Softw Syst Model (2017) 16:1145–1178
DOI 10.1007/s10270-015-0511-z

REGULAR PAPER

On the formal interpretation and behavioural consistency checking of SysML blocks

Jaco Jacobs[1] · Andrew Simpson[1]

Received: 8 October 2014 / Revised: 28 October 2015 / Accepted: 3 November 2015 / Published online: 21 December 2015

Abstract The Systems Modeling Language (SysML) is a semi-formal, graphical modelling language used in the specification and design of systems. We describe how Communicating Sequential Processes (CSP) and its associated refinement checker, FDR3, may be used to underpin an approach that facilitates the refinement checking of the behavioural consistency of SysML diagrams. We achieve this by utilising CSP as a semantic domain for reasoning about SysML behavioural aspects: activities and state machines are given a formal, process-algebraic semantics. These behaviours execute within the context of the structural diagrams to which they relate, and this is reflected in the CSP descriptions that depict their characteristic patterns of interaction. We describe how CSP and FDR3 can be used in conjunction with SysML in a formal, top-down approach to systems engineering. Moreover, the compositionality afforded by CSP alleviates the state space explosion problem frequently encountered with complex formal models and complements the top-down approach of SysML. Typically, a system is composed from constituent systems using the concept of blocks. SysML permits two alternative interpretations with regard to the behaviour of the resulting composition. We argue that the use of a process-algebraic formalism enables us to explore the relationships between these interpretations in a more rigorous fashion than would otherwise be the case.

Keywords SysML · CSP · State machines · Activities · Blocks

Communicated by Dr. Bernhard Schaetz.

✉ Andrew Simpson
Andrew.Simpson@cs.ox.ac.uk

Jaco Jacobs
Jaco.Jaobs@cs.ox.ac.uk

[1] Department of Computer Science, University of Oxford, Wolfson Building, Parks Road, Oxford OX1 3QD, UK

1 Introduction

Modern systems are typically characterised as compositions of interconnecting components or systems, functioning as a whole in order to achieve a shared goal. This makes it increasingly difficult to orchestrate combined behaviour and to ensure that the intended behavioural characteristics of the complete system are adhered to, while maintaining operational independence of the components or systems that make up the whole. The need to establish solutions to technologically challenging problems, often in a short time frame, further complicates the matter. Moreover, the environment external to a system is constantly evolving and ever more demanding, resulting in external interactions of increased complexity. These interactions are more taxing, either because the systems are being used in an unfamiliar or previously unforeseen context, or because the systems themselves are more complex. Thus, it is clear that a systematic approach to deal with this inherent complexity and entanglement would be beneficial in a number of different ways.

The *Systems Modeling Language* (SysML) [17], which was proposed by the Object Management Group (OMG)[1], is a graphical modelling notation that can be used to describe complex, heterogeneous systems comprised of various components. Modelling a system with SysML relies on the concept of *blocks*—each of which has an associated set

[1] http://www.omg.org.

of states—communicating via events, possibly resulting in a change of state for one or more of the communicating blocks. The architecture of these systems allows a top-down design, starting from an abstract level with high-level concepts, down to levels with increasingly more detail. These successive transformations give rise to the pleasing opportunity of allowing the replacement of an abstract block with a composition of parts; however, a big drawback of this decomposition approach is that it is at best semi-formal and, as such, cannot guarantee consistency between a block and its parts.

Communicating Sequential Processes (CSP) [9, 19] is a process algebra that can be used to describe complex patterns of interaction between processes, with each process having its own characteristic behaviour. In this paper, we show how CSP can be used to precisely define the differing notions of composition outlined above. Moreover, the associated refinement checker, FDR3 [8], gives rise to a practical approach that enables us to reason about these interactions. We make use of a case study to illustrate the approach.

SysML and CSP have clear differences. We would argue, however, that it would be beneficial to develop a framework that integrates the two, with a view to offering the benefits of both. Specifically, by translating SysML into CSP, there is the potential to give a precise definition of the intended behaviour of a given SysML model, by making use of the underlying formal semantics of CSP. Consequently, this would allow us to undertake refinement checking of the SysML model. It should be noted that the intention is not to replace existing SysML modelling tools; rather, the intention is to develop a formal framework that can be used in conjunction with such tools in order to complement the modelling activity being undertaken.

Activities and state machines are the core behavioural constructs used to ascribe behaviour to SysML blocks. The aforementioned constructs are frequently used in combination: activities are used to assign behavioural features that ought to execute in a particular state, or on a given transition [17]. In this paper, we provide a behavioural semantics for the conjoined behaviour of state machines and activities. There have been several contributions where the sole focus was either the formalisation of state machines or the formalisation of activities. To the best of our knowledge, this paper is the first contribution where the intention is on the provision of a behavioural semantics that encompasses both these formalisms.

At the structural level, SysML takes a compositional approach with regard to systems specification: a block can be comprised of other blocks, which, in turn, might themselves consist of blocks. However, for the approach to be effective and useful, the behavioural conducts of these blocks need to be specified in a consistent manner. Moreover, the approach needs to enable the modeller to sufficiently abstract away details irrelevant to a particular level of abstraction. Friedenthal et al. [7] suggest two alternative interpretations with regard to the combined behaviour. These can be characterised as follows.

1. The classifier behaviour of the block can serve as an abstraction of the behaviours of its parts. The abstraction serves as a specification that the parts must realise: the parts must interact in such a way that their combined behaviour conforms to the abstraction.
2. Alternatively, the classifier behaviour of the block acts as a controller in order to actively orchestrate the behaviours of its parts. In this case, the behaviour of the block is a combination of its behaviour and that of its parts.

We argue that the use of a process-algebraic formalism enables us to explore the relationships between these interpretations in a more rigorous fashion than would otherwise be the case. CSP was selected due to its compositional nature and good tool support.

When mapping between two seemingly disparate notations, it is crucial that the source notation be adequately constrained in order to ensure that the resulting formulation in the target notation is sensible. To this end, we place certain restrictions on, or make certain assumptions about, the state machines we consider. These are detailed below.

– Every top-level state machine[2] in our formalisation is a simple composite state: the region of this state houses the entire state machine. This state has no associated state-based behaviours.
– Orthogonal composite states are not allowed. As such, we do not consider fork and join states.
– We do not support do behaviours.
– Transitions between regions are limited to source and target states that are simple or simple composite states themselves. Thus transitions that cross-regional boundaries are not permitted to start or terminate on pseudostates.
– Submachine states are not supported.
– We do not consider history pseudostates in our formalisation.
– Every region contains a valid state machine. In particular, this state machine is allowed only one compulsory initial state. Multiple final or terminate states are permissible, as appropriate.
– Each initial state has a single outgoing transition: by insisting on a single outgoing transition, we ensure that the first active starting state is unambiguously defined.
– The outgoing transition of an initial state may not contain explicit triggers or guards.

[2] The state machine at the root of the state hierarchy.

Similarly, we constrain the activities we consider by imposing the following restrictions.

- Every activity has a unique starting point designated either by a single initial node, or, if there is no initial node, a single parameter node from which control starts.
- Flow final nodes are not permitted.
- We do not allow for activity partitions in our formalisation.
- An action can have either outgoing object flows or outgoing control flows, but not both.
- The behaviour of a call behaviour action in our formalisation is assumed to be an activity.
- We exclude activities that have activity parameter output nodes from consideration.
- We only consider simple fork and join nodes: if a fork node splits into k separate flows, then those k flows will eventually be joined via a join node.
- Incoming flows are not allowed for initial nodes.
- Initial nodes are allowed a single outgoing control flow.

The structure of the remainder of this paper is as follows. In Sect. 2, we give the background to our work. Then, in Sect. 3, we present our CSP model of state machines. Section 4 formalises activities, while Sect. 5 considers a formal model of SysML blocks. We then illuminate and validate our approach, via a case study, in Sect. 6. Finally, in Sect. 7, we summarise the contribution of this paper and place it in the context of work undertaken by other authors.

2 Background

In this section, we give consideration to our languages of interest—CSP and SysML. We start by presenting a brief introduction to CSP.

2.1 Communicating Sequential Processes

2.1.1 Syntax

Events are at the heart of CSP—they are fundamental to the synchronisation mechanism that is employed—with an event being an indivisible communication or interaction. We denote by Σ the set of all possible events for a particular specification. We can also give consideration to the *alphabet* of a process—the events that it can perform. We write αP to denote the alphabet of a process P.

A communication takes place when two or more processes agree on an event. The communication can be a primitive event, or it can take a more structured, message-passing form, utilising channels. The message-passing mechanism is based on the principle of a rendezvous between a sending and a receiving process: if the communication takes place on a channel, c, and a sending process wants to output a value, e, then the receiving process has to allow for this (by inputting on c). Once this has happened, the event is abstracted as $c.e$. A process indicates that it intends to output a value on a channel using the syntax $c!e$; the willingness to receive an input on a channel is expressed as $c?x$.

CSP is compositional in the sense that it provides operators that allow us to define a process in terms of other, constituent processes. In the following, we consider those operators that are of relevance to this contribution.

The CSP syntax utilised in this paper is defined as follows:

$$
\begin{aligned}
P = {} & P \mid \\
& \mathit{Stop} \mid \\
& \mathit{Skip} \mid \\
& e \rightarrow P \mid \\
& P \mathbin{\Box} P \mid \\
& \Box\, e : X \bullet e \rightarrow P \mid \\
& P \sqcap P \mid \\
& \sqcap\, e : X \bullet e \rightarrow P \mid \\
& P \setminus X \mid \\
& P \mathbin{;} P \mid \\
& P\,[\,X \parallel Y\,]\,P \mid \\
& P\,[|\,X\,|]\,P \mid \\
& \parallel i \bullet [X_i] P_i \mid \\
& P \mathbin{|||} P \mid \\
& ||| \, i \bullet P_i \mid \\
& \mathit{if}\ b\ \mathrm{then}\ P\ \mathrm{else}\ P \mid \\
& \mathit{let}\ P_1, .., P_n\ \mathit{within}\ P
\end{aligned}
$$

In the above, P, P_1, P_n and P_i denote processes, e denotes an event, X, Y and X_i denote sets of events, and b denotes a Boolean condition.

Stop denotes the deadlocked CSP process: it refuses to participate in all events. *Skip* is the process that communicates the special internal event, ✓, before behaving like *Stop*; it is used to model successful termination.

The process $e \rightarrow P$, modelled using the *prefixing* operator, $\rightarrow$, performs the event e and subsequently behaves as the process P.

CSP provides two choice operators: the *external* or *deterministic choice* operator, $\Box$, offers the environment the choice between the initial events of its argument processes; conversely, the *internal* or *nondeterministic choice* operator, $\sqcap$, offers no such choice and the observed behaviour may be that of either of the two participating processes. Indexed versions exist for both operators. For example, $\Box\, i : I \bullet P_i$ is an external choice between processes P_i, where i serves as an index for the parameterised process P.

The application of the *hiding* operator, $\setminus$, in the process $P \setminus X$ conceals the events of X from the view of the external environment of P.

The process $P_1 \fatsemi P_2$ represents the *sequential composition* of processes P_1 and P_2. This process behaves as P_1 until it terminates successfully, after which it behaves as P_2.

Several parallel operators exist in CSP.

First, the process $P_1 \, [| \, X \, |] \, P_2$ uses the *generalised parallel* operator to define an interface, consisting of the events of X, on which P_1 and P_2 must synchronise. Events outside X may occur independently in either process. The process $P_1 \, [\, X \parallel Y \,] \, P_2$ denotes *alphabetised parallel*, where synchronisation takes place on events in the set $X \cap Y$. Finally, the *interleaving* operator, $|||$, expresses the unsynchronised concurrent interleaving of the events of its two constituent processes.

Indexed forms exist for all of these parallel operators. For example, $||| \, x : X \bullet P_x$ represents the indexed interleaving of processes of the form P_x, where $x \in X$.

A conditional choice construct is available in the form $\mathit{if}\ b$ then P_1 else P_2, where a process behaves as P_1 if b is true and P_2 otherwise.

The $\mathit{let\ within}$ construct allows us to use local definitions (of the form of $P_1, .., P_n$) in the definition of a complex process: $\mathit{let}\ P_1, .., P_n\ \mathit{within}\ P$.

2.1.2 Semantics

CSP allows us to compare the behaviour of one process against that of another. Several semantic models exist, but the consideration of *traces* and *failures* is sufficient for our purposes in this paper.

The traces of a process, P, written $\mathit{traces} [\![P]\!]$, are the set of all finite sequences of observable events.

For example, $\mathit{traces} [\![a \rightarrow b \rightarrow \mathit{Stop} \sqcap c \rightarrow d \rightarrow \mathit{Stop}]\!]$ is the set

$$\{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle c\rangle, \langle c, d\rangle\}$$

P/t represents the state of the process P after the observation of the trace t, and $\mathit{refusals} [\![P]\!]$ represents the initial set of events refused by P, no matter how long they are offered. To quote Roscoe [18]: "A *refusal set* is a set of events that a process can fail to accept anything from however long it is offered" [18]. It follows that the elements of $\mathit{failures} [\![P]\!]$ are the pairs of the form (t, X) for some $t \in \mathit{traces} [\![P]\!]$, and such that $X \subseteq \mathit{refusals} [\![P/t]\!]$. As an example, the pair $(\langle a\rangle, \{a, c, d\})$ is a failure of the above internal choice, as is $(\langle c\rangle, \{a, b, c\})$.

We define *traces-refinement*, using reverse containment, as

$$\begin{array}{l} P_1 \sqsubseteq_T P_2 \\ \Leftrightarrow \\ \mathit{traces} [\![P_2]\!] \subseteq \mathit{traces} [\![P_1]\!] \end{array}$$

For example, the process $a \rightarrow b \rightarrow \mathit{Stop}$—the traces of which are given by the set $\{\langle\rangle, \langle a\rangle, \langle a, b\rangle\}$—is a traces-refinement of the process $a \rightarrow b \rightarrow \mathit{Stop} \sqcap c \rightarrow d \rightarrow \mathit{Stop}$:

$$\begin{array}{l} a \rightarrow b \rightarrow \mathit{Stop} \sqcap c \rightarrow d \rightarrow \mathit{Stop} \\ \sqsubseteq_T \\ a \rightarrow b \rightarrow \mathit{Stop} \end{array}$$

We define *failures-refinement* similarly:

$$\begin{array}{l} P_1 \sqsubseteq_F P_2 \Leftrightarrow \\ \quad \mathit{traces} [\![P_2]\!] \subseteq \mathit{traces} [\![P_1]\!] \\ \quad \wedge \\ \quad \mathit{failures} [\![P_2]\!] \subseteq \mathit{failures} [\![P_1]\!] \end{array}$$

For example, the following refinement holds as the second process is more deterministic than the first:

$$\begin{array}{l} a \rightarrow b \rightarrow \mathit{Stop} \sqcap c \rightarrow d \rightarrow \mathit{Stop} \\ \sqsubseteq_F \\ a \rightarrow b \rightarrow \mathit{Stop} \,\Box\, c \rightarrow d \rightarrow \mathit{Stop} \end{array}$$

The traces of both processes are given by

$$\{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle c\rangle, \langle c, d\rangle\}$$

The failures of $a \rightarrow b \rightarrow \mathit{Stop} \sqcap c \rightarrow d \rightarrow \mathit{Stop}$ are given by

$$\begin{array}{l} \{X : \mathbb{P}\{a, b, d\} \bullet (\langle\rangle, X)\} \\ \cup \\ \{X : \mathbb{P}\{b, c, d\} \bullet (\langle\rangle, X)\} \\ \cup \\ \{X : \mathbb{P}\{a, c, d\} \bullet (\langle a\rangle, X)\} \\ \cup \\ \{X : \mathbb{P}\{a, b, c\} \bullet (\langle c\rangle, X)\} \\ \cup \\ \{X : \mathbb{P}\{a, b, c, d\} \bullet (\langle a, b\rangle, X)\} \\ \cup \\ \{X : \mathbb{P}\{a, b, c, d\} \bullet (\langle c, d\rangle, X)\} \end{array}$$

In the above, $\mathbb{P}\{a, b, d\}$ denotes the power set of $\{a, b, d\}$, i.e. the set of all subsets of $\{a, b, d\}$.

The failures of $a \rightarrow b \rightarrow \mathit{Stop} \,\Box\, c \rightarrow d \rightarrow \mathit{Stop}$ are given by

$$\begin{array}{l}\{X : \mathbb{P}\{b,d\} \bullet (\langle\rangle, X)\}\\ \cup\\ \{X : \mathbb{P}\{a,c,d\} \bullet (\langle a\rangle, X)\}\\ \cup\\ \{X : \mathbb{P}\{a,b,c\} \bullet (\langle c\rangle, X)\}\\ \cup\\ \{X : \mathbb{P}\{a,b,c,d\} \bullet (\langle a,b\rangle, X)\}\\ \cup\\ \{X : \mathbb{P}\{a,b,c,d\} \bullet (\langle c,d\rangle, X)\}\end{array}$$

The refinement checking tool FDR3 [8]—which uses the machine-readable dialect of CSP, CSP$_M$—uses this theory of refinement to investigate whether a potential design meets its specification. A pleasing feature of FDR3 is that if such a test fails, a counter-example is returned to indicate why this is so. The interested reader should refer to [9] for a comprehensive introduction to the semantics of CSP.

2.2 Systems Modelling Language

SysML is a more compact language than UML. This fact is reflected by the total number of diagrams and constructs present in the specification: SysML has a total of nine diagrams, whereas UML has 14.

SysML is an extension of a subset of UML: some of the UML diagrams are reused (state machine diagram, sequence diagram and package diagram); some are extended (activity diagram); some are modified (block definition diagram and internal block diagram modify the class diagram and composite structure diagram, respectively); and others are newly introduced (requirement diagram and parametric diagram).

Blocks are the core modelling constructs of SysML and provide the context in which behaviours execute. A block is often composed of other blocks, termed *parts*, each of which has its own associated behaviour. As per Sect. 1, there are two alternative interpretations with regard to the combined behaviour [7]: the *classifier behaviour* of the block can serve as an abstraction of the behaviours of its parts; alternatively, the classifier behaviour of the block acts as a controller in order to actively orchestrate the behaviours of its parts.

Classifier behaviours are the main behaviours of blocks. They execute from the instant the instance is created until the point of destruction. The modelling construct most frequently used to represent the classifier behaviour is a state machine. In most systems engineering methodologies, activities are typically used as a complementary modelling notation to state machines: it is the behavioural formalism normally associated with the effect component of a transition; alternatively, it is used to model behaviours related to a particular state.

A *signal* is a classifier that types the asynchronous messages that are communicated between blocks. Each signal optionally has an associated set of attributes that correspond to the parameters that make up the content of the message.

Typically, two block instances communicate using *signal events*—instances of signals. The initiating block sends a signal event to a target block. This signal event is defined as part of the supplementary behaviours—described using activities—associated with the initiating state machine: the entry or exit behaviours of the active state or the effect component of the enabled transition. The receipt of the signal event in the target block may subsequently trigger a transition in its state machine. The approach described above is popular when modelling event-based systems.

A *connector* connects two or more parts or references. The connection formally allows the connected components to interact, although the connector does not characterise the nature of the interaction. Instead, the interaction is stipulated by the behaviours of the connected blocks.

State machines allow one to depict state-dependent behaviour in a graphical fashion in terms of nodes and labelled edges: nodes represent states, whereas the edges correspond to transitions between states.

Activities provide the modeller with the ability to describe complex routes (termed *flows*) along which actions execute. In SysML activities, there are two types of flows: control flows and object flows.

We defer detailed discussions on state machines and activities until Sects. 3 and 4, respectively, where the relevant concepts will be introduced alongside the respective formal models.

3 A CSP model for SysML state machines

Our formalisation of state machines can be considered to be a hybrid of the semantics previously presented by Ng and Butler [15] and those previously presented by Bolton Davies [4], and Davies and Crichton [6]. Specifically, we use the semantics of [15] as a basis, extended with an event queuing mechanism proposed in [4] and [6].

3.1 Abstract syntax

Let $\mathcal{M}$ denote the set containing all state machines. In the following, we consider the formalisation as it relates to a single state machine, $M \in \mathcal{M}$.

A state machine describes state-based, event-driven behaviour in terms of a finite collection of states, and transitions between those states. A state machine M is thus an ordered pair (S_M, T_M), where:

- S_M represents the set of states;
- T_M represents the set of transitions.

3.1.1 Transitions

Every transition exists between a source and a target state:

$$\mathit{source} : T_M \rightarrow S_M$$
$$\mathit{target} : T_M \rightarrow S_M$$

A transition consists of a *trigger*, a *guard*[3] and an *effect*:

$$\mathit{trigger} : T_M \rightarrow \mathcal{S}$$
$$\mathit{guard} : T_M \rightarrow \mathcal{B}$$
$$\mathit{effect} : T_M \rightarrow \mathcal{A}$$

In the above, $\mathcal{S}$ is the set of signals and $\mathcal{A}$ is the set of activities. We denote by $\mathcal{B}$ the set $\{\mathit{true}, \mathit{false}\}$.

3.1.2 States

We partition S_M such that:

- S_M^I represents the set of *initial states*;
- S_M^F represents the set of *final states*;
- S_M^T represents the set of *terminate states*;
- S_M^J represents the set of *junction states*;
- S_M^C represents the set of *choice states*;
- S_M^S represents the set of *simple states*; and
- S_M^{SC} represents the set of *simple composite states*.

The aforementioned sets are pairwise disjoint and fully partition S_M.

Every state in our formalisation has a unique name (i.e. name is an injective function):

$$\mathit{name} : S_M \rightarrowtail N_M$$

In the above, N_M denotes the set of state names of state machine M.

Every simple or simple composite state has an optional *entry* and *exit behaviour*; *do* behaviours are excluded from consideration. All state-based behaviours are modelled via activities.

$$\mathit{entry} : S_M \rightarrow \mathcal{A}$$
$$\mathit{exit} : S_M \rightarrow \mathcal{A}$$

In each case, an activity modelling the state-based behaviour is returned.

The function

$$\mathit{outgoing} : S_M \rightarrow \mathbb{P}\, T_M$$

returns the set of outgoing transitions for a given state. The transitions returned are those emanating from the state at a syntactical level, as per the graphical depiction. In other words, the function does not return transitions emanating from states that transitively enclose the input state.

Our formalisation makes use of the concept of a pseudostate. A *pseudostate* is a state in which a state machine may temporarily find itself. For example, all initial, junction and choice states are considered pseudostates.

3.2 Modelling concepts

3.2.1 Enclosing states

A simple composite state has exactly one region:

$$\mathit{region} : S_M^{SC} \rightarrowtail S_M^R$$

Note that the concept of region only exists for composite states. A state machine has an enclosing top state $s \in S_M^{SC}$ at the root of the state hierarchy. This state is a simple composite state. We assume the existence of a function that, given a state machine, returns the top state:

$$\mathit{top} : M \rightarrowtail S_M^{SC}$$

We define a helper function, $\mathit{surround}$, that associates a state with its immediately enclosing state.

$$\mathit{surround} : S_M \nrightarrow S_M$$

The function above is partial by virtue of the fact that no state encloses the top state. We may also wish to consider all states that enclose a particular state. A second helper function

$$\mathit{enclose} : S_M \nrightarrow \mathbb{P}\, S_M$$

returns, for a particular state $s \in S_M$, the set containing those states that immediately or transitively enclose s. Thus

$$\forall\, s : \operatorname{dom} \mathit{enclose} \bullet \mathit{enclose}(s) = \mathit{surround}^{+} (\!|\ \{s\}\ |\!)$$

We require the functions $\mathit{enclose}$ and $\mathit{surround}$ to be anti-reflexive: the hierarchy imposed on states may not include circular definitions where a state is allowed to be its own enclosing state.

$$\forall\, s : \operatorname{dom} \mathit{surround} \bullet s \mapsto s \notin \mathit{surround}$$
$$\forall\, s : \operatorname{dom} \mathit{enclose} \bullet s \mapsto s \notin \mathit{enclose}$$

[3] A guard is more formally a Boolean expression to be evaluated at runtime.

Hierarchical state machines need to take entry and exit behaviours of transitively enclosing/enclosed states into consideration during the firing of transitions.

A *pivot state* is the state, for a given transition, where the nested exit behaviour of the source state has finished execution, and the behaviour of the effect component is executed, before the nested entry behaviour of the target state begins.

We assume the existence of a function, $pivot$:

$$pivot : T_M \rightarrow S_M$$

Before we define $pivot$ precisely, we must introduce the μ-notation. The expression

$$(\mu\, x : X \mid p)$$

denotes the unique object x drawn from the set X such that the predicate p holds.

For a transition, $t \in T_M$, we define a pivot state to be the innermost jointly transitive enclosing state with regard to the source and target states of t. Thus

$$\begin{aligned}&\forall\, t : \mathrm{dom}\, pivot \bullet\\&\quad pivot(t) =\\&\qquad (\mu\, p : S_M \mid p \in innermost\,(\, enclose(source(t))\\&\qquad\qquad\qquad\qquad\qquad\cap\\&\qquad\qquad\qquad\qquad\qquad enclose(target(t))))\end{aligned}$$

In the above, the function

$$innermost : \mathbb{P}\, S_M \rightarrow \mathbb{P}\, S_M$$

returns, given a set of states, the set of states at the innermost level of the state hierarchy. More specifically, $innermost$ returns from the input set of states only those that do not enclose any of the other states:

$$\begin{aligned}&\forall\, S : \mathrm{dom}\, innermost \bullet\\&\quad S = \{top_M\} \Rightarrow innermost(S) = \{top_M\}\\&\quad \wedge\\&\quad S \neq \{top_M\} \Rightarrow\\&\qquad innermost(S) =\\&\qquad\quad \{s_i : S \mid\\&\qquad\qquad (\forall\, s_j : S \mid s_j \neq top_M \bullet s_i \notin enclose(s_j))\}\end{aligned}$$

3.2.2 Example

As an example of the above mathematical constructs, consider Fig. 1: state machine M_{ex} with top state $top_{M_{ex}}$. We have the following.

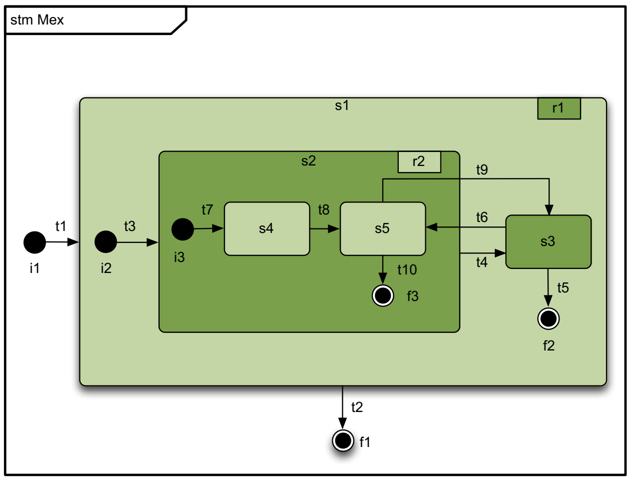

Fig. 1 State machine M_{ex}. M_{ex} is a hierarchical state machine. State s_1 is a simple composite state with a single region, r_1. We have merely labelled the transitions, omitting triggers, guards and effects. Similarly, entry and exit behaviours of the respective states are elided

$$\begin{aligned}&source(t_7) = i_3\\&target(t_7) = s_4\\&outgoing(s_4) = \{t_8\}\\&incoming(s_4) = \{t_7\}\\&surround(s_1) = top_{M_{ex}}\\&enclose(s_5) = \{s_2, s_1, top_{M_{ex}}\}\\&region(s_1) = r_1\end{aligned}$$

Consider transition t_9 of Fig. 1, the source and target states of which are s_5 and s_3, respectively. The sets of states that enclose s_5 and s_3 are the following.

$$\begin{aligned}&enclose(s_5) = \{s_2, s_1, top_{M_{ex}}\}\\&enclose(s_3) = \{s_1, top_{M_{ex}}\}\end{aligned}$$

The function pivot returns the pivot state s_1 of transition t_9.

$$\begin{aligned}&pivot(t_9)\\&\quad = (\mu\, p : S_M \mid\\&\qquad p \in innermost(\{s_2, s_1, top_{M_{ex}}\} \cap \{s_1, top_{M_{ex}}\}))\\&\quad = (\mu\, p : S_M \mid p \in innermost(\{s_1, top_{M_{ex}}\}))\\&\quad = s_1\end{aligned}$$

3.2.3 Behaviours

In our formalisation, all transition effects, entry and exit behaviours are modelled using activities. In general, an activity can be as complex as required, or may, instead, be reduced to a single, simple event. We shall discuss activities formally in Sect. 4.

The aforementioned behaviours are all optional: the effect component of a transition can be omitted if not needed; similarly, entry and exit behaviours are not compulsory. For the case where the SysML behaviour is left unspecified, we make use of the empty activity. This approach is chosen in order to make the CSP formalisation more concise.

The empty activity, process A_ε, is defined thus.

$$A_\varepsilon = \mathit{Skip}$$

The *exit behaviour* is executed upon exiting a state. In particular, this happens after the triggering event, but before the behaviour specified by the effect component of the selected transition. Exit behaviours cannot be interrupted. SysML allows the outgoing transitions of an enclosing state to be triggered from within one of the nested states. In this case, the exit behaviours of the transitively enclosing states will be triggered starting with the innermost active state until we reach the pivot state in the state hierarchy. These transitions that emanate from a composite state are termed *high-level transitions*.

We now formulate a CSP process that models exit behaviour.

Let s be the current active state with outgoing transition t. Furthermore, restrict the source and target states of t to simple or simple composite states. Thus

$$\begin{array}{l} s \in S^S_M \cup S^{SC}_M \\ \wedge \\ t \in \mathit{outgoing}(s) \\ \wedge \\ \mathit{target}(t) \in S^S_M \cup S^{SC}_M \end{array}$$

We can formalise exit behaviour as follows.

$$\begin{array}{l} \mathit{Exit}(s,t) = \mathit{exit}(s) \mathbin{⨟} \mathit{exit}(s_0) \mathbin{⨟} \cdots \mathbin{⨟} \mathit{exit}(s_n) \\ \text{where} \\ \quad \{s_0 \ldots s_n\} = \{s_i : \mathit{enclose}(s) \mid \\ \qquad\qquad s_i \notin \mathit{enclose}(\mathit{target}(t))\} \\ \qquad s_0 = \mathit{surround}(s) \\ \qquad s_n = \mathit{surround}(s_{n-1}) \\ \quad \mathit{pivot}(t) = \mathit{surround}(s_n) \end{array}$$

It is important to note that the above is only defined between states that are not pseudostates. By imposing this restriction, we can be confident that the function exit is defined for every state in the construction above. Moreover, integrating the semantics of high-level transitions with that of complex transitions[4] would unnecessarily complicate the formalisation in CSP.

[4] A complex transition is a transition through a junction or choice pseudostate.

The *entry behaviour* is executed upon entering a state. Again, this behaviour is not susceptible to interruption. After the behaviour of the effect component has finished execution, the nested entry behaviour is executed starting with the outermost state inwards towards the designated target state.

We now turn to model entry behaviour and assume a current active state s with outgoing transition t. Therefore

$$\begin{array}{l} s \in S^S_M \cup S^{SC}_M \\ \wedge \\ t \in \mathit{outgoing}(s) \\ \wedge \\ \mathit{target}(t) \in S^S_M \cup S^{SC}_M \end{array}$$

The entry behaviour can be formalised as follows.

$$\begin{array}{l} \mathit{Entry}(s,t) = \mathit{entry}(s_0) \mathbin{⨟} \cdots \mathbin{⨟} \mathit{entry}(s_n) \\ \text{where} \\ \quad \{s_0 \ldots s_n\} = \{s_i : \mathit{enclose}(\mathit{target}(t)) \mid \\ \qquad\qquad s_i \notin \mathit{enclose}(s)\} \\ \qquad s_n = \mathit{target}(t) \\ \qquad s_{n-1} = \mathit{surround}(s_n) \\ \quad \mathit{pivot}(t) = \mathit{surround}(s_0) \end{array}$$

Again, we insist that the above is defined only for states that are not pseudostates to ensure that the construction is valid.

The *effect* is a behaviour, executed upon the transition between states; in this paper, such behaviours are described via activities. In particular, the behaviour of the effect component is executed after the nested exit behaviours of the source state, but before the nested entry behaviours of the target state. In the state hierarchy, this corresponds to the pivot state. In our formal model, the effect component of a transition t is given by $\mathit{effect}(t)$.

3.2.4 Termination condition

The *termination condition* relates to the generation of a completion event. A completion event is generated only once the relevant completion criteria, defined in terms of the termination condition of the currently active state, have been met. The completion criteria can be summarised thus.

– For a simple state $s \in S^S_M$ the termination condition is, according to the standards [16, 17], the completion of the do behaviour. However, since we do not model do behaviours in our semantics, this condition is trivially satisfied upon entry into a state. We argue that the completion event should therefore be offered along with permitted explicit transitions.

- For a simple composite state $s \in S_M^{SC}$ the termination condition is satisfied, according to the standards [16, 17], if either:
  - the final state of the state machine residing in the region of the composite state is reached; or
  - the do behaviour of s terminates.

  As we do not model do behaviours, it follows that the termination condition is satisfied if and only if the state machine residing in the region of the composite state s has reached a final state.

3.2.5 Transitions

Transitions are key to our formalisation. We elaborate on the different components that constitute a transition and present basic concepts that aid in their formalisation.

The *trigger* is the stimulus that enables a transition to fire. In this paper, we consider triggers to be either signal events or completion events.

- A signal event corresponds to the arrival of an asynchronous message typed by a signal. A signal event is an explicit trigger. As such, we must annotate the trigger component of the transition with the signal that typed the triggering event.
- A completion event is an implicit event that signifies the exit from a state. Diagrammatically, it is annotated on a transition by omitting the trigger component.

In our formal model, implicit (or completion) events are denoted by π. The definitions for implicit and explicit transitions can be stated thus.

$$\begin{aligned}
&\forall\, s : S_M^S \bullet \mathit{explicit}(s) = \\
&\qquad \{t : \mathit{outgoing}(s) \mid \mathit{trigger}(t) \in \mathbb{S} \setminus \{\pi\}\} \\
&\forall\, s : S_M^S \bullet \mathit{implicit}(s) = \\
&\qquad \{t : \mathit{outgoing}(s) \mid \mathit{trigger}(t) = \pi\}
\end{aligned}$$

Consider a state $s \in S_M$ with a single outgoing transition t. Thus

$$t \in T_M \wedge t \in \mathit{outgoing}(s) \wedge \#\mathit{outgoing}(s) = 1$$

We model the CSP process describing s as follows.[5]

$$\mathit{name}(s) = \mathit{trigger}(t) \rightarrow \mathit{name}(\mathit{target}(t))$$

This approach is chosen in order to provide a uniform treatment for all triggering events. Thus, for a transition $t \in T_M$, $\mathit{trigger}(t)$ denotes the trigger, regardless of whether that transition is triggered explicitly or implicitly.

In our formalisation, completion events are modelled using CSP events of the form π. Explicit events correspond to CSP events that reflect the name of the classifying signal, along with any parameters communicated as part of the event; a signal with an argument associated with it will input on the CSP channel with the corresponding name. For example, if the signal is named $S_j \in \mathbb{S}$ has argument a, we use the CSP event $S_j?a$ as the trigger.

The *guard* of a transition must evaluate to true in order for the transition to occur; alternatively, if the guard is false, the triggering event is consumed without effect.

Consider a state $s \in S_M$ with a single outgoing transition t. Thus

$$t \in T_M \wedge t \in \mathit{outgoing}(s) \wedge \#\mathit{outgoing}(s) = 1$$

We model the CSP process describing s as follows.[6]

```
$name(s) =$
  $trigger(t) \rightarrow$
    if $guard(t)$ then
      $name(target(t))$
    else
      $name(s)$
```

In the above, the evaluation of the guard following the triggering event determines the next active state.

A guard is evaluated based on the arguments that are served up along with the triggering event. For example, if we had the event $S_j?a$ as a trigger, the a can be used in the evaluation of the guard. To preserve the clarity of the presentation, we simply write $\mathit{guard}(t)$ to denote the evaluation of the guard. Recall that a guard is a Boolean expression to be evaluated at runtime. In the above example, the a can be used as part of an expression to be evaluated at runtime—the result of which will be one of true or false. Typically, the expression to be evaluated will involve testing the value passed as part of the triggering event for equality/inequality to known constants in the model (modelled using enumerations).

The combined behaviour of a transition can now be modelled. Consider a state with a single high-level transition t that contains a trigger, a guard and an effect; the high-level transition is between nonpseudostates.

[5] We ignore the guard and effect components for now, as well as all state-based behaviours.

[6] Again, we ignore the effect component for a moment, as well as all state-based behaviours.

```
name(s) =
  trigger(t) →
    if guard(t) then
      Exit(s, t) ⨾
      effect(t) ⨾
      Entry(s, t) ⨾
      name(target(t))
    else
      name(s)
```

Alternatively, if either the source or target state is a *pseudostate*, we know that no transitions crossing regional boundaries are permitted. It follows that entry and exit behaviours, where applicable, reduce to $entry(target(t))$ or $exit(s)$, respectively.

3.2.6 Mapping function

We make use of a mapping function $\mathcal{F}$ that maps the structural constructs to their CSP counterparts. Broadly speaking: each state in SysML corresponds to a process in CSP; and each SysML event corresponds to a CSP event. Thus, the idea is that the mapping rules take us from a given SysML state to the next: in CSP, this corresponds to initially behaving like one process, and then behaving like another. In each state, the CSP process behaves like the state machine would in the corresponding SysML state. Every rule, $\mathcal{F}(M, s)$, is defined such that it describes the behaviour of state machine M at state s. These rules define process definitions, where each state is represented by a CSP process. In CSP, the name of the process describing state s is given by $name(s)$; the behaviour of this process will be given by $\mathcal{F}(M, s)$. Therefore

$$name(s) = \mathcal{F}(M, s)$$

The mapping rules for state machine M start from the initial state contained in the region of the top-level state, $top(M)$. This approach is based on that taken by Ng and Butler [15].

3.3 Behavioural semantics

This section outlines an approach to integrate SysML state machines and CSP by providing a behavioural semantics for the former in terms of the latter. Throughout, we make use of the syntactical structures and modelling constructs of Sect. 3.1.

The behavioural semantics of state machines, as defined in this paper, is based on the type of the currently active state. As such, the outline of our presentation mirrors this by providing a formalisation for each of the different state types in the remainder of this section.

3.3.1 Initial state

An *initial state* designates the first active state of state machine M. Consider an initial state, $s \in S_M^I$, with a lone outgoing transition, $t \in outgoing(s)$. A completion event, which serves as an implicit trigger, is generated upon entry to the *pseudostate*. Recall that *effect* returns the empty behaviour if no effect component is defined for t. The CSP process modelling s follows.

```
F(M, s) =
  trigger(t) →
    effect(t) ⨾ entry(target(t)) ⨾ F(M, target(t))
```

Because no explicit triggers are allowed, only an implicit trigger is generated and $trigger(t) = \pi$. An explicit transition is not allowed due to the fact that a state machine is not allowed to linger in a pseudostate. Similarly, a state machine is not allowed to be hindered from transitioning out of the initial state by means of a guard that evaluates to false—guards are therefore not allowed.

We insist on the presence of a unique initial state in every region. This rule ensures that the state machine residing in the said region is well defined in the scenario where it is entered by default: an incoming transition ends on the state hosting the region.

$$\forall s_r : S_M^R \bullet (\exists_1 s_i : S_M^I \bullet surround(s_i) = s_r)$$

3.3.2 Final and terminate states

A *final state* indicates the termination of a region. Subsequently, it has no outgoing transitions. Once a final state is reached, a completion event is generated to indicate the completion of all behaviour of the containing region, and, thus, the termination of the state machine. There is, however, no requirement for a region to contain a final state. For example, a state machine that never terminates would not have a final state.

Let $s \in S_M^F$ be a final state. If the final state resides within a simple composite state and there is at least one outgoing transition emanating from any of the transitively enclosing states, that is

$$\begin{array}{l} surround(s) \in S_M^{SC} \\ \wedge \\ \exists s' : enclose(s) \bullet outgoing(s') \neq \emptyset \end{array}$$

then reaching the final state represents the termination of the state machine in the region of the state $surround(s)$. The completion transition emanating from $surround(s)$, if present, or any of the explicit transitions emanating from transitively enclosing states, may be taken. Completion transitions emanating from transitively enclosing states other than $surround(s)$ may only be taken once their respective final states are reached.

```
$\mathcal{F}(M, s) =$
  $\Box\, s' : enclose(s) \bullet$
    $\Box\, t : explicit(s') \bullet$
      $trigger(t) \rightarrow$
        if $guard(t)$ then
          $Exit(s, t)\ ⨾$
          $effect(t)\ ⨾$
          $Entry(s, t)\ ⨾$
          $\mathcal{F}(M, target(t))$
        else
      $\mathcal{F}(M, s)$
  $\Box$
  $\sqcap\, t : implicit(surround(s)) \bullet$
    $trigger(t) \rightarrow$
      $Exit(s, t)\ ⨾$
      $effect(t)\ ⨾$
      $Entry(s, t)\ ⨾$
      $\mathcal{F}(M, target(t))$
```

Alternatively, if the final state resides within a simple composite state and there are no outgoing transitions emanating from any of the transitively enclosing states, that is

$$surround(s) \in S_M^{SC} \\ \wedge \\ \forall\, s_e : enclose(s) \bullet outgoing(s_e) = \emptyset$$

then reaching the final state denotes the termination of M:

$$\mathcal{F}(M, s) = Skip$$

Reaching a final state in a nonhierarchical state machine is a special case of the last scenario.

The formalisation of a *terminate state* is trivial: the state machine behaves as the CSP process $Skip$.

$$\mathcal{F}(M, s) = Skip$$

3.3.3 Junction and choice states

A *junction state* is a transient point along a compound transition. The first leg of the compound transition contains the trigger and optional guard; note that an effect component is not allowed. Similarly, on the second leg, a trigger component is not allowed due to the fact that a complex transition must be selected in response to a single event. As the triggering event is assumed to have occurred on the first leg of the transition, it is disallowed for the second. Guard and effect components are allowed for the second leg of the compound transition.

A *choice state* differs from a junction state with regard to the first leg of the compound transition in that it additionally allows for an effect component. The consequence of this is that the effect component can influence the outcome of the guards on the second leg. In contrast, a junction state does not permit an effect component on the incoming transition, and, as such, the guards on the second leg are in principle evaluated the instant the triggering event is served up for processing. However, as our formalisation is centred around the outgoing transitions, there is no difference between the CSP formalisation for a junction state and that of a choice state. The distinction, however, would be made during the formalisation of the source state of the first leg of the compound transition.

Because junction and choice states are transient points along a complex transition, we do not allow explicit events to be triggered from the transitively enclosing states. Explicit events from the states that transitively enclose the junction or choice state can only be triggered once both legs of the compound transition have been taken. This semantics is in line with the notion that these are merely pseudostates along a compound transition, and that the transition must be completed before other events can be triggered.

The process modelling the junction or choice state is instantiated with the arguments passed on the trigger of the first leg of the complex transition. These arguments are used to evaluate the guards of the transitions that emanate from the state, with the parameters indicated between the square brackets following the mapping function.

```
$\mathcal{F}(M, s)[params(trigger('t))] =$
  $\Box\, t : outgoing(s) \bullet$
    (if $eval(guard(t), params(trigger('t)))$ then
      $effect(t)\ ⨾$
      $entry(target(t))\ ⨾$
      $\mathcal{F}(M, target(t))$
    else
      $Stop$)
```

Note that in our formulation the transitions that terminated and emanated from junction or choice states are not allowed to cross-regional boundaries. In the above, the formal parameters of the signal that typed the send signal event that served as the triggering event correspond to $params(trigger('t))$. The transition $'t$ is that of the first leg of the compound transition. In addition, $eval(guard(t), params(trigger('t)))$ denotes the evaluation of the guard condition of t, by substituting the associated parameters of $trigger('t)$ in the expression $guard(t)$.

3.3.4 Simple state

A *simple state* $s \in S^S_M$ does not contain a region, and, as such, cannot host nested states. The behaviour of a simple state is formalised in terms of its own outgoing transitions, and the explicit transitions of the states that transitively enclose it.

```
$\mathcal{F}(M,s) =$
  $\sqcap\, t : implicit(s) \bullet$
    $trigger(t) \rightarrow$
      if $guard(t)$ then
        $Exit(s,t)\ ⨾$
        $effect(t)\ ⨾$
        $Entry(s,t)\ ⨾$
        $\mathcal{F}(M, target(t))$
      else
        $\mathcal{F}(M,s)$
  $\square$
  $\square\, t : explicit(s) \bullet$
    $trigger(t) \rightarrow$
      if $guard(t)$ then
        $Exit(s,t)\ ⨾$
        $effect(t)\ ⨾$
        $Entry(s,t)\ ⨾$
        $\mathcal{F}(M, target(t))$
      else
        $\mathcal{F}(M,s)$
  $\square$
  $\square\, s' : enclose(s) \bullet$
    $\square\, t : explicit(s') \bullet$
      $trigger(t) \rightarrow$
        if $guard(t)$ then
          $Exit(s,t)\ ⨾$
          $effect(t)\ ⨾$
          $Entry(s,t)\ ⨾$
          $\mathcal{F}(M, target(t))$
        else
          $\mathcal{F}(M,s)$
```

The above composition can be deconstructed as follows.

- The process offers the nondeterministic choice between all completion or implicit transitions. The indexed form of the choice operator is used as it is possible for a state to have more than one completion transition.
- The process offers the deterministic choice between all explicit, outgoing transitions of s. Deterministic choice is selected as the transition taken is dependent upon the triggering signal; the decision as to which transition to take is not internalised as is the case for completion transitions. Thus, all possible transitions ought to be offered.
- For transitively enclosing states of s, we permit only explicit transitions. Transitions emanating from the states that transitively enclose s are allowed to trigger, while s is active. When a higher level transition fires while s is active, the state s is exited and the transition followed.

Each of the above corresponds to a different scenario under which s might be exited. However, the behaviour once an exit is triggered remains the same, as described by the expositions following the guard.

3.3.5 Simple composite state

A *simple composite state* $s \in S^{SC}_M$ has a single region which contains nested states. For that region, only one of the nested substates is allowed to be active. Recall that each region has a unique initial state. The behaviour of the composite state is described by the behaviours of the states that it contains. It follows that a transition to a composite state is equivalent to a transition to the initial state of that composite state. The composite state thus behaves like the initial state that it contains. The subsequent behaviour is then completely described by the target state of the initial state, and so forth. This is by virtue of the fact that our formalisation for noncomposite states, like initial, final and simple states incorporate the transitions of the states that transitively enclose them.

Given $s_i \in (surround^{-1}\llparenthesis\, \{s\}\, \rrparenthesis) \cap S^I_M$, the unique initial state of s, the formalisation is as follows.

$$\mathcal{F}(M,s) = \mathcal{F}(M,s_i)$$

3.3.6 Execution environment

Run-to-completion semantics The execution semantics for SysML state machines is as follows. A *run-to-completion semantics* is defined: a state machine is only permitted to consume a single triggering event, namely the *current event*, at any one instant and must do so until processing of the said event is complete. For this purpose, each state machine has an associated *event queue*.

- An event is *received* when it is accepted and waiting for processing. The event is placed at the back of the event queue.
- The event at the front of the queue is removed and presented to the state machine. At this instant, it becomes the current event. We say the event is *dispatched*.
- The state machine finishes the processing of the current event, uninterrupted, and until completion. Once this has happened, the event is *consumed*. A consumed event is no longer available for processing.

In the formalisation, we assume a FIFO queue, although the standard [16] does not define the order of dequeuing.

Event queue We model the event queue of a state machine using a CSP process called EQ that communicates on a channel $queue$.

$$EQ = queue?e \rightarrow local?p!e \rightarrow EQ$$

The above formulation is in essence a buffer process that sits between the state machine and its external environment. A triggering event e is communicated along channel $queue$ by the external environment. The event is then consumed (and therefore removed from the buffer) by the state machine on channel $local$. An event can be consumed in one of two ways.

– The event can be processed: the current active state has an outgoing transition with a triggering event that corresponds to the event served up for processing. In this case, the event takes the form $local.proc.e$.
– The event can be discarded: the current active state has no outgoing transition corresponding to the event served up for processing. In this case, the event takes the form $local.disc.e$.

The event queue is willing to communicate either event by inputting on the channel; the events offered by the current active state will determine whether the event is processed or discarded.

Here, we assume a queue with a maximum capacity of 1; the queue blocks when full. Nonblocking semantics, where events are discarded when the queue becomes full, is conceivable; so are event queues with different capacities. A semantics with an unbounded queue is also conceivable, although this is not finite state, and therefore not amenable to verification with FDR3. However, animators can be used to explore the behaviour in the case of unbounded event queues.

The process definitions, as they stand in Sect. 3.3, are not entirely complete: we need to adapt these to consider whether a particular event should be processed or discarded, depending on the current active state of the state machine M. In particular, triggering events need to be extended to include communication on the channel $local$[7]—an event e that is processed takes the form $local.proc.e$ or $local.disc.e$, depending on whether the event is processed or discarded at the time it is served up for consumption (as determined by the active state of the state machine).

[7] The channel $local$ is a parameter of the process modelling M.

Thus, we need to consider the eventuality where the state machine M receives a signal event not expected in the current state $s \in S_M$: that is, an instance of a signal event $S_j \in \mathcal{S}$ such that $S_j \notin \{t : outgoing(s) \bullet trigger(t)\}$. Here, the state machine discards the unexpected event.

Assume that the function

$$unexpected : S_M \rightarrow \mathbb{P}\,\mathcal{S}$$

returns the set of unexpected events for state $s \in S_M$. The interpretation of unexpected events depends on the type of the current state.

If the current state is a pseudostate, that is

$$s \in \bigcup\{S_M^I, S_M^F, S_M^T, S_M^J, S_M^C\}$$

then unexpected events are receive signal events that are not outgoing transitions of the current state s.

Alternatively, if the current state is a nonpseudostate, that is

$$s \in \bigcup\{S_M^S, S_M^{SC}\}$$

then unexpected events are receive signal events that are not contained within the outgoing transitions of the transitively enclosing simple or simple composite states of s, and outgoing transitions of s itself.

Thus,

$$\begin{array}{l}\forall\, s : \mathrm{dom}\, unexpected \bullet \\ \quad s \in \bigcup\{S_M^I, S_M^F, S_M^T, S_M^J, S_M^C\} \Rightarrow \\ \qquad unexpected(s) = \\ \qquad\quad \{S_j : \mathcal{S} \mid S_j \notin \{t : outgoing(s) \bullet trigger(t)\}\} \\ \quad \wedge \\ \quad s \in \bigcup\{S_M^S, S_M^{SC}\} \Rightarrow \\ \qquad unexpected(s) = \\ \qquad\quad \{S_j : \mathcal{S} \mid S_j \notin \{t : (s) \bullet trigger(t)\}\}\end{array}$$

State machine The overall state machine is modelled as a single process that contains, as appropriate, localised process descriptions corresponding to the state machine's syntactic structure, with the overall structure being similar to that described in [6]. The environment external to the state machine communicates with the event queue. The state machine receives all communications via its associated event queue, which is modelled as a CSP buffer of size 1. It communicates with this buffer on a CSP channel, $local$. Each of the localised processes has access to this channel in order to receive communications from the event queue.

The process that models the state machine, M, first behaves as that associated with the initial state, $\mathcal{F}(M, i)$, and then as each of the subsequent processes, until it terminates in state f. The local process EQ models the event queue.

```
$M(queue, local) =$
  let
    $\mathcal{F}(M, i) = \ldots$
    $\vdots$
    $\mathcal{F}(M, f) = Skip$
    $EQ = queue?e \rightarrow local?p!e \rightarrow EQ$
  within
    $\mathcal{F}(M, i) \,[|\, \{|\, local \,|\} \,|]\, EQ$
```

In the above, the local definitions $\mathcal{F}(M,i) \ldots \mathcal{F}(M,f)$ take into consideration whether an event will be processed or discarded.

In Sect. 5.3, we discuss the communication semantics for multiple state machines, as well as integration with other constructs, such as activities.

4 A CSP model for SysML activities

4.1 Abstract syntax

Recall that $\mathcal{A}$ denotes the set containing all activities. We consider the formalisation as it relates to a single activity $A \in \mathcal{A}$. An activity A is a quintuple

$$(A_A, R_A, P_A, CF_A, OF_A)$$

where:

- A_A denotes the set of action nodes;
- R_A denotes the set of routing nodes;
- P_A denotes the set of parameter nodes;
- CF_A denotes the set of control flows; and
- OF_A denotes the set of object flows.

An activity defines flow-based behaviour in terms of nodes and edges: the nodes are represented by the sets A_A, R_A and P_A; the edges are denoted by CF_A and OF_A.

4.1.1 Nodes

Actions are the building blocks of activities. The set of action nodes A_A is further partitioned such that:

- A_A^{SS} denotes the set of send signal event actions;
- A_A^{RS} denotes the set of receive signal event actions;
- A_A^{VS} denotes the set of value specification actions;
- A_A^{O} denotes the set of opaque actions; and
- A_A^{CB} denotes the set of call behaviour actions.

The aforementioned sets are pairwise disjoint and fully partition A_A.

Routing nodes are used to model complex flow-based logic. The set of control nodes R_A is partitioned thus:

- R_A^I denotes the set of initial nodes;
- R_A^F denotes the set of final nodes;
- R_A^{FK} denotes the set of fork nodes;
- R_A^{JN} denotes the set of join nodes;
- R_A^D denotes the set of decision nodes; and
- R_A^M denotes the set of merge nodes.

The aforementioned sets are pairwise disjoint and fully partition R_A.

Activity parameter nodes, denoted by P_A, are used to model the input and output arguments of activities. For our purposes, we only consider activity parameter nodes that serve as inputs to activities; this is partly due to a desire to keep the semantics elegant and partly due to the context in which we use activities. We assume that a single argument is passed per node.

4.1.2 Edges

An *object flow* is used to model the passage of a parameter between two nodes: the parameter flows from the source or outputting node to the target or inputting node. We define the following functions for an object flow $of \in OF_A$, to yield the outputting and inputting object node, $source_{of}$ and $target_{of}$, respectively.

$$\begin{aligned} source_{of} &: OF_A \rightarrow \bigcup\{\, A_A^{RS}, A_A^{VS}, R_A^{FK}, R_A^{JN}, \\ &\qquad R_A^{D}, R_A^{M}, P_A\} \\ target_{of} &: OF_A \rightarrow \bigcup\{\, A_A^{SS}, A_A^{CB}, R_A^{FK}, \\ &\qquad R_A^{JN}, R_A^{D}, R_A^{M}\} \end{aligned}$$

Note that the aforementioned are total functions: every object flow must have an associated source (or target) node.

The outgoing edges of a decision node have mutually exclusive guards that determine the next flow to be taken. The function

$$guard : OF_A \nrightarrow \mathcal{B}$$

denotes the evaluation of the guard of an object flow emanating from a decision node. The guard is evaluated based on the value of the object passed along the incoming flow of the decision node.

Control flows model the passage of control between the constituent nodes of an activity. Control flows are permitted between action and routing nodes. We define the following functions for a control flow, $cf \in CF_A$, to yield the source and target node, respectively.

$$source_{cf} : CF_A \rightarrow \bigcup\{ (A_A \backslash A_A^{VS}), R_A^I, R_A^{FK}, R_A^{JN}, R_A^M \}$$

$$target_{cf} : CF_A \rightarrow \bigcup\{ (A_A \backslash A_A^{VS}), R_A^F, R_A^{FK}, R_A^{JN}, R_A^M \}$$

Guards on control flows are not modelled. Our formalisation of state machines evaluated guards based on the value of a parameter passed with the triggering event. In SysML, guards on control flows are used in a more general sense and typically expressed in natural language; however, we would argue that interpreting guards using natural language is unsuitable in a formal framework, where a precise meaning is desirable. Moreover, we seek to provide a semantics where guards are interpreted in a uniform manner. Activities typically execute within the context of state machines, and we will therefore interpret the guards analogously.

4.2 Modelling concepts

4.2.1 Outgoing edges

The function

$$outgoing_{of} : \bigcup\{ A_A^{RS}, A_A^{VS}, A_A^{CB}, R_A^{FK}, R_A^{JN}, R_A^D, R_A^M, P_A \} \rightarrow \mathbb{P}\, OF_A$$

returns the set of outgoing object flows for certain types of nodes.

We also define a similar auxiliary function that returns, for certain types of nodes, the set of outgoing control flow edges:

$$outgoing_{cf} : \bigcup\{ (A_A \backslash A_A^{VS}), R_A^I, R_A^{FK}, R_A^{JN}, R_A^M \} \rightarrow \mathbb{P}\, CF_A$$

We require an action to either have an outgoing object flow or an outgoing control flow, but not both:

$$\begin{aligned} &\forall\, a : \mathrm{dom}\, outgoing_{of} \cap \mathrm{dom}\, outgoing_{cf} \bullet \\ &\quad outgoing_{of}(a) \neq \emptyset \Rightarrow outgoing_{cf}(a) = \emptyset \\ &\quad \wedge \\ &\quad outgoing_{cf}(a) \neq \emptyset \Rightarrow outgoing_{of}(a) = \emptyset \end{aligned}$$

4.2.2 Mapping function

To define behavioural semantics for activities, we once again make use of a mapping function, $\mathcal{F}$, that maps the structural constructs to their CSP counterparts. Every node in SysML corresponds to a process in CSP; similarly, every edge has an associated CSP process. The idea is that the mapping rules take us from the SysML source node to the target node: in CSP, this entails initially behaving like the source process, then behaving like the edge connecting the source and target nodes and then behaving like the target node. The behaviour of the edge is usually the behaviour of its target node; an object flow edge has an additional process parameter to model the flow of objects. Every rule, $\mathcal{F}(A, c)$, is defined such that it describes the behaviour of activity A at a particular construct c. These constructs are the nodes and edges of A. The mapping rules for activity A start from the initial node.

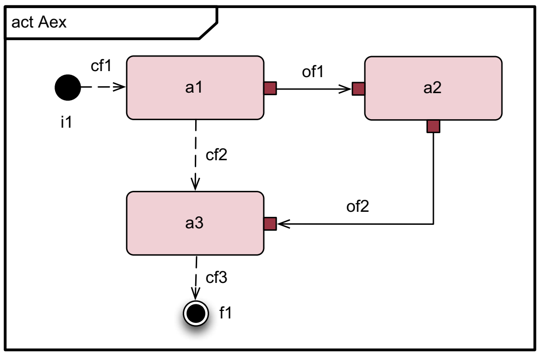

Fig. 2 Activity A_{ex}. A_{ex} is not allowed in our formalisation due to the fact that action a_1 has an outgoing control flow cf_2 as well as an outgoing object flow of_1. However, A_{ex} would have exactly the same behaviour if cf_2 was removed

4.2.3 Example

Refer to Fig. 2 and activity A_{ex}. We have

$$\begin{aligned} A_{A_{ex}} &= \{a_1, a_2, a_3\} \\ R_{A_{ex}} &= \{i_1, f_1\} \\ OF_{A_{ex}} &= \{of_1, of_2\} \\ CF_{A_{ex}} &= \{cf_1, cf_2, cf_3\} \end{aligned}$$

4.3 Behavioural semantics

The behavioural semantics of activities, as defined in this paper, is based upon the type of the currently active node. As such, the outline of our presentation mirrors this by providing a formalisation for each of the different node types in the remainder of this section. The semantics further distinguishes and provides alternative interpretations based on the type of the connecting edges, activity parameter nodes and value specification actions.

4.3.1 Control flow

Control flows are used to route the flow of control between different nodes of an activity. In our formalisation, a control flow $cf \in CF_A$ can be thought of as a CSP process. The behaviour of this process is dependent on the target node of the control flow, given by $target_{cf}(cf)$. If the target is not a join node, i.e. $target_{cf}(cf) \notin R_A^{JN}$, the process simply designates its behaviour to be that of the target node.

$$\begin{array}{ll} \mathcal{F}(A, cf) = & \\ \quad \mathcal{F}(A, target_{cf}(cf)) & \text{if } target_{cf}(cf) \notin R_A^{JN} \\ \quad Join(cf) & \text{otherwise} \end{array}$$

In the case where $target_{cf}(cf) \in R_A^{JN}$, there will be $k-1$ other control flows that terminate in the same join node.[8]

Let the control flows be $cf_0 \mathinner{..} cf_{k-1}$. Exactly one of the control flows, cf_0, will exhibit the behaviour of the join node.

$$\begin{array}{ll} Join(e) = & \\ \quad join \rightarrow Skip & \text{if } e \neq cf_0 \\ \quad join \rightarrow \mathcal{F}(A, target_{cf}(e)) & \text{otherwise} \end{array}$$

The above construction ensures that exactly one of the previously forked flows continues after the join node.

4.3.2 Object flow

Object flows model the flow of objects between the different nodes in an activity. In our formalisation, an object flow $of \in OF_A$ behaves similarly to a control flow edge. However, the value of the object being passed along the flow is passed as a process argument between the processes representing the different constructs. We do not permit object flows to terminate on or emanate from join nodes (see the formalisation of join nodes). Thus, the behaviour of an object flow is given by the behaviour of the node that it terminates on, $target_{of}(of)$.

The process modelling the object flow simply designates its behaviour to be that of the target node, passing the object o as a process parameter.

$$\mathcal{F}(A, of)[o] = \mathcal{F}(A, target_{of}(of))[o]$$

In the above, the CSP process to the left is initialised by the source node of object flow of, $source_{of}(of)$.

4.3.3 Initial node

An *initial node* $i \in R_A^I$ is a routing node that designates the starting point of an activity A. Subsequently, only a single control flow is permitted to emanate from i; object flows are disallowed and can neither emanate from, nor terminate on i. Let the unique control flow be

$$cf = (\mu f : CF_A \mid f \in outgoing_{cf}(i))$$

The formalisation of the initial node follows.

$$\mathcal{F}(A, i) = \mathcal{F}(A, cf)$$

Only one initial node is permitted per activity.

4.3.4 Final node

A *final node* $f \in R_A^F$ has no outgoing edges and a single incoming control flow edge. It is modelled trivially as the CSP process $Skip$.

$$\mathcal{F}(A, f) = Skip$$

4.3.5 Send signal event

A *send signal event* action is used as a means of communication between different activities that execute within the context of state machines; the action corresponds to the sending of a signal event. A send signal event action node is entered either via a control flow or an object flow edge; in the case where the node is entered via a control flow an incoming object flow is possible, provided the object flow emanates from a parameter node. Note that a send signal event action is always exited via a control flow edge.

A send signal event action has an input pin corresponding to the attribute of the signal to be sent[9] and one input pin to specify the target for the signal.

Entry via control flow A send signal event action $ss \in A_A^{SS}$, entered via a control flow edge, with a single outgoing control flow,

$$cf = (\mu f : CF_A \mid f \in outgoing_{cf}(ss))$$

can be formalised thus:

$$\mathcal{F}(A, ss) = target(ss).signal(ss) \rightarrow \mathcal{F}(A, cf)$$

A send signal event action has an input pin that names the target of the send signal action. In CSP, this corresponds to the channel name used to communicate with the target state machine, denoted by $target(ss)$ in the above. The name of the signal event is given by $signal(ss)$.

Optionally, an incoming object flow of is possible. This serves as input to the send signal event action and models the parameters sent as part of the send signal event. In our semantics, the object flow of, if present, emanates from an activity parameter node $p \in P_A$ and terminates on the send signal event node ss; ss is therefore not entered via of, but

[8] We only consider simple fork/join constructs.

[9] No input pin is present if the signal does not have an associated attribute.

of is instead used to pass a value to be used as part of the action. Note that an incoming control flow is still present and also terminates on ss. The construction $par(p)$ is the parameter available within the context of the owning activity, as defined per the arguments of the process modelling the activity.

$$\mathcal{F}(A, ss) = target(ss).signal(ss).par(p) \rightarrow \mathcal{F}(A, cf)$$

Entry via object flow Consider a send signal event action $ss \in A_A^{SS}$, entered via an object flow edge, with a single outgoing control flow

$$cf = (\mu f : CF_A \mid f \in outgoing_{cf}(ss))$$

In this case, the process modelling the send signal event action would have an input argument, p, passed from the process modelling the incoming object flow.

$$\mathcal{F}(A, ss)[p] = target(ss).signal(ss).p \rightarrow \mathcal{F}(A, cf)$$

4.3.6 Receive signal event

A *receive signal event* node represents the action of receiving a signal event. A receive signal event action node can only be entered via a control flow, but may be exited either via a control or via an object flow.

A receive signal event action may output the received signal attribute on an output pin. An input pin is used to specify the source of the event.

Exit via control flow A receive signal event node is typically exited via a control flow edge if the receive signal event has no associated attributes. Assume a receive signal event action $rs \in A_A^{RS}$ with a single outgoing control flow

$$cf = (\mu f : CF_A \mid f \in outgoing_{cf}(rs))$$

and no signal attributes associated with the signal. In the following, $source(rs)$ denotes the source specified for the receive signal event.

$$\mathcal{F}(A, rs) = source(rs).signal(rs) \rightarrow \mathcal{F}(A, cf)$$

If a signal attribute is present, we have an input on the CSP channel corresponding to the attribute a.

$$\mathcal{F}(A, rs) = source(rs).signal(rs)?a \rightarrow \mathcal{F}(A, cf)$$

Exit via object flow A receive signal event action node can only be exited via an object flow if the signal has attributes that are output along the object flow. Assume a receive signal event $rs \in A_A^{RS}$ with a single outgoing object flow

$$of = (\mu f : OF_A \mid f \in outgoing_{of}(rs))$$

and an associated attribute a.

$$\mathcal{F}(A, rs) = source(rs).signal(rs)?a \rightarrow \mathcal{F}(A, of)[a]$$

The attribute a is passed along the object flow by instantiating the process modelling the flow correspondingly.

4.3.7 Value specification action

A *value specification action* is a primitive action that outputs a constant value on its output pin. A value specification node is always entered via a control flow edge. Let $vs \in A_A^{VS}$ be a value specification action with outgoing object flow

$$of = (\mu f : OF_A \mid f \in outgoing_{of}(vs))$$

We have

$$\mathcal{F}(A, vs) = \mathcal{F}(A, of)[value(vs)]$$

In the above, $value(vs)$ denotes the value output by the action.

4.3.8 Opaque action

An *opaque action* is an action executed in a language external to SysML. An opaque action may optionally take an input and, after executing the action, produce an output. These opaque actions are modelled as CSP events.

There are three possible formalisations for opaque actions permitted in our formalisation. We consider each in turn.

Entry and exit via control flows Assume the existence of an opaque action $oa \in A_A^{O}$ node entered via an incoming control flow, and an outgoing control flow

$$cf = (\mu f : CF_A \mid f \in outgoing_{cf}(oa))$$

In the following, $action(o)$ denotes the event corresponding to the opaque action of node oa.

$$\mathcal{F}(A, oa) = action(oa) \rightarrow \mathcal{F}(A, cf)$$

Entry and exit via object flows Assume the existence of an opaque action $oa \in A_A^O$ node with an incoming object flow and an outgoing object flow

$$of = (\mu f : OF_A \mid f \in outgoing_{of}(oa))$$

In the composition below, o denotes the object that serves as input to the opaque action, passed via the object flow.

$$\mathcal{F}(A, oa)[o] = action(oa) \rightarrow \mathcal{F}(A, of)[g(o)]$$

It is possible to view opaque actions as functional: the action takes an input and produces an output. In CSP, this can be modelled as a function g.

Entry via object flow; exit via control flow Assume the existence of an opaque action $oa \in A_A^O$ node with an incoming object flow passing object o and an outgoing control flow

$$cf = (\mu f : CF_A \mid f \in outgoing_{cf}(oa))$$

We have

$$\mathcal{F}(A, oa)[o] = action(oa) \rightarrow \mathcal{F}(A, cf)$$

4.3.9 Call behaviour action

A *call behaviour action* allows an activity to call another behaviour as one of its actions. While this can be any behavioural formalism of SysML, we consider only the calling of activities. We restrict the activities used as call behaviour actions to contain only input pins; output pins are not permitted. Alternatively, a call behaviour action node may be entered via a control flow. In our model, call behaviour nodes are always exited using control flows.

Entry via control flow Consider a call behaviour action $cb \in A_A^{CB}$ with an incoming control flow and a single outgoing control flow

$$cf = (\mu f : CF_A \mid f \in outgoing_{cf}(cb))$$

The construction $behaviour(cb)$ represents the CSP process modelling the activity specified as part of the call behaviour action cb.

$$\mathcal{F}(A, cb) = behaviour(cb) \mathbin{⨾} \mathcal{F}(A, cf)$$

Entry via object flow Consider a call behaviour action $cb \in A_A^{CB}$ with an incoming object flow and a single outgoing control flow

$$cf = (\mu f : CF_A \mid f \in outgoing_{cf}(cb))$$

The object o is passed to the CSP process $behaviour(cb)$.

$$\mathcal{F}(A, cb)[o] = behaviour(cb)[o] \mathbin{⨾} \mathcal{F}(A, cf)$$

4.3.10 Fork node

A *fork node* splits a single flow into multiple separate flows. The flows can be either control flows or object flows. However, if the flow terminating on a fork node is a control flow, then control flows must exit the node; similarly, if an object flow terminates on the fork node, then multiple object flows must leave the fork node.

Control flows A fork node $fk \in R_A^{FK}$ operating on control flows splits the incoming control flow in k separate outgoing flows $cf_0 \mathinner{..} cf_{k-1}$. The alphabetised indexed parallel construction ensures that all the different threads of control only synchronise on the *join* event; all other events are interleaved.

$$\mathcal{F}(A, fk) = [|join|] \; cf : outgoing_{cf}(fk) \bullet \mathcal{F}(A, cf)$$

Object flows The formalisation for object flows is similar. The difference is that the object is passed to each of the k separate forked flows. A fork node $fk \in R_A^{FK}$ operating on object flows splits the incoming object flow in k separate outgoing object flows $cf_0 \mathinner{..} cf_k$.

$$\begin{aligned}&\mathcal{F}(A, fk)[o] = \\ &\quad [|join|] \; of : outgoing_{of}(fk) \bullet \mathcal{F}(A, of)[o]\end{aligned}$$

4.3.11 Join node

A *join node* synchronises previously forked flows: it has k incoming flows and a single outgoing flow. We only formalise join nodes that operate on control flows. The reason for this is that the semantics of join nodes that operates on object flows does not sit well with our formalisation. The object flows in our semantics essentially deal with process arguments, i.e. concrete values. However, token flow semantics on object flows via join nodes requires that each token on every flow is passed to the outgoing flow. This would be inconsistent with our treatment of object flows and are thus excluded.

A join node $jn \in R_A^{JN}$ synchronises k parallel control flows and has a single outgoing control flow, cf:

$$cf = (\mu f : CF_A \mid f \in outgoing_{cf}(jn))$$

Recall that only one of the previously forked flows will behave as the join node.

$$\mathcal{F}(A, jn) = \mathcal{F}(A, cf)$$

4.3.12 Decision node

A *decision node* offers the choice between possible alternative flows, based on the evaluation of guards. A decision node has an incoming flow edge and several outgoing flow edges, with mutually exclusive guards placed on the outgoing edges. However, only one of the outgoing edges—the edge where the guard evaluates to true—is taken. Our model only incorporates object flows via decision nodes. This is intrinsically linked with the evaluation of the guards of the outgoing edges.

A decision node $d \in R_A^D$ over object flows passes an object along one of several possible object flows. The k alternative outgoing object flows $of_k \in outgoing_{of}(d)$ are evaluated. The following assumes mutually exclusive guards. A guard may contain an else clause, in which case it is trivially mapped to a CSP else construct. Machine-readable CSP has only an *if then else* conditional construct; we are therefore forced to adapt the formalisation to include the process $Stop$, as presented below.

```
$\mathcal{F}(A,d)[o] =$
  if $guard(of_0)$ then
    $\mathcal{F}(A,of_0)[o]$
  $\vdots$
  else if $guard(of_k)$ then
    $\mathcal{F}(A,of_k)[o]$
  else
    $Stop$
```

4.3.13 Merge node

A *merge node* has several incoming flows, but only one outgoing flow edge. The merge node behaves like the process modelling its outgoing flow.

Control flow A merge node $m \in R_A^M$ has a single outgoing control flow, cf:

$$cf = (\mu f : CF_A \mid f \in outgoing_{cf}(m))$$

The behaviour can be modelled thus.

$$\mathcal{F}(A,m) = \mathcal{F}(A,cf)$$

Object flow A merge node $m \in R_A^M$ has a single outgoing object flow

$$of = (\mu f : OF_A \mid f \in outgoing_{of}(m))$$

along which the object originating from one of the incoming flows is passed.

$$\mathcal{F}(A,m)[o] = \mathcal{F}(A,of)[o]$$

4.3.14 Execution environment

The activities in our formalisation execute within the context of a state machine. Each state machine has an event queue that interacts with the external environment under a run-to-completion assumption. The activities here therefore execute under that same assumption, using the event queue of the associated, owning state machine. These activities are akin to local process definitions that execute under the process M of Sect. 3.3.

The activity as a whole is modelled with a single process that contains, as appropriate, localised process descriptions corresponding to its syntactic structure. We provide two formalisations, based on whether the activity has an initial node, or whether execution starts from the activity parameter node.

Activity with initial node An activity with an initial node always starts execution from the initial node, whether an activity parameter node is present or not. Initially, the overall process behaves like the initial node $i \in R_A^I$. In the following, the process argument a corresponds to the argument of the activity parameter node; if no parameter node is present, the process argument is elided. The argument a is a global parameter available for use anywhere in the activity; diagrammatically, the use of this parameter would be indicated with an object flow connecting the parameter node and the node using a as input.

```
$A(a) =$
  let
    $\mathcal{F}(A,i) = \ldots$
    $\vdots$
    $\mathcal{F}(A,f) = Skip$
  within
    $\mathcal{F}(A,i)$
```

Activity without initial node It is possible for an activity to initially behave as an activity parameter node. In this case, we assume there is no initial node, and that a single object flow connects another node with the activity parameter node. Thus, the activity initially behaves as the activity parameter node. In this case, we assume a single activity parameter node with a single outgoing object flow. Let $p \in P_A$ be the lone activity parameter node, with argument a.

```
$A(a) =$
  let
    $\mathcal{F}(A,p)[o] = \ldots$
    $\vdots$
  within
    $\mathcal{F}(A,p)[a]$
```

The activity parameter node behaves like the outgoing object flow:

$$of = (\mu f : OF_A \mid f \in outgoing_{of}(p))$$

$$\mathcal{F}(A,p)[o] = \mathcal{F}(A,of)[o]$$

5 A CSP model for SysML blocks

In this section, we provide formalisations for the structural counterparts of our framework. The behavioural formalisms we have introduced thus far—state machines and activities—all execute within the context of a structural counterpart. The structural constructs of interest to us are enumerations, signals, blocks, parts and connectors.

5.1 Enumerations and signals

Signal and enumeration definitions introduce the messages and associated parameters communicated between state machines and activities.

Let $\mathcal{E}$ denote the set of all enumerations. An enumeration is a user-defined type where the enumeration literals represent distinct constants in the model. The CSP counterpart of a SysML enumeration is a CSP datatype. Here, there is a one-to-one mapping between the constants of the CSP datatype, and the enumeration literals defined for the SysML enumeration.

Let $\mathcal{S}$ denote the set of all signals. Signals are communicated along the connectors that connect parts. The signals used by the communicating state machines correspond to constants of a CSP datatype definition. For each block, the signals corresponding to the provided receptions of the particular block are used. Where a signal has associated parameters, these are included in the datatype definition.

5.2 Blocks, parts and connectors

The fundamental modelling construct present in SysML is the block. Each block is assigned an associated main behaviour, called its *classifier* behaviour. Depending on the purpose of the block, the classifier behaviour can either be a state machine or an activity.

Let $\mathcal{B}$ denote the set of all blocks. A block $B \in \mathcal{B}$ is a classifier that describes common behavioural and structural features of its instances, and can be considered akin to a UML class. We assume that the classifier behaviour is specified using state machines, and given by the function

$$classifier : \mathcal{B} \to \mathcal{M}$$

These blocks communicate via events (instances of signals) that act as stimuli for the respective state machines.

A block makes known the names of the *receptions*, each corresponding to a signal, that: it responds to (i.e. the block provides the behaviour); or, alternatively, expects its SysML environment to respond to (i.e. the environment provides the behaviour). These behavioural features are designated as *provided* and *required behavioural features*. We define the functions

$$prov : \mathcal{B} \to \mathbb{P}\,\mathcal{S}$$
$$reqd : \mathcal{B} \to \mathbb{P}\,\mathcal{S}$$

to return provided and required receptions.

The *internal block diagram* graphically sets out the internal structure of a block from its parts. In contrast, the *block definition diagram* depicts the composition of a block, but abstracts away from the internal structure. A *part* is connected to another part via a connector; it is an instance of a block. As such, it represents a particular usage of its classifying block within the context of its owning block. Each part $P \in \mathcal{P}$ is typed by a block $B \in \mathcal{B}$; the function

$$type : \mathcal{P} \to \mathcal{B}$$

reflects this.

The *connector* serves as a bidirectional link between the block instances and is used to convey signals between communicating block instances. The state machine of a block B only receives (through its event queue) the provided receptions, $prov(B)$. The required features, $reqd(B)$, are communicated across the connectors linking parts. SysML block instances are connected using connectors; connectors are modelled using CSP channels. For simplicity, we use the name of the association end for the purposes of communication and assume this to be the name of the associated block instance. For example, if another part P_i provides a feature S_j that a part P_k requires, the state machine of P_k will use the name of part P_i as the CSP channel on which to send the required event.

The structure, and subsequent overall behaviour, of a block $B \in \mathcal{B}$, composed from N constituent block instances, $\{P_0 \mathinner{..} P_{N-1}\} \subseteq \mathcal{P}$, is expressed in CSP via parallel composition. The classifier behaviour of each part P_j is modelled via a state machine M_j, given by

$$M_j = classifier(type(P_j))$$

The complete system, B_i, can be modelled by placing in parallel the processes corresponding to each of the state machines M_j, where $0 \leq j \leq N-1$:

$$B_i = \| \, j : \{0 \mathinner{..} N-1\} \bullet [\alpha M_j] M_j$$

5.3 Reasoning about blocks

We now discuss and formalise the different interpretations (abstraction and controller) attributed to the resulting composition when a block is composed of other blocks.

5.3.1 Abstraction

The first of the interpretations above assumes that the behaviour of the composite block serves as an abstraction of the behaviour of its parts. Assume a block $B_i \in \mathcal{B}$ composed of K constituent blocks $B_0 \mathinner{..} B_{K-1}$, where $i \geq K$. We know that the aggregate behaviour exhibited by blocks $B_0 \mathinner{..} B_{K-1}$ must adhere to that of the composite block B_i. B_i is an abstract specification block that the more concrete implementation blocks $B_0 \mathinner{..} B_{K-1}$ must implement. Stated in terms of CSP: the characteristic process of B_i serves as the specification process and $B_0 \mathinner{..} B_{K-1}$, suitably combined using parallel composition, form the implementation process.

Assume that $\mathit{classifier}(B)$ represents the classifier behaviour of a SysML block. Using CSP the conformance of the implementation process to that of the specification can be stated thus.

$$\begin{aligned}&\mathit{CONCRETE} = \| \, P : \{B_0 \mathinner{..} B_{K-1}\} \bullet \mathit{classifier}(P)\\&\mathit{classifier}(B_i) \sqsubseteq \mathit{CONCRETE}\end{aligned}$$

Events introduced at the lower level of implementation are excluded from the above observation; the hiding operator of CSP can be used to conceal such events.

Using this approach, and assuming the refinement holds, B_i can be safely substituted for the concrete composition $B_0 \mathinner{..} B_{K-1}$. This stepwise, compositional approach to systems specification and design sits well with CSP's approach to refinement. This statement is not necessarily true for conventional model checkers that rely on temporal logics to assert safety or liveness properties. In a *system of systems*, B_i, previously our *system of interest*, is now just a component block representing one of the subsystems.

5.3.2 Controller

The second interpretation assumes that a block acts as a controller for its parts. The behaviour of the block is a combination of its behaviour and that of its parts: the behaviour of its parts is considered as part of the behaviour of the composite.

Assume a block $B_i \in \mathcal{B}$, which is composed of K constituent blocks $B_0 \mathinner{..} B_{K-1}$, where $i \geq K$. We known that the aggregate behaviour is that of blocks $B_0 \mathinner{..} B_{K-1}$ as well as the composite block B_i. Here, we think of all of the blocks as implementation blocks that combine in order to give the desired behaviour. This can be formalised in terms of CSP thus.

$$\begin{aligned}&\mathit{CONCRETE} =\\&\quad \| \, P : \mathit{Union}(\{\{B_0 \mathinner{..} B_{K-1}\}, \{B_i\}\}) \bullet\\&\qquad \mathit{classifier}(P)\end{aligned}$$

6 Consistency checking: an illustrative case study

6.1 A case study

We now give consideration to a case study, which we will use to illustrate our contribution. The case study, reprised from [13], is used as a means to evaluate and illuminate the contribution of this paper.

We study a single component—a robotic arm—of a fully fledged case study that is well known within the formal methods community. The *production cell* is an industrial installation of a metal processing plant located in Karlsruhe, Germany [14]. However, in the interest of brevity and clarity, we consider the Arm as our system of interest. The Arm is one subsystem of the travelling crane, which is yet another component of the much bigger system—the production cell itself.

Actuators and *sensors* are individual components that communicate with the system controller: actuators receive outputs from the controller in order to coordinate the operation of several components; sensors, as the name suggests, are sensory components that send inputs to the system controller. Examples of actuators include *bidirectional motors*, which can operate in two opposing directions, and *electromagnets*, which can activate or deactivate a magnetic field using an electric current. A *potentiometer* is an example of a sensor: it provides a value within certain limits so as to indicate the range of extension.

The Arm is equipped with a bidirectional motor responsible for vertical extension. An electromagnet is placed at the front of the Arm for handling metal objects; a potentiometer is present to indicate the range of extension of the Arm.

The case study is explored from two different perspectives, related to the different interpretations that can be attributed to the composition of the composing block.

6.2 The Arm

The Arm is the block of interest for the purposes of the case study. In this subsection, we explore the behaviours of the blocks that make up the composition: BDMotor, PDMeter and Magnet. These blocks or parts have exactly the same behaviour, regardless of the interpretation assigned to their composition. We introduce the SysML constructs common

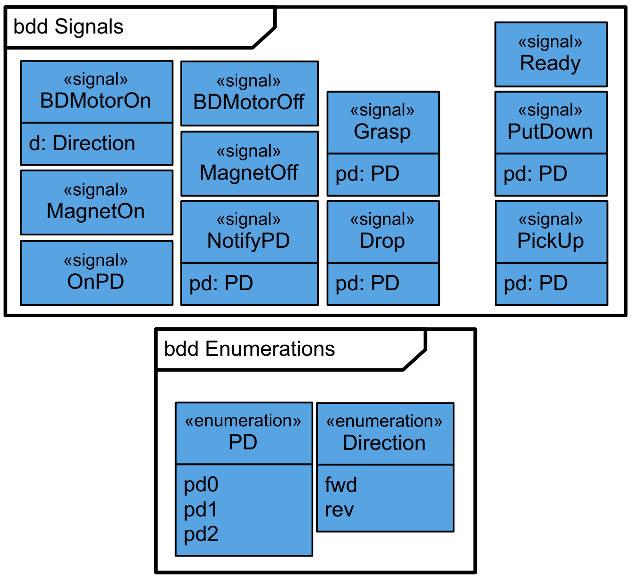

Fig. 3 The block definition diagram introducing signals and enumerations

to both interpretations and their relationships to their CSP counterparts in this section. We start by looking at the structural aspects, followed by behavioural constructs, such as state machines and activities.

6.2.1 Enumerations and signals

Refer to Fig. 3. Signal and enumeration definitions introduce the messages and associated parameters communicated between state machines and activities. We introduce all the signals and enumerations utilised in both interpretations here.

The Direction and PD enumerations of Fig. 3 can be represented with CSP datatypes:

$$\text{datatype } Direction = fwd \mid rev$$
$$\text{datatype } PD = pd_0 \mid pd_1 \mid pd_2$$

In the above, the potential differences are denoted using different constants, with each corresponding to a reading returned by the potentiometer.

The signals used by the communicating state machines are defined similarly. For each block, the signals that correspond to the provided receptions of the particular block are used. Where a signal has associated parameters, these are included in the datatype definition.

$$\text{datatype } BDMotorSignal = BDMotorOn.Direction \mid BDMotorOff$$
$$\text{datatype } MagnetSignal = MagnetOff \mid MagnetOn$$
$$\text{datatype } PDMeterSignal = NotifyPD.PD$$

SysML blocks are connected using connectors, which are modelled using CSP channels. For the sake of simplicity, we use the name of the association end for the purposes of communication and assume this to be the name of the associated block. Thus, for every block, we require two CSP channels: the first models the event queue that the block uses to communicate with the external environment; the second is used for internal communication between the block and its associated event queue.

$$\text{channel } bdmotor : BDMotorSignal$$
$$\text{channel } bdmotorlocal : Dispatched.BDMotorSignal$$
$$\text{channel } magnet : MagnetSignal$$
$$\text{channel } magnetlocal : Dispatched.MagnetSignal$$
$$\text{channel } pdmeter : PDMeterSignal$$
$$\text{channel } pdmeterlocal : Dispatched.PDMeterSignal$$

For example, the channel *bdmotor* is used to communicate with the state machine of the bidirectional motor via its associated event queue and the channel *bdmotorlocal* is used by the event queue of the bidirectional motor to dispatch events for processing. The datatype *Dispatched* models this: an event can either be processed, or, if the state machine is in a state where the dispatched event is not expected, discarded. Using the above assumption, any block connected to *BDMotor* via a connector uses the channel *bdmotor* to send signal events destined for *BDMotor*.

$$\text{datatype } Dispatched = proc \mid disc$$

6.2.2 State machines

The *classifier behaviour* is the main behaviour of a block and executes as soon as the instance is created until the point of destruction. The modelling construct that is most frequently used to represent the classifier behaviour is a state machine. In most systems engineering methodologies, activities are typically used as a complementary modelling notation to state machines: it is the behavioural formalism normally associated with the effect component of a transition; alternatively, it is used to model behaviours related to a particular state.

Figure 4 shows the state machines of the magnet, the bidirectional motor and the potentiometer. Activities that execute on the transitions as the effect component are italicised after the trigger, which is set in bold typeface. Activities are shown in Fig. 5 and are discussed in Sect. 6.2.3. For now, it is sufficient to assume that these are modelled using CSP processes.

The CSP processes modelling the state machines for the BDMotor, the Magnet and the PDMeter blocks follow. Local process definitions model each state in the associated state

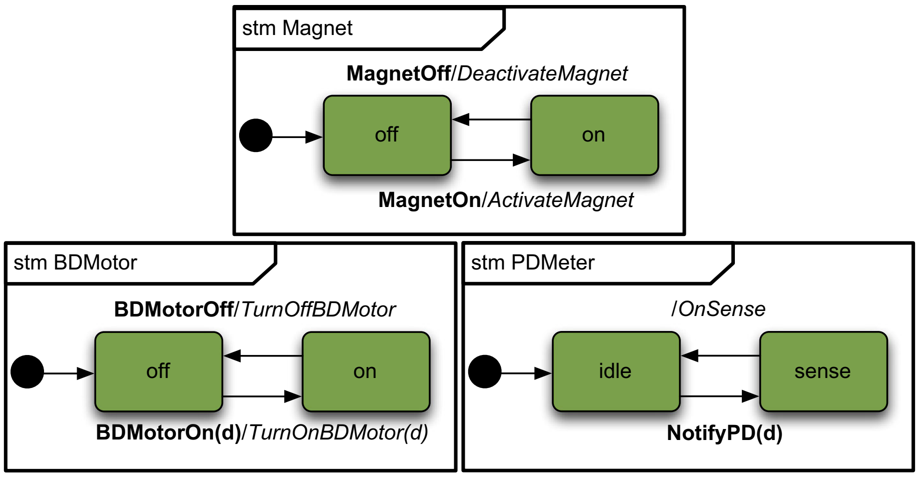

Fig. 4 The state machine diagrams of the Magnet, BDMotor and PDMeter blocks

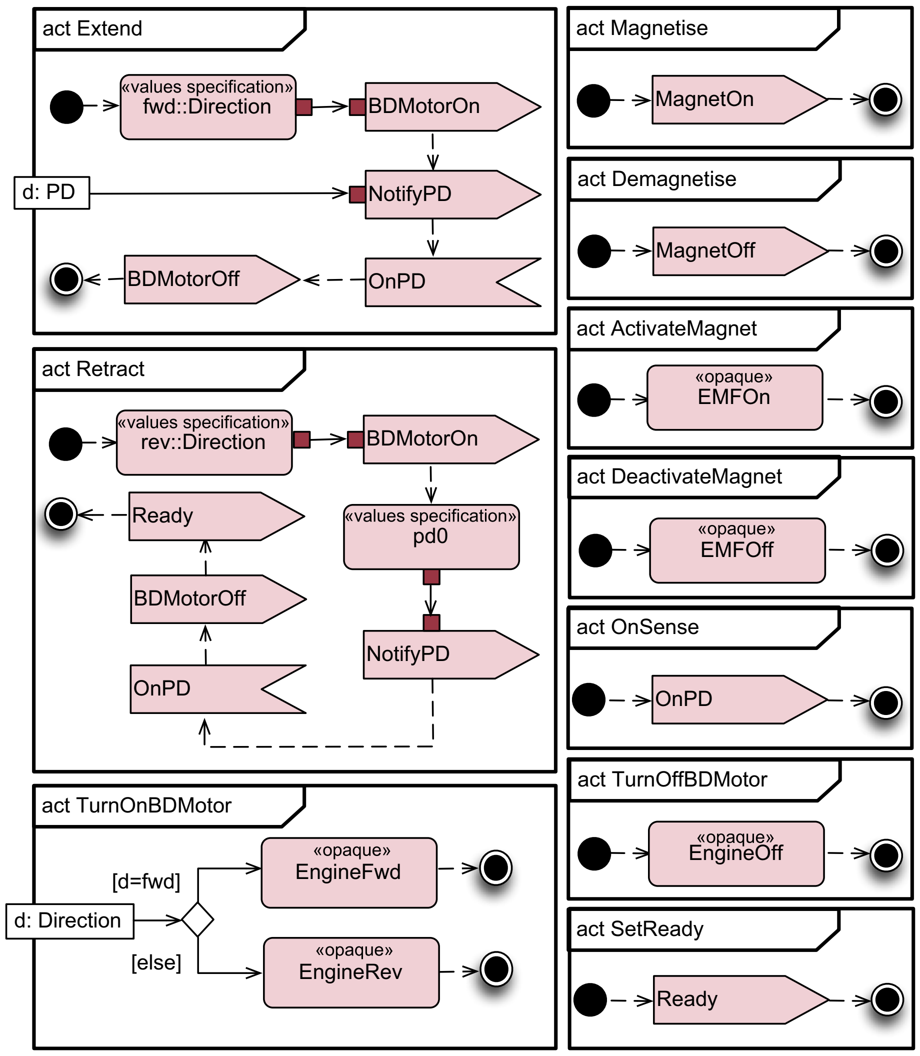

Fig. 5 Activity diagrams modelling additional behaviours executed within the context of state machines

machine. The deterministic choice between permissible triggers is offered to the external environment. Recall that if a CSP event corresponding to a permitted SysML triggering event is received:

– the process modelling the exit behaviour of the source state executes;
– the process modelling the effect of the transition executes;

– the process modelling the entry behaviour of the target state executes; and
– the target state is entered.

The above is modelled using the sequential composition operator. The aforementioned behaviours are all SysML activities with corresponding CSP processes; if a behaviour is not present, it is simply not included in the sequential composition.[10] Note that, in every state, the dispatched, unexpected events are discarded and thus removed from the event queue without effect: this corresponds to communications of the form $local.disc.e$, where e is an signal event. Events that are served up for processing and successfully processed by the state machine correspond to communications of the form $local.proc.e$.

Alphabets of the individual processes are defined below each process definition. The alphabet of a state machine is the set of events that it can communicate, as well as the alphabets of its associated activities.

$$
\begin{array}{l}
BDMotor(queue, local) = \\
\quad \text{let} \\
\qquad I_0 = OFF \\
\qquad OFF = \\
\qquad\quad local.proc.BDMotorOn?d \to \\
\qquad\qquad TurnOnBDMotor(d) \mathbin{;} \\
\qquad\qquad ON \\
\qquad\quad \Box \\
\qquad\quad local.disc?e : \{|\ BDMotorOff\ |\} \to \\
\qquad\qquad OFF \\
\qquad ON = \\
\qquad\quad local.proc.BDMotorOff?d \to \\
\qquad\qquad TurnOffBDMotor(d) \mathbin{;} \\
\qquad\qquad OFF \\
\qquad\quad \Box \\
\qquad\quad local.disc?e : \{|\ BDMotorOn\ |\} \to \\
\qquad\qquad ON \\
\qquad EQ = queue?e \to local?p!e \to EQ \\
\quad \text{within} \\
\qquad I_0 \,[|\, \{|\ local\ |\} \,|]\, EQ \\
BDMOTOR = BDMotor(bdmotor, bdmotorlocal) \\
\\
\alpha BDMOTOR = \\
\quad Union(\{\ \{|\ bdmotor, bdmotorlocal\ |\}, \\
\qquad\qquad \alpha\, TurnOnBDMotor, \\
\qquad\qquad \alpha\, TurnOffBDMotor\})
\end{array}
$$

$$
\begin{array}{l}
Magnet(queue, local) = \\
\quad \text{let} \\
\qquad I_0 = OFF \\
\qquad OFF = \\
\qquad\quad local.proc.MagnetOn \to \\
\qquad\qquad ActivateMagnet \mathbin{;} \\
\qquad\qquad ON \\
\qquad\quad \Box \\
\qquad\quad local.disc?e : \{|\ MagnetOff\ |\} \to \\
\qquad\qquad OFF \\
\qquad ON = \\
\qquad\quad local.proc.MagnetOff \to \\
\qquad\qquad DeactivateMagnet \mathbin{;} \\
\qquad\qquad OFF \\
\qquad\quad \Box \\
\qquad\quad local.disc?e : \{|\ MagnetOn\ |\} \to \\
\qquad\qquad ON \\
\qquad EQ = queue?e \to local?p!e \to EQ \\
\quad \text{within} \\
\qquad I_0 \,[|\, \{|\ local\ |\} \,|]\, EQ \\
MAGNET = Magnet(magnet, magnetlocal) \\
\\
\alpha MAGNET = \\
\quad Union(\{\ \{|\ magnet, magnetlocal\ |\}, \\
\qquad\qquad \alpha ActivateMagnet, \\
\qquad\qquad \alpha DeactivateMagnet\})
\end{array}
$$

$$
\begin{array}{l}
PDMeter(queue, local) = \\
\quad \text{let} \\
\qquad I_0 = IDLE \\
\qquad IDLE = \\
\qquad\quad local.proc.NotifyPD?pd \to SENSE \\
\qquad SENSE = \\
\qquad\quad OnSense \mathbin{;} IDLE \\
\qquad\quad \Box \\
\qquad\quad local.disc?e : \{|\ NotifyPD\ |\} \to SENSE \\
\qquad EQ = queue?e \to local?p!e \to EQ \\
\quad \text{within} \\
\qquad I_0 \,[|\, \{|\ local\ |\} \,|]\, EQ
\end{array}
$$

[10] Alternatively, it can be modelled using the CSP process, *Skip*.

$$PDMETER = PDMeter(pdmeter, pdmeterlocal)$$

$$\alpha PDMETER = Union(\{\{| pdmeter, pdmeterlocal |\}, \alpha OnSense\})$$

The channel *pdmeter* is used for communication with the state machine of the potentiometer, and *pdmeterlocal* is used for internal communication between the event queue and state machine.

6.2.3 Activities

The activities that serve to augment the classifier behaviour of the blocks introduced in Sect. 6.2.2 are formalised in the following.

Each activity has an associated CSP process with localised process definitions corresponding to the actions. Activity parameter nodes are modelled with local process variables. Opaque actions are communicated on the CSP channel, *opaque*.

The activities of the bidirectional motor—TurnOnBD Motor and TurnOffBDMotor—can be modelled thus.

```
$TurnOnBDMotor(d) =$
  let
    $MERGE_0 =$
      if $d == fwd$ then
        $OA_0$
      else
        $OA_1$
    $OA_0 = opaque.enginefwd \rightarrow F_0$
    $OA_1 = opaque.enginerev \rightarrow F_0$
    $F_0 = Skip$
  within
    $MERGE_0$
```

$$\alpha\, TurnOnBDMotor = \{| opaque.enginefwd, opaque.enginerev |\}$$

```
$TurnOffBDMotor =$
  let
    $I_0 = OA_0$
    $OA_0 = opaque.engineoff \rightarrow F_0$
    $F_0 = Skip$
  within
    $I_0$
```

$$\alpha\, TurnOffBDMotor = \{| opaque.engineoff |\}$$

The processes corresponding to the remainder of the activities of Fig. 5—ActivateMagnet, DeactivateMagnet and OnSense—are defined similarly. We omit the definitions here in the interest of brevity.

6.3 Interpretations

We now give consideration to the different notions that can be attributed to the behavioural composition of a collection of blocks using a process-algebraic approach.

6.3.1 Abstraction

We now explore the behavioural composition of the Arm block with the first interpretation of Sect. 1—“The classifier behaviour of the block can serve as an abstraction of the behaviours of its parts”—in mind.

The block definition diagram showing the composition of the Arm is given Fig. 6; the interconnections among the parts are depicted with the internal block definition diagram of Fig. 7. The structural aspects of the system are modelled using blocks for the controller, the bidirectional

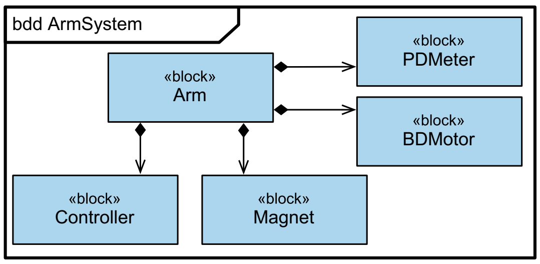

Fig. 6 The block definition diagram of the Arm system

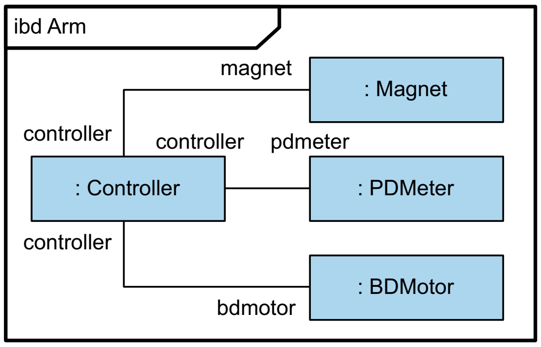

Fig. 7 The internal block definition diagram of the Arm block

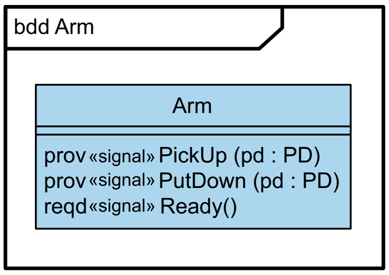
bdd Arm
Arm
prov «signal» PickUp (pd : PD)
prov «signal» PutDown (pd : PD)
reqd «signal» Ready()

Fig. 8 The block definition diagram of the Arm block

motor, the electromagnet and the potentiometer. The classifier behaviour of the Arm is to serve as an abstraction of the behaviours of its parts: the BDMotor, Magnet, PDMeter and Controller. We have seen CSP definitions modelling the behaviours of the BDMotor, Magnet and PDMeter; the controller is introduced below.

The channel and datatype definitions are similar to those defined earlier.

datatype $ArmSignal = PickUp.PD \mid PutDown.PD$
datatype $ControllerSignal =$
$\quad Grasp.PD \mid Drop.PD \mid OnPD$

channel $controller : ControllerSignal$
channel $controllerlocal :$
$\quad Dispatched.ControllerSignal$

The provided and required receptions of the Controller and Arm blocks are shown below.[11] There is a clear correspondence between the CSP datatype definitions and the provided receptions of the SysML blocks. Required receptions should appear in the CSP datatype definitions of other blocks in the system that receive these signal events as triggers in their classifying state machines (Fig. 8, 9).

The CSP process describing the characteristic behaviour of the controller's state machine follows. The activity

[11] The detailed block definition diagrams of other blocks are omitted in the interest of brevity.

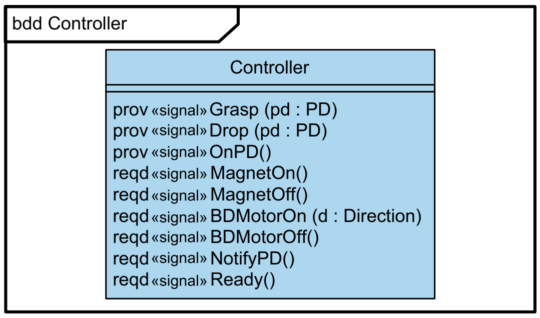
bdd Controller
Controller
prov «signal» Grasp (pd : PD)
prov «signal» Drop (pd : PD)
prov «signal» OnPD()
reqd «signal» MagnetOn()
reqd «signal» MagnetOff()
reqd «signal» BDMotorOn (d : Direction)
reqd «signal» BDMotorOff()
reqd «signal» NotifyPD()
reqd «signal» Ready()

Fig. 9 The block definition diagram of the Controller block

Extend is associated with the effect component of the transitions emanating from the idle state; the activity Magnetise represents the entry behaviour of the grasp state. Activities are shown in Fig. 4.

$Controller(queue, local) =$

let

$\quad I_0 = IDLE$

$\quad IDLE =$
$\quad\quad local.proc.Grasp?e \rightarrow$
$\quad\quad\quad Extend(local, e) \mathbin{⨟}$
$\quad\quad\quad Magnetise \mathbin{⨟}$
$\quad\quad\quad GRASP$
$\quad\quad \square$
$\quad\quad local.proc.Drop?e \rightarrow$
$\quad\quad\quad Extend(local, e) \mathbin{⨟}$
$\quad\quad\quad Demagnetise \mathbin{⨟}$
$\quad\quad\quad DROP$
$\quad\quad \square$
$\quad\quad local.disc?e : \{| \, OnPD \, |\} \rightarrow$
$\quad\quad\quad IDLE$
$\quad GRASP =$
$\quad\quad Retract(local) \mathbin{⨟} IDLE$
$\quad\quad \square$
$\quad\quad local.disc?e : \{| \, Grasp, Drop, OnPD \, |\} \rightarrow$
$\quad\quad\quad GRASP$

$\quad DROP =$
$\quad\quad Retract(local) \mathbin{⨟} IDLE$
$\quad\quad \square$
$\quad\quad local.disc?e : \{| \, Grasp, Drop, OnPD \, |\} \rightarrow$
$\quad\quad\quad DROP$

$\quad EQ = queue?e \rightarrow local?p!e \rightarrow EQ$

within

$\quad I_0 \, [| \, \{| \, local \, |\} \, |] \, EQ$

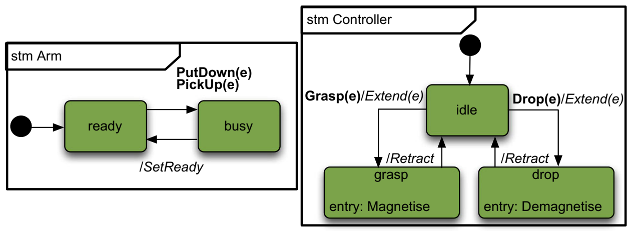

Fig. 10 The state machine diagrams of the classifier behaviours of the Arm and Controller blocks

$$CONTROLLER = \\ \quad Controller(controller, controllerlocal)$$

$$\alpha CONTROLLER = \\ \quad Union(\{\{| controller, controllerlocal |\}, \\ \qquad \alpha Magnetise, \\ \qquad \alpha Demagnetise, \\ \qquad \alpha Extend, \\ \qquad \alpha Retract\})$$

The blocks described above—Controller, BDMotor, Magnet and PDMeter—are all concrete implementation blocks in SysML (Fig. 10). The abstract block, Arm, which serves as an implementation that the parts must realise, is modelled below.

$Arm(queue, local) =$

let

$I_0 = READY$

$READY =$
$\quad local.proc.PickUp?e \rightarrow BUSY$
$\quad \square$
$\quad local.proc.PutDown?e \rightarrow BUSY$

$BUSY =$
$\quad SetReady \fatsemi READY$
$\quad \square$
$\quad local.disc?e : \{| PickUp, PutDown |\} \rightarrow$
$\qquad BUSY$

$EQ = queue?e \rightarrow local?p!e \rightarrow EQ$

within

$I_0 [| \{| local |\} |] EQ$

$ARM = Arm(arm, armlocal)$

$$\alpha ARM = \\ \quad Union(\{\{| arm, armlocal |\}, \\ \qquad \alpha SetReady\})$$

The process $SetReady$ used within the ARM process follows.

$SetReady =$

let

$I_0 = SS_0$

$SS_0 = client.Ready \rightarrow F_0$

$F_0 = Skip$

within

I_0

$\alpha SetReady = \{| client.Ready |\}$

The processes responsible for modelling the activities used in the $CONTROLLER$ process follow.

All activities in this paper execute within the context of their owning state machine. An activity can take parameters, passed from the arguments of the triggering event of the owning state machine, as input. Some activities have receive signal events as actions; these receive signal events need to be passed via the event queue mechanism of the state machine. It follows that the channel used for local communication with the state machine ought to be passed in as an argument to the activity.

$Extend(local, pd) =$

let

$I_0 = VS_0$

$VS_0 = SS_0(fwd)$

$SS_0(o) = bdmotor.BDMotorOn.o \rightarrow SS_1$

$SS_1 = pdmeter.NotifyPD.pd \rightarrow RS_0$

$RS_0 =$
$\quad local.proc.OnPD \rightarrow SS_2$
$\quad \square$
$\quad local.disc?ev : \{| Grasp, Drop |\} \rightarrow RS_0$

$SS_2 = bdmotor.BDMotorOff \rightarrow F_0$

$F_0 = Skip$

within

I_0

$$\begin{array}{l}\alpha Extend = \\ \quad \{\mid bdmotor.BDMotorOn.fwd, \\ \quad\quad bdmotor.BDMotorOff, \\ \quad\quad pdmeter.NotifyPD \mid\}\end{array}$$

$$\begin{array}{l}Retract(local) = \\ \quad \text{let} \\ \quad\quad I_0 = VS_0 \\ \quad\quad VS_0 = SS_0(rev) \\ \quad\quad SS_0(o) = bdmotor.BDMotorOn.o \rightarrow VS_1 \\ \quad\quad VS_1 = SS_1(pd_0) \\ \quad\quad SS_1(o) = pdmeter.NotifyPD.0 \rightarrow RS_0 \\ \quad\quad RS_0 = \\ \quad\quad\quad local.proc.OnPD \rightarrow SS_2 \\ \quad\quad\quad \Box \\ \quad\quad\quad local.disc?ev : \{\mid Grasp, Drop \mid\} \rightarrow RS_0 \\ \quad\quad SS_2 = bdmotor.BDMotorOff \rightarrow SS_3 \\ \quad\quad SS_3 = client.Ready \rightarrow F_0 \\ \quad\quad F_0 = Skip \\ \quad \text{within} \\ \quad\quad I_0\end{array}$$

$$\begin{array}{l}\alpha Retract = \\ \quad \{\mid bdmotor.BDMotorOn.rev, \\ \quad\quad bdmotor.BDMotorOff, \\ \quad\quad pdmeter.NotifyPD.pd_0, \\ \quad\quad client.Ready \mid\}\end{array}$$

$$\begin{array}{l}Magnetise = \\ \quad \text{let} \\ \quad\quad I_0 = SS_0 \\ \quad\quad SS_0 = magnet.magnetOn \rightarrow F_0 \\ \quad\quad F_0 = Skip \\ \quad \text{within} \\ \quad\quad I_0\end{array}$$

$$\alpha Magnetise = \{\mid magnet.MagnetOn \mid\}$$

$$\begin{array}{l}Demagnetise = \\ \quad \text{let} \\ \quad\quad I_0 = SS_0 \\ \quad\quad SS_0 = magnet.magnetOff \rightarrow F_0 \\ \quad\quad F_0 = Skip \\ \quad \text{within} \\ \quad\quad I_0\end{array}$$

$$\alpha Demagnetise = \{\mid magnet.MagnetOff \mid\}$$

Send signal event actions may have input object pins that determine the argument sent as part of the event; similarly, receive signal event actions may receive arguments and therefore have object output pins. Value specification actions[12] are used in object flows to output a constant value that serve as input to another action. In every case, the internal channel is used to receive events using the event passing mechanism. Signal events[13] are sent using the channel with the same name as the target block, similar to the approach taken for state machines.[14]

Assuming that

$$\begin{array}{l}P = \{CONTROLLER, MAGNET, \\ \quad\quad BDMOTOR, PDMETER\}\end{array}$$

we then have

$$CONCRETE = \parallel p : P \bullet [\alpha p]p$$

The set of processes, P, represent the concrete implementation blocks whose conjoined behaviour must be that of the block arm that serves as its specification. The similarity with CSP here is striking: refinement in CSP is expressed between specification and implementation processes.

12 As an example, the action outputting the forward direction in Extend.

13 As examples, see BDMotorOn in Extend for a send signal event and OnPD in Retract for a receive signal event.

14 In addition, send and receive signal events have input and output pins that can identify the target and source of an action.

$CONCRETE^R$ is the process with events suitably renamed to ensure compatible alphabets.

$$\begin{aligned}
&CONCRETE^R = \\
&\quad CONCRETE\,[\\
&\qquad controller.Grasp.pd_0 \leftarrow arm.PickUp.pd_0, \\
&\qquad controller.Drop.pd_0 \leftarrow arm.PutDown.pd_0, \\
&\qquad controller.Grasp.pd_1 \leftarrow arm.PickUp.pd_1, \\
&\qquad \vdots \\
&\qquad]
\end{aligned}$$

The set $Hidden$ contains those events not present in the alphabet of the concrete specification process, ARM; Σ denotes the set of all CSP events within the context of the specification. Thus

$$\begin{aligned}
&Hidden = \\
&\quad \Sigma \setminus \{|\ arm.PickUp, \\
&\qquad arm.PutDown, \\
&\qquad armlocal.proc.PickUp, \\
&\qquad armlocal.proc.PutDown, \\
&\qquad armlocal.disc.PickUp, \\
&\qquad armlocal.disc.PutDown, \\
&\qquad client\ |\}
\end{aligned}$$

FDR3 verifies the assertion

$$ARM \sqsubseteq_T CONCRETE^R \setminus Hidden \qquad [\sqsubseteq_T\ holds]$$

As the traces-refinement holds, ARM can be substituted for its parts in the complete system: the behaviour of the concrete implementation processes (which is captured by $CONCRETE^R$) can neither accept nor refuse an event unless ARM can. Stated another way, the characteristic behaviour of $CONCRETE^R$ is completely contained within that of ARM. The compositional approach presented above is effective in alleviating the state space explosion problem: subsystems can be developed and formally verified in isolation and subsequently combined to form an integrated system description.

6.3.2 Controller

We explore the behavioural composition of the Arm block with the second interpretation of Sect. 1—“the classifier behaviour of the block acts as a controller in order to actively orchestrate the behaviours of its parts”—in mind. The second interpretation calls for the classifier behaviour of the Arm block to act as a controller in order to actively orchestrate the behaviours of its parts. Thus, the behaviour of the Arm block must be a combination of its own behaviour and that of its parts (Fig 11).

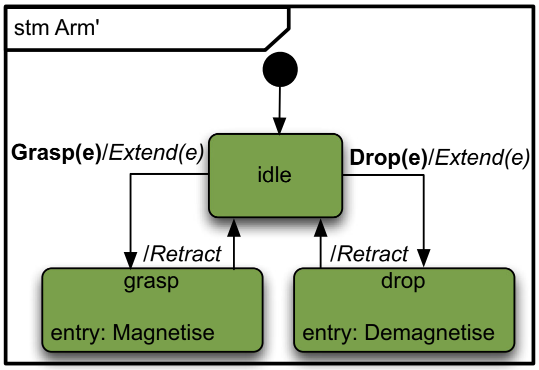

Fig. 11 The state machine diagram of the classifier behaviour of the Arm’ block

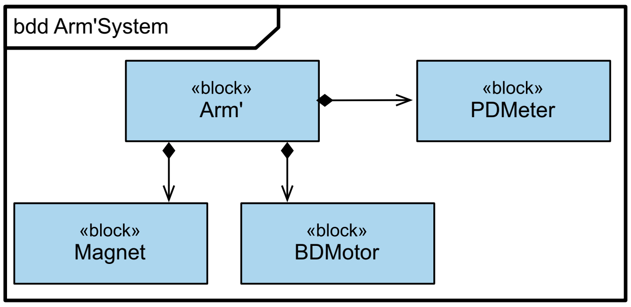

Fig. 12 The block definition diagram of the Arm’ system

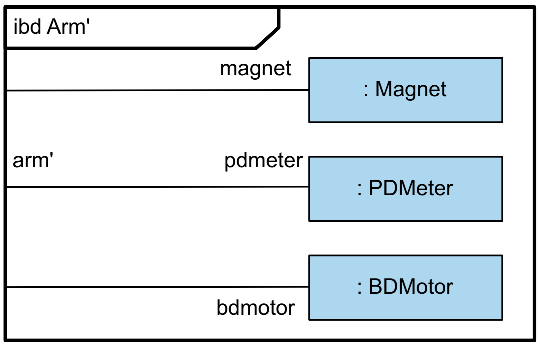

Fig. 13 The internal block definition diagram of the Arm’ block

Figure 12 shows the new composition of the Arm using the second interpretation. The Arm is still composed from instances of the potentiometer, bidirectional motor and magnet blocks. The controller block, however, is missing. The interconnection between the parts is depicted in Fig. 13: the parts now directly communicate with the composite block. The resulting composition thus behaves as a combination of its own behaviour (the classifying behaviour of the Arm) and that of its parts (the behaviours of the magnet, the potentiometer and the bidirectional motor).

The behaviour of the state machine for the Arm in the second interpretation is exactly the behaviour of the controller in the first interpretation. In the first interpretation, the controller orchestrated the behaviour of the rest of the parts, and the Arm block served as the specification. Here, the Arm block itself orchestrates the behaviours of its parts.

The behaviour of the state machine for the Arm using the second interpretation follows.

```
datatype $Arm'Signal =$
    $Grasp.PD \mid Drop.PD \mid OnPD$

channel $arm' : Arm'Signal$
channel $arm'local : Dispatched.Arm'Signal$

$Arm'(queue, local) =$

  let

    $I_0 = IDLE$

    $IDLE =$
      $local.proc.Grasp?e \rightarrow$
        $Extend(local, e) \fatsemi$
        $Magnetise \fatsemi$
        $GRASP$
      $\Box$
      $local.proc.Drop?e \rightarrow$
        $Extend(local, e) \fatsemi$
        $Demagnetise \fatsemi$
        $DROP$
      $\Box$
      $local.disc?e : \{| \ OnPD \ |\} \rightarrow$
        $IDLE$

    $GRASP =$
      $Retract(local) \fatsemi IDLE$
      $\Box$
      $local.disc?e : \{| \ Grasp, Drop, OnPD \ |\} \rightarrow$
        $GRASP$

    $DROP =$
      $Retract(local) \fatsemi IDLE$
      $\Box$
      $local.disc?e : \{| \ Grasp, Drop, OnPD \ |\} \rightarrow$
        $DROP$

    $EQ = queue?e \rightarrow local?p!e \rightarrow EQ$

  within

    $I_0 \ [| \ \{| \ local \ |\} \ |] \ EQ$

$ARM' = Arm'(arm', arm'local)$

$\alpha ARM' =$
    $Union(\{\{| \ arm', arm'local \ |\},$
          $\alpha Magnetise,$
          $\alpha Demagnetise,$
          $\alpha Extend,$
          $\alpha Retract\})$
```

The complete behaviour of the Arm and all its parts can be expressed in CSP as follows. Assuming that

$$P = \{ARM, MAGNET, BDMOTOR, PDMETER\}$$

we then have

$$CONCRETE' = \| \ p : P \bullet [\alpha p] p$$

The set of processes, P, represent the Arm that acts as a controller and its constituent parts: the magnet, the bidirectional motor and the potentiometer. The behaviour is a combination of the Arm and that of the magnet, the bidirectional motor and the potentiometer.

The second interpretation above has the drawback that there is no specification process that can be substituted for the composition. However, this interpretation sits well where the overall system architecture is described in terms of high-level blocks. These high-level blocks might be specification level or abstract blocks, each obtained from previous refinements using the first interpretation. At the architectural level, however, the integrated behaviour of all the components would be of interest to the modeller. Here, techniques that would assist in assured requirements traceability would be beneficial [10].

Figure 14 depicts these concepts. For example, the designers of a travelling crane might use the first interpretation above that results in an abstract block that denotes the robotic arm. This block might then be substituted in place of its components[15] in the system of interest—the travelling crane—along with other blocks, such as sensors and bidirectional motors, using the second interpretation. Alternatively, the first interpretation might be used again to obtain a single abstract block, modelling the travelling crane, when the system of interest is the production cell. At this level, one might have refinements modelling behavioural safety requirements, as outlined in, for example [10].

7 Conclusions

We have shown how the refinement checker FDR3 can be used in a practical setting to ensure that different behavioural formalisms—activities and state machines—are consistent. Moreover, we have demonstrated how refinement can be used to decompose a complex design problem to give rise to a top-down approach to designing a system comprised of subsystems. In doing this, we have defined the semantics of state machines and activities that execute within this context. This paper represents a significant extension to the

[15] Presuming that the refinement holds.

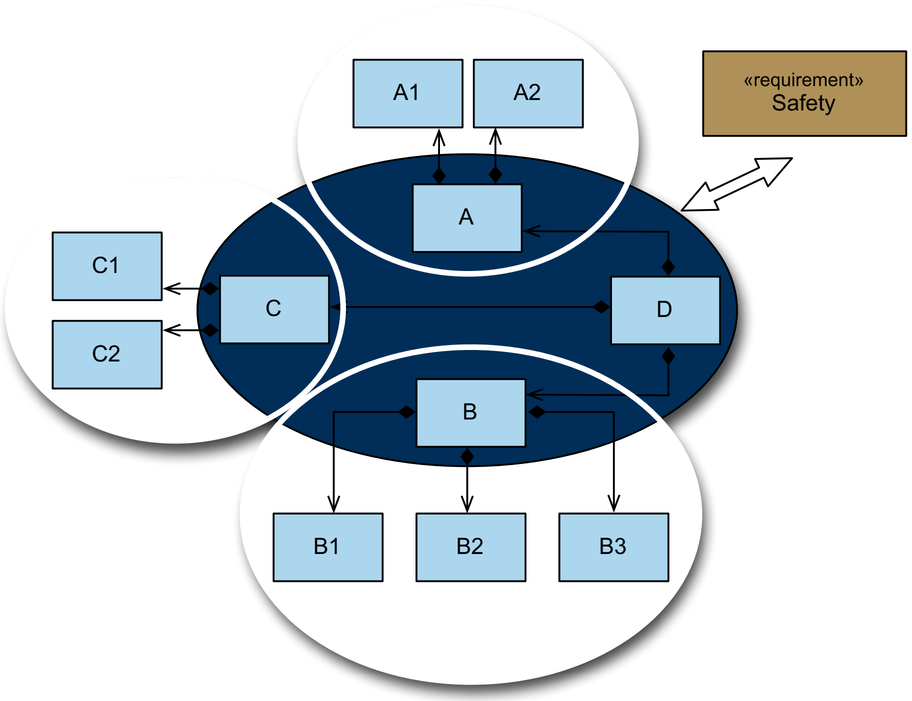

Fig. 14 The compositional approach afforded by CSP. The *white ellipses* denote the behavioural interpretation of blocks using the abstraction approach. The system of interest, block *D*, is composed of abstract blocks and serves as a controller that orchestrates the behaviour. The second interpretation, shown inside the *dark ellipse*, applies here. The behavioural of the overall system is the combined behaviour of blocks *A, B, C* and *D*. Furthermore, block *A* serves as a behavioural specification that must be satisfied by its constituted blocks *A1* and *A2*. Block *A* can be substituted for its components in the overall system. A similar line of argument can be followed for blocks *B* and *C*, together with their component blocks. Safety requirements can be allocated behavioural constructs to further refine the intentions of the modeller and checked for conformance using CSP [10]

contribution of [12]. The interested reader is referred to [10] for a consideration of how this approach can support formal requirements traceability via refinement checking.

To the best of our knowledge, this is the first contribution that has explored the different notions of behavioural integration from a formal, process-algebraic perspective. Furthermore, we are not aware of any other contribution that integrates the combined behaviour of the formalisms explored in this paper—state machines and activities—using CSP.

Formal semantics for some of the SysML diagrams has been given in terms of the *COMPASS Modelling Language* (CML) [21]. A set of translation rules are given that maps SysML diagrams to their counterparts in CML. The work described in this paper differs in that CML integrates state-based, as well as process-algebraic description techniques. Our work, on the other hand, is concerned solely with defining a process-algebraic approach to ensure behavioural conformance among the behaviour diagrams of SysML.

Ng and Butler [15] proposed the formalisation of UML state machine diagrams, with CSP being used as the semantic domain [15]. They define the translation in terms of a mapping from structural diagrammatic constructs to their CSP counterparts. The translation starts from an initial state and then proceeds to deduce the behaviour of the entire state machine in terms of CSP descriptions. Broadly speaking, each state is mapped to a process and each UML event is mapped to a CSP event. The work of Yeung et al. [25] builds on that of Ng and Butler by generalising inter-level transitions and prioritising transitions at different levels of the state hierarchy. The authors therefore provide an alternative semantics for state machines. However, their approach only takes into account those constructs formalised in [15].

Zang and Liu [26] mapped state machine diagrams to CSP using the model checker PAT [20]. A state machine is modelled by a single CSP process; translation rules are presented that map state machine constructs to CSP# [20], which is the input language of the *Process Analysis Toolkit* (PAT) [20]. Refinement checking, as well as model checking, is possible: both are natively supported by the model checker. The transformation methodology is presented via a set of rules.

Xu et al. [22,23] formalised activity diagrams in CSP. A transformation function is defined that maps the mathematical representation of an activity to the semantic domain of

Table 1 Comparison of CSP formalisations of state machines

State machine construct	This paper	Ng and Butler [15]	Bolton, Crichton and Davies [4,6]	Yeung et al. [25]
Initial state	✓	✓	✓	✓
Final state	✓	✓	✓	✓
Terminate state	✓			
Simple state	✓	✓	✓	✓
Junction state	✓			
Choice state	✓	✓		
Simple composite state	✓	✓		✓
Orthogonal composite state		✓		
Triggers	✓	✓	✓	✓
Guards	✓	✓		
Effects	✓	✓	✓	
Entry and exit behaviours	✓	✓		✓
Do behaviours		✓		
Event queue	✓		✓	
Multiple state machines	✓		✓	
Integration with activities	✓			
Global variables		✓		

Table 2 Comparison of CSP formalisations of activities

Activity construct	This paper	Xu et al. [22,23]	Bolton, Crichton and Davies [4,6]	Abdelhalim et al. [1,2]
Initial node	✓	✓	✓	✓
Final node	✓	✓	✓	✓
Fork node	✓	✓	✓	
Join node	✓	✓	✓	
Decision node	✓	✓	✓	✓
Merge node	✓	✓	✓	✓
Signal event nodes	✓			✓
Value specification nodes	✓			✓
(Opaque) actions	✓	✓	✓	✓
Call behaviour actions	✓			
Parameter nodes	✓			✓
Interruptible region		✓		
Control flows	✓	✓	✓	✓
Object flows	✓			✓
Integration with state machines	✓			
Global variables				

CSP. The goal of the work described in [22,23] is on providing a formal semantics for activities in terms of CSP, rather than checking behavioural conformance. Only a limited number of diagrammatic constructs are considered, and object flows are omitted. Constructs such as send and receive event actions are not addressed.

Our work differs from that of the aforementioned contributions in a number of ways. Primarily, we present a compositional approach to refinement and specification, evaluated within the context of SysML. In addition, we consider the behaviour of several interacting state machines, supplemented with behaviours described via activities. In contrast, previous approaches placed emphasis on the formalisation of a single state machine (or activity); considering the execution semantics in terms of interaction with other state machines (or activities) was not the primary focus. Tables 1 and 2 attempt to draw comparisons between our work and that of others, with respect to the constructs addressed in the various contributions.

With respect to future work, we plan to integrate the semantics of state machine and activity diagrams with that of sequence diagrams in a CSP framework that encompasses all behavioural diagrams of SysML. To this end, a formal semantics for sequence diagrams in terms of CSP has already been given in [11].

The work described in this paper significantly extends the work of [12] by providing a more comprehensive semantics for both SysML activities and state machines. The purpose of [12] was to demonstrate an integrated semantics for activities and state machines, rather than formulate a comprehensive semantics of each. In this paper, we aim to demonstrate a comprehensive, integrated semantics considering both behavioural constructs. The case study presented in [13] is employed to illuminate and validate the contribution of this paper.

Another formal method often used to formalise UML and SysML diagrams is Petri nets: state machine diagrams are given a formal semantics in [3] and [5]; activity diagrams are formalised in [24].

In conclusion, the contributions of this paper are as follows.

– We have presented a formal model of SysML blocks using CSP. In particular, we have demonstrated one interpretation of SysML blocks for modelling and integrating system behaviour in a formal setting.
– We have presented an overarching behavioural semantics for state machines and activities. To the best of our knowledge, this is the first formalisation that encompasses and considers the combined behaviour of both of these constructs.
– We have demonstrated how CSP can be used in conjunction with SysML in a compositional, refinement-based approach to specification. The proposed methodology was evaluated using a case study that we would argue is ideally suited to illustrate the principles of systems engineering.

Acknowledgements The authors would like to thank the anonymous reviewers for their helpful and constructive comments.

References

1. Abdelhalim, I., Schneider, S.A., Treharne, H.: Towards a practical approach to check UML/fUML models consistency using CSP. In: Qin, S., Qiu, Z. (eds.) Proceedings of the 13th International Conference on Formal Engineering Methods (ICFEM 2011), Lecture Notes in Computer Science, vol. 6991, pp. 33–48. Springer, Berlin (2011)
2. Abdelhalim, I., Schneider, S.A., Treharne, H.: An integrated framework for checking the behaviour of fUML models using CSP. Int. J. Softw. Tools Technol. Transf. **15**(4), 375–396 (2012)
3. André, E., Benmoussa, M.M., Choppy, C.: Formalising concurrent UML state machines using coloured Petri nets. In: Nguyen, V.-H., Le, A.-C., Huynh, V.-N. (eds.) Knowledge and Systems Engineering, Advances in Intelligent Systems and Computing, vol. 326, pp. 473–486. Springer, Berlin (2015) et al. (2015)
4. Bolton, C., Davies, J.W.M.: Using relational and behavioural semantics in the verification of object models. In: Proceedings of the 4th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2000), IFIP Advances in Information and Communication Technology, vol. 49, pp. 163–182. Springer, Berlin (2000)
5. Choppy, C., Klai, K., Zidani, H.: Formal verification of UML state diagrams: a Petri net based approach. ACM SIGSOFT Softw. Eng. Notes **36**(1), 1–8 (2011)
6. Davies, J.W.M., Crichton, C.R.: Concurrency and refinement in the Unified Modeling Language. Electron. Notes Theor. Comput. Sci. **70**(3), 217–243 (2002)
7. Friedenthal, S., Moore, A., Steiner, R.: A Practical Guide to SysML: The Systems Modeling Language. Morgan Kaufmann Publishers, Los Altos (2008)
8. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3—a modern refinement checker for CSP. In: Ábrahám, E., Havelund, K. (eds.) Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014), Lecture Notes in Computer Science, vol. 8413, pp. 187–201. Springer, Berlin (2014)
9. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall, Englewood Cliffs (1985)
10. Jacobs, J., Simpson, A.C.: Towards a process algebra framework for supporting behavioural consistency and requirements traceability in SysML. In: Groves, L., Sun, J. (eds.) Proceedings of the 15th International Conference on Formal Engineering Methods (ICFEM 2013), Lecture Notes in Computer Science, vol. 8144, pp. 266–281. Springer, Berlin (2013)
11. Jacobs, J., Simpson, A.C.: On a process algebraic representation of sequence diagrams. In: Canal, C., Idani, A. (eds.) Proceedings of the 1st International Workshop on Safety and Formal Methods (SaFoMe 2014), Lecture Notes in Computer Science, vol. 8938, pp. 71–85. Springer, Berlin (2014)
12. Jacobs, J., Simpson, A.C.: A formal model of SysML blocks using CSP for assured systems engineering. In: Proceedings of the 3rd International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS 2014), Communications in Computer and Information Science, vol. 476, pp. 1–15. Springer (2015). (To appear)
13. Jacobs, J., Simpson, A.C.: On the formal interpretation of SysML blocks using a safety critical case study. In: Proceedings of the 8th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS 2014). IEEE (2015). (To appear)
14. Lewerentz, C., Lindner, T. (eds.): Formal development of reactive systems: case study production cell. In: Lecture Notes in Computer Science, vol. 891. Springer, Berlin (1995)
15. Ng, M.Y., Butler, M.: Towards formalizing UML state diagrams in CSP. In: Proceedings of the 1st IEEE International Conference on Software Engineering and Formal Methods (SEFM 2003), pp. 138–147. IEEE (2003)
16. Object Management Group: Unified Modeling Language Specification, version 2.4.1 (2011). http://www.omg.org/spec/UML/2.4.1 (March 2014)
17. Object Management Group: Systems Modeling Language Specification, version 1.3 (2012). http://www.omg.org/spec/SysML/1.3 (March 2014)
18. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall, Englewood Cliffs (1997)

19. Roscoe, A.W.: Understanding Concurrent Systems. Springer, Berlin (2010)
20. Sun, J., Liu, Y., Dong, J.S.: Model checking CSP revisited: introducing a process analysis toolkit. In: Proceedings of the 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2008), Communications in Computer and Information Science, vol. 17, pp. 307–322. Springer (2008)
21. Woodcock, J.C.P., Cavalcanti, A.L.C., Fitzgerald, J., Larsen, P.G., Miyazawa, A., Perry, S.: Features of CML: a formal modelling language for systems of systems. In: Proceedings of the 7th International Conference on System of Systems Engineering (SoSE 2012), pp. 1–6. IEEE (2012)
22. Xu, D., Miao, H., Philbert, N.: Model checking UML activity diagrams in FDR. In: Proceedings of the 8th International Conference on Computer and Information Science (ICIS 2009), pp. 1035–1040. IEEE (2009)
23. Xu, D., Philbert, N., Liu, Z., Liu, W.: Towards formalizing UML activity diagrams in CSP. In: Proceedings of the 2008 International Symposium on Computer Science and Computational Technology (ISCSCT 2008), pp. 450–453. IEEE (2008)
24. Yang, N., Yu, H., Sun, H., Qian, Z.: Mapping UML activity diagrams to analyzable Petri net models. In: Proceedings of the 10th International Conference on Quality Software (QSIC 2010), pp. 369–372. IEEE (2010)
25. Yeung, W.L., Leung, K.R.P.H., Dong, W., Wang, J.: Improvements towards formalizing UML state diagrams in CSP. In: Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC 2005), pp. 176–182. IEEE (2005)
26. Zhang, S.J., Liu, Y.: An automatic approach to model checking UML state machines. In: Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement Companion (SSIRI-C 2010), pp. 1–6. IEEE (2010)

Jaco Jacobs gained an honours degree in Engineering Sciences from the University of Stellenbosch, South Africa. Subsequently, he gained an M.Sc., with distinction, and a D.Phil. from the University of Oxford. He now works as a software engineer for Perspectum Diagnostics.

Andrew Simpson received a first-class honours degree in Computer Science from the University of Wales, Swansea, and an M.Sc., and a D.Phil., from the University of Oxford. He is currently a University Lecturer in Software Engineering at the University of Oxford.