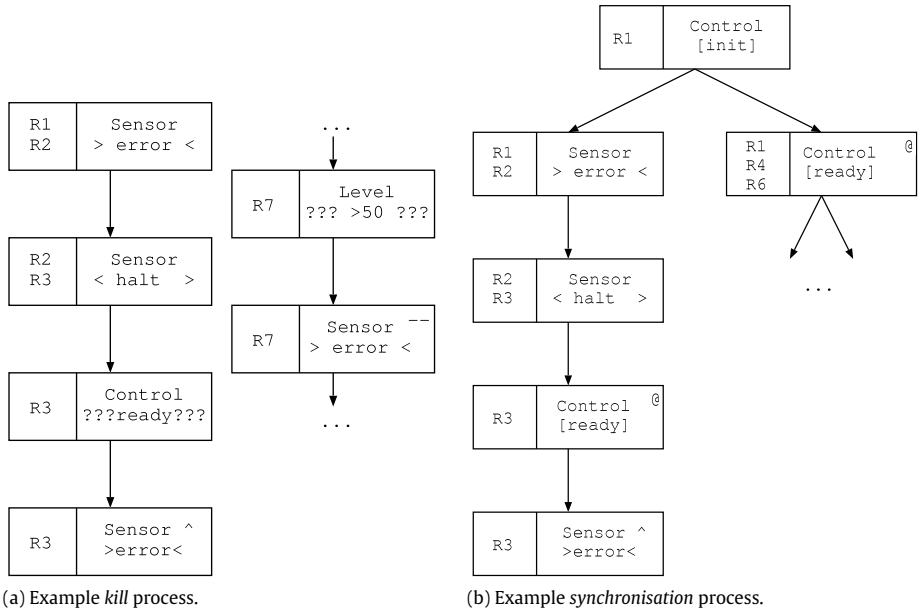
This is a stronger, and perhaps more correct, version of the specification, since the *Sensor* will restart as soon as the *Control* becomes ready; using a guard, the *Sensor* will not become ready until the next time the *Sensor* process takes a step. This is a common issue (a "race condition") with shared-variable communication.

5.4.3. Atomic composition

Consider the modification of the controller system in Fig. 14(c), which specifies that when the *Control* component is in the ready state and button 1 is pushed, the *Control* immediately becomes active. Unarrowed lines are used to connect the first three nodes. The three nodes are combined into a single atomic action, which is enabled based on the state of the component *Control*, a message being received, and includes an update of the state.

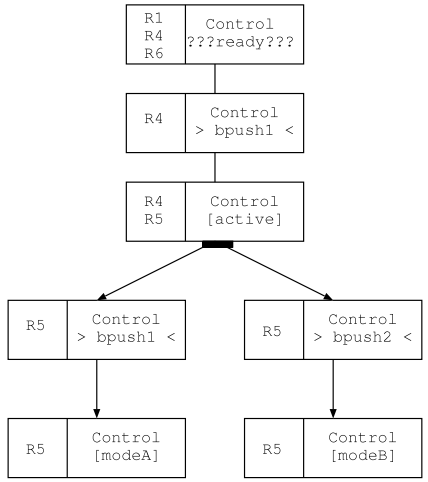
Using relational composition as described earlier, we generate the following single action, in which we abbreviate *Control* by C, to represent the three nodes.

$$(C : [C = ready \wedge C' = active], \mathsf{recv}\ bpush1) \rightarrow ..$$



(a) Example *kill* process.

(b) Example *synchronisation* process.



(c) Example *atomic* process.

Fig. 14. Translation examples.

5.5. Summary

We have presented a general process for translating Behavior Trees into an extended version of CSP. The structure of the Behavior Tree itself is preserved, with only minor additions required for handling reversion and process termination. The most complex translation was that for reversion, due to its subtle difference to recursion; however, the restart operator is roughly of the same complexity as the *interrupt* and *exception* operators of CSP [11,34].

Since there is a structure-preserving mapping from Behavior Trees to an extended version of CSP, there is scope for using program verification methods such as animation, model checking, and refinement, which can build on existing support for CSP. Furthermore, the structure-preserving nature of the translation also admits representing the animation of the dynamic behaviour of the models back to the original graphical Behavior Tree. This "backwards-translation" process would show the dynamic flow of control through the requirements document using the tags on the nodes.

6. Conclusions

In this paper we have presented a semantics for Behavior Trees. This is a graphical notation used for building a model of a system from requirements found in informally written documents, and as such it contains a collection of language constructs: state changes and tests, message passing, and synchronisation. Typical specification languages are either state

based, such as *Z* [41] and VDM [42], or event based, such as CSP [10,11] and CCS [43]. Our approach to defining the semantics for Behavior Trees was to extend CSP, which natively handles process synchronisation, to include hierarchical state and a publish/subscribe notion of message passing. We then gave a translation from the Behavior Tree notation into a process in the extended CSP language. The formal semantics provides a definition of the Behavior Tree notation which can be used as the basis for developing tool support for simulation, model checking, and theorem proving.

Plotkin's seminal paper on operational semantics [35] defines transition rules for imperative languages with state. There are also many other examples of such semantics in the literature, notably the semantics of Hoare and He Jifeng [44], and the semantics for the programming language Occam [36]. Our approach is different in that the state is treated as part of the process, and guards and updates are treated as labels to the transition relation. This allows state accesses to be (perhaps partially) instantiated within a context which defines the values of the local state. The traditional operational semantics approach defines the transition relation on program/state pairs, and the state is updated in the rule for each construct (e.g., update). This approach does not so easily support the hierarchical construction of the state as in our approach, with local variables in the traditional style being captured as global variables with syntactic restrictions. In the approach adopted here, by treating state access as transition labels, the state-based reasoning is 'quarantined' to a single, general rule (Rule 12), allowing the construct rules, e.g., Rule 1, to be defined concisely, and without explicit reference to a particular state.

Acknowledgements

The authors are indebted to the late R. Geoff Dromey for inspiring the Behavior Tree formalisation research. We thank Kirsten Winter, and other members of the *Dependable Complex Computer-based Systems* group, for their help with the Behavior Tree notation. We also thank the three anonymous referees of [20] for their comments on the CSP_σ language, and three anonymous referees of this paper for their suggestions for improvement. This work is supported, in part, by the Australian Research Council (ARC) Linkage Grant LP0989363, *Reducing the risks associated with developing large-scale, critical software-integrated systems*.

References

[1] R.G. Dromey, Formalizing the transition from requirements to design, in: J. He, Z. Liu (Eds.), Mathematical Frameworks for Component Software: Models for Analysis and Synthesis, Component-Based Development, World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2006, pp. 156–187.
[2] R.G. Dromey, From requirements to design: formalizing the key steps, keynote address, in: 1st International Conference on Software Engineering and Formal Methods (SEFM), IEEE Computer Society, 2003, pp. 2–11.
[3] C. Smith, K. Winter, I.J. Hayes, R.G. Dromey, P.A. Lindsay, D.A. Carrington, An environment for building a system out of its requirements, in: 19th IEEE International Conference on Automated Software Engineering, ASE, IEEE Computer Society, 2004, pp. 398–399.
[4] R.G. Dromey, D. Powell, Early requirements defects detection, TickIT J. 4Q05 (2005) 3–13.
[5] D. Powell, Requirements evaluation using Behavior Trees—findings from industry, Industry Track Papers, Australian Software Engineering Conference, ASWEC, 2007, http://aswec07.cs.latrobe.edu.au/itp-aswec2007.htm.
[6] Raytheon Australia, http://www.raytheon.com.au/.
[7] P. Papacostantinou, T. Tran, P. Lee, V. Phillips, Implementing a Behavior Tree analysis tool using Eclipse development frameworks, in: A. Aitken, S. Rosbotham (Eds.), 19th Australian Software Engineering Conference, Experience Report Proceedings, Curtin University of Technology, 2008, pp. 61–66.
[8] L. Grunske, K. Winter, N. Yatapanage, Defining the abstract syntax of visual languages with advanced graph grammars—a case study based on Behavior Trees, J. Vis. Lang. Comput. 19 (3) (2008) 343–379.
[9] L. Wen, R. Colvin, K. Lin, J. Seagrott, N. Yatapanage, R.G. Dromey, Integrare, a collaborative environment for behavior-oriented design, in: Y. Luo (Ed.), Cooperative Design, Visualization, and Engineering, CDVE, in: Lecture Notes in Computer Science, vol. 4674, Springer, 2007, pp. 122–131.
[10] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.
[11] A.W. Roscoe, The Theory and Practice of Concurrency, Prentice Hall, 1998.
[12] P.T. Eugster, P.A. Felber, R. Guerraoui, A.-M. Kermarrec, The many faces of publish/subscribe, ACM Comput. Surv. 35 (2) (2003) 114–131.
[13] Failures-Divergence Refinement: FDR2 User Manual, Formal Systems (Europe) Ltd., 1999.
[14] M. Leuschel, M. Fontaine, Probing the depths of CSP-M: a new FDR-compliant validation tool, in: S. Liu, T.S.E. Maibaum, K. Araki (Eds.), Formal Methods and Software Engineering, 10th International Conference on Formal Engineering Methods, ICFEM, in: Lecture Notes in Computer Science, vol. 5256, Springer, 2008, pp. 278–297.
[15] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, Addison-Wesley, 1998.
[16] R. Colvin, I.J. Hayes, A semantics for Behavior Trees, ACCS Technical Report ACCS-TR-07-01, ARC Centre for Complex Systems, April 2007.
[17] K. Winter, Formalising Behavior Trees with CSP, in: Integrated Formal Methods, in: LNCS, vol. 2999, Springer Verlag, 2004, pp. 148–167.
[18] R. Colvin, L. Grunske, K. Winter, Timed Behavior Trees for failure mode and effects analysis of time-critical systems, J. Syst. Softw. 81 (12) (2008) 2163–2182.
[19] R. Colvin, L. Grunske, K. Winter, Probabilistic timed Behavior Trees, in: J. Davies, J. Gibbons (Eds.), Proceedings of the International Conference on Integrated Formal Methods, IFM, in: Lecture Notes in Computer Science, vol. 4591, Springer-Verlag, 2007, pp. 156–175.
[20] R. Colvin, I.J. Hayes, CSP with hierarchical state, in: M. Leuschel, H. Wehrheim (Eds.), Integrated Formal Methods, IFM 2009, in: Lecture Notes in Comp. Sci., vol. 5423, Springer, 2009, pp. 118–135.
[21] J.C.P. Woodcock, A.L.C. Cavalcanti, The semantics of circus, in: D. Bert, J.P. Bowen, M.C. Henson, K. Robinson (Eds.), ZB 2002: Formal Specification and Development in Z and B, in: Lecture Notes in Computer Science, vol. 2272, Springer-Verlag, 2002, pp. 184–203.
[22] G. Smith, A semantic integration of Object-Z and CSP for the specification of concurrent systems, in: J.S. Fitzgerald, C.B. Jones, P. Lucas (Eds.), 4th International Symposium of Formal Methods Europe, FME 97, in: Lecture Notes in Computer Science, vol. 1313, Springer, 1997, pp. 62–81.
[23] C. Fischer, H. Wehrheim, Model-Checking CSP-OZ Specifications with FDR, in: K. Araki, A. Galloway, K. Taguchi (Eds.), Integrated Formal Methods, 1st International Conference, Proceedings, Springer, 1999, pp. 315–334.
[24] M. Butler, A CSP approach to action systems, Ph.D. Thesis, Computing Laboratory, Oxford University, 1992.
[25] M.J. Butler, M. Leuschel, Combining CSP and B for specification and property verification, in: J. Fitzgerald, I.J. Hayes, A. Tarlecki (Eds.), FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Proceedings, in: Lecture Notes in Computer Science, vol. 3582, Springer, 2005, pp. 221–236.
[26] S. Schneider, H. Treharne, CSP theorems for communicating B machines, Formal Asp. Comput. 17 (4) (2005) 390–422.

[27] J.C.M. Baeten, J.A. Bergstra, Global renaming operators in concrete process algebra, Inf. Comput. 78 (3) (1988) 205–245.
[28] J.C.M. Baeten, J.A. Bergstra, Process algebra with propositional signals, Theor. Comput. Sci. 177 (2) (1997) 381–405.
[29] K.G. Larsen, L. Xinxin, Compositionality through an operational semantics of contexts, J. Log. Comput. 1 (6) (1991) 761–795.
[30] J. Sun, Y. Liu, J.S. Dong, C. Chen, Integrating specification and programs for system modeling and verification, in: W.-N. Chin, S. Qin (Eds.), Third IEEE International Symposium on Theoretical Aspects of Software Engineering, TASE, IEEE Computer Society, 2009, pp. 127–135.
[31] J. Sun, Y. Liu, J.S. Dong, J. Pang, PAT: Towards flexible verification under fairness, in: A. Bouajjani, O. Maler (Eds.), 21st International Conference on Computer Aided Verification, CAV, in: Lecture Notes in Computer Science, vol. 5643, Springer, 2009, pp. 709–714.
[32] S. Schneider, Concurrent and Real-time Systems: The CSP Approach, Wiley, 2000.
[33] C. Morgan, Programming from Specifications, 2nd ed., Prentice Hall, 1994.
[34] A.W. Roscoe, The three platonic models of divergence-strict CSP, in: J.S. Fitzgerald, A.E. Haxthausen, H. Yenigün (Eds.), International Colloquium on Theoretical Aspects of Computing, ICTAC, in: Lecture Notes in Computer Science, vol. 5160, Springer, 2008, pp. 23–49.
[35] G.D. Plotkin, A structural approach to operational semantics, J. Log. Algebr. Program. 60–61 (2004) 17–139.
[36] Y. Gurevich, L.S. Moss, Algebraic operational semantics and Occam, in: E. Börger, H.K. Büning, M.M. Richter (Eds.), Proceedings of 3rd Workshop on Computer Science Logic, CSL 89, in: Lecture Notes in Computer Science, vol. 440, Springer, 1990, pp. 176–192.
[37] Y. Chevalier, L. Compagna, J. Cuéllar, P.H. Drielsma, J. Mantovani, S. Mödersheim, L. Vigneron, A high-level protocol specification language for industrial security-sensitive protocols, in: Specification and Automated Processing of Security Requirements, Austrian Computer Society, 2004, pp. 193–205.
[38] A. Armando, D.A. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuéllar, P.H. Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, L. Vigneron, The AVISPA tool for the automated validation of internet security protocols and applications, in: K. Etessami, S.K. Rajamani (Eds.), Computer Aided Verification, 17th International Conference, CAV 2005, Proceedings, in: Lecture Notes in Computer Science, vol. 3576, Springer, 2005, pp. 281–285.
[39] L. Cardelli, A.D. Gordon, Mobile ambients, in: M. Nivat (Ed.), Foundations of Software Science and Computation Structure, First International Conference, FoSSaCS'98, in: Lecture Notes in Computer Science, vol. 1378, Springer, 1998, pp. 140–155.
[40] K. Winter, I.J. Hayes, R. Colvin, Integrating requirements: the Behavior Tree philosophy, in: J.L. Fiadeiro, S. Gnesi (Eds.), Proceedings of International Conference on Software Engineering and Formal Methods, SEFM 2010, IEEE Computer Society Press, 2010, pp. 41–50.
[41] J.M. Spivey, The Z Notation: A Reference Manual, 2nd ed., Prentice Hall, 1992.
[42] C.B. Jones, Systematic Software Development using VDM, Prentice Hall, 1990.
[43] R. Milner, A Calculus of Communicating Systems, Springer-Verlag New York, Inc., 1982.
[44] C.A.R. Hoare, J. He, Unifying Theories of Programming, Prentice Hall, 1998.