# A Formal Model of SysML Blocks using CSP for Assured Systems Engineering

Jaco Jacobs and Andrew Simpson

Department of Computer Science, University of Oxford
Wolfson Building, Parks Road
Oxford OX1 3QD
{jaco.jacobs, andrew.simpson}@cs.ox.ac.uk

**Abstract.** The Systems Modeling Language (SysML) is a semi-formal, visual modelling language used in the specification and design of systems. In this paper, we describe how Communicating Sequential Processes (CSP) and its associated refinement checker, Failures Divergences Refinement (FDR), gives rise to an approach that facilitates the refinement checking of the behavioural consistency of SysML diagrams. We formalise the conjoined behaviour of key behavioural constructs — state machines and activities — within the context of SysML. Furthermore, blocks, the fundamental modelling construct of the SysML language, can be combined in a compositional approach to system specification. The use of a process-algebraic formalism enables us to explore the behaviour of the resulting composition more rigorously. We demonstrate how CSP, in conjunction with SysML, can be used in a formal top-down approach to systems engineering. A small case study validates the contribution.

## 1 Introduction

Accidents associated with complex systems are frequently the result of unforeseen interactions amongst components that all satisfy their individual requirements [1]. These *component interaction accidents* are increasingly common: state of the art systems are more interdependent on other technologically advanced systems and interact in ways not foreseen or intended by the original designer. The *Mars Polar Lander* accident is one example of such a failure: both the landing legs and the control software of the descent engines functioned as specified by their respective behavioural specifications. The systems engineers, however, did not consider all the potential interactions between the landing legs and the control software of the descent engines [1].

The OMG's *Systems Modeling Language* (SysML) [2] is a graphical modelling notation used in the specification and integration of complex, large-scale systems. A keystone of this activity is ensuring that requirements, as imposed by the various stakeholders, are adequately captured and subsequently addressed when specifying a potential solution. The intention of SysML, thus, is to accurately specify intended component behaviour with the expectation to minimise interaction accidents. However, SysML is a semi-formal notation. If we are to carry out an extensive analysis of component interactions, more mathematical rigour is indispensable.

Reasoning about behaviour — in particular, the myriad of interactions between components — is a rather cumbersome activity for the human mind. In addition, our cognitive ability to cope with multiple, separate descriptions of behaviour, and ultimately fuse these into a unified interpretation, is rather limited. We need to augment our faculties with appropriate notations in order to effectively reason about such behaviours. Moreover, if we are going to utilise these notations in a meaningful fashion, we require mechanised tool support. *Communicating Sequential Processes* (CSP) [3] is one such notation, backed up by *Failures Divergences Refinement* (FDR) in the form of a refinement checker.

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 [2]. In this paper, we provide a behavioural semantics for the conjoined behaviour of state machines and activities. In the past, there have been several contributions where the sole focus lied either with the formalisation of state machines, or 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 stance with regards 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 conduct 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.

This paper is a companion of sorts to the work presented in [4]: it extends the formalisation of state machines to encompass entry, exit, and do behaviours modelled via activities. In doing so, a formal behavioural semantics is provided for activities, in terms of CSP.

The structure of the remainder of this paper is as follows. In Section 2, we provide a brief introduction to SysML. Section 3 outlines our process-algebraic approach to formalise SysML activities, state machines, and blocks. We show how CSP can be employed to analyse expositions composed of multiple, communicating state machine and activity constructs. In Section 4, we employ a small case study to illuminate and validate the contribution. Section 5 summarises the contributions of this paper, and places it in context with respect to other research.

## 2 Background

In this section, we give a necessarily brief introduction to SysML. We assume familiarity with CSP.

**Blocks** Blocks are the fundamental 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. The classifier behaviour of a block can serve as an abstraction of the behaviours of its parts. Thus, 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. This interpretation also sits well with the concept of refinement and abstraction in CSP.

The *classifier behaviour* is the main behaviour of a block, and executes 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.

Typically, two block instances communicate using signal events. 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 *signal* is a classifier that types the asynchronous messages that are communicated between blocks. Each signal optionally has an associated set of attributes which correspond to the parameters that make up the content of the message. 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.

**Activities** *Activities* allow the modeller to describe complex routes along which actions execute. These routes are termed *flows*. In SysML activities there are two types of flows: control flows and object flows.

*Actions* are the fundamental building blocks of *activities* and always execute within the context of an activity. An action accepts inputs and produces outputs. The flow of input and output items between actions are described using *object flows*. *Control flows*, on the other hand, impose additional constraints on the execution of actions. When a control flow connects one action to another, the target action cannot start until the source action has completed. *Control nodes* are used in the specification of control flow: they are used to impose control logic on the execution of actions. The control nodes are the fork, join, decision, merge, initial and final nodes.

Several types of actions exist: the *send signal event action* sends a signal event; the *receive signal event action* waits on the receipt of a particular signal event; and the *value specification action* allows the specification of a particular value to an input of an action. *Opaque actions* allow the specification of actions in a language external to SysML.

**State machines** *State machines* graphically depict state-dependent behaviour in terms of nodes and labelled edges: nodes represent states, whereas the edges correspond to transitions between states.

In SysML, a *state* is an abstraction of the mode that the owning block finds itself in. A change of state is effected by the arrival of a triggering event, causing an appropriate transition to fire. A *transition* consists of a trigger, a guard and an effect. The *trigger* denotes the event that serves as stimulus for the transition to fire; the *guard* is a conditional expression used to decide whether the transition is to fire at all; and the *effect* is a supplementary behaviour that executes on the transition.

## 3 A CSP view of SysML blocks

This section outlines an approach to integrate the semi-formal SysML notation with the process algebra CSP. In order to define a formal semantics for blocks, parts and state machines, we need a precise description of their syntax. To this end, we define simple mathematical constructs that are closely related to the syntactical structure of their corresponding SysML counterparts.

**Activities** Broadly speaking, our approach maps every node and every edge in an activity diagram to a CSP process. We restrict actions to either have either a single outgoing control or object flow, but not both; our semantics allows for simple forks and joins in the sense that a fork node splits control into multiple flows that eventually all end in a corresponding join node. We present the formalisation as it relates to a single activity $A$; $\mathcal{A}$ denotes the set containing all activities in our universe of discourse.

An *activity* $A \in \mathcal{A}$ consists of a finite collection of *nodes*, denoted $N_A$, and *edges* between those nodes, denoted $E_A$. We partition $N_A$ such that $N_A^I$ represents the set of *initial nodes*, $N_A^F$ the set of *final nodes*, $N_A^{FK}$ the set of *fork nodes*, $N_A^{JN}$ the set of *join nodes*, $N_A^{SS}$ the *send signal event actions*, $N_A^{RS}$ the *receive signal event actions*, $N_A^O$ the *opaque actions*, and $N_A^{PN}$ the set of activity parameter nodes. The edges are partitioned such that $E_A^{OF}$ represents the object flows, and $E_A^{CF}$ represents the set of control flows.

We define the following functions, to return for a particular flow $f \in E_A$: the source node, $\mathit{source} : E_A \to N_A$; and the target node, $\mathit{target} : E_A \to N_A$. Additionally, we define functions to return for a particular node $n \in N_A$: the set of outgoing control flows, $\mathit{outgoing}_{cf} : N_A \nrightarrow \mathbb{P}\, E_A^{CF}$; and the outgoing object flow, $\mathit{outgoing}_{of} : N_A \nrightarrow E_A^{OF}$. Assume that the construction $\mathit{name}(n)$ returns the name of the send or receive signal event, or opaque action for $n \in N_A^{SS} \cup N_A^{RS} \cup N_A^O$.

The formalisation makes use of a mapping function $\mathcal{F}$. In particular, $\mathcal{F}(A, c)$ is the process modelling the construct $c$, either an edge or a node, of activity $A$.

*Activity parameter node.* An activity parameter node $n \in N_A^{PN}$, models a parameter, $p$, that can be used within the context of the activity. In CSP, the node is modelled as an argument to the process modelling the activity. Diagrammatically, an object flow $\mathit{of} \in E_A^{OF}$ connects the parameter node with other nodes that use this as a parameter. For the purpose of this paper we assume that a single argument is represented by each activity parameter node that serve as input to the activity. The activity's behaviour starts as the process modelling the initial node $n_0 \in N_A^I$

$$\begin{array}{l} A(p) = \\ \quad \mathsf{let} \\ \quad\quad \mathcal{F}(A, n_0) = \ldots \\ \quad \mathsf{within} \\ \quad\quad \mathcal{F}(A, n_0) \end{array}$$

An activity without a parameter is modelled similarly, but the process parameter $p$ is elided.

*Control flow edge.* A control flow $\mathit{cf} \in E_A^{CF}$ 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 $\mathit{target}(\mathit{cf})$. If the target is not a join node, i.e. $\mathit{target}(\mathit{cf}) \notin N_A^{JN}$, the process simply designates its behaviour to be that of the target node.

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

In the case where $\mathit{target}(\mathit{cf}) \in N_A^{JN}$, there will be, based on our assumption of activities above, $k-1$ other control flows which terminate in the same join node. Let the control flows be $\mathit{cf}_0 \mathrel{..} \mathit{cf}_{k-1}$. Exactly one of the control flows, $\mathit{cf}_0$, will exhibit the behaviour of the join node.

$$\begin{array}{ll} \mathit{Join}(e) = & \\ \quad \mathit{join} \rightarrow \mathit{Skip} & \text{if } \ e \neq \mathit{cf}_0 \\ \quad \mathit{join} \rightarrow \mathcal{F}(A, \mathit{target}(e)) & \text{otherwise} \end{array}$$

The above construction ensures that exactly one of the previously forked flows continues after the join. Many interpretations of activity diagrams assume control flows to have associated guards, typically expressed in natural language. Due to obvious reasons natural language guards are not suitable for a precise behavioural semantics and are thus excluded.

*Object flow edge.* An object flow $\mathit{of} \in E_A^{OF}$ is used to model the passing of parameters[1] between activity parameter nodes, call behaviour actions or send and receive signal events. The behaviour of an object flow edge is a parametrised process that takes as input the value of the argument, say $p$, passed along the

[1] We restrict ourselves to signal parameters here, although in SysML these can be any classifier that can serve as an input to an activity.

object flow. Throughout, process arguments are placed within square brackets to denote them as such.

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

*Initial node.* An initial node $n \in N_A^I$ has a single outgoing edge, a control flow $\mathit{cf} \in \mathit{outgoing}_{\mathit{cf}}(n)$. The process behaves like the control flow edge emanating from the initial node.

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

*Send signal event action.* A send signal event action $n_1 \in N_A^{SS}$ has a single outgoing control flow $\mathit{cf} \in \mathit{outgoing}_{\mathit{cf}}(n_1)$.

$$\mathcal{F}(A, n_1) = \mathit{name}(n_1) \rightarrow \mathcal{F}(A, \mathit{cf})$$

Optionally, an incoming object flow $\mathit{of}$ is possible, which serves as input to the send signal event action, and models the parameters send as part of the send signal event. In our semantics, the object flow $\mathit{of}$, if present, emanates from an activity parameter node $n_2 \in N_A^{PN}$ and terminates on send signal event[2] node $n_1$[3]. The construction $\mathit{par}(n_2)$ is the parameter available within the context of the owing activity (defined within the let within construct).

$$\mathcal{F}(A, n_1) = \mathit{name}(n_1).\mathit{par}(n_2) \rightarrow \mathcal{F}(A, \mathit{cf})$$

Alternatively, the send signal event has a single incoming object flow, but no incoming control flow. In this case the process modelling the send signal event action would have an input argument, $p$, passed from the process modelling the object flow. The outgoing control flow is given by $\mathit{cf} \in \mathit{outgoing}_{\mathit{cf}}(n_1)$. The formalisation follows.

$$\mathcal{F}(A, n_1)[p] = \mathit{name}(n_1).p \rightarrow \mathcal{F}(A, \mathit{cf})$$

The above models the case where the parameter comes from: an object flow emanating from a value specification action; the output of an opaque action; or the output of a receive signal event action.

*Receive signal event action.* A receive signal event action $n \in N_A^{RS}$ has a single outgoing control flow $\mathit{cf} \in \mathit{outgoing}_{\mathit{cf}}(n)$. Note that it is not possible to have an outgoing object flow if an outgoing control flow is present.

$$\mathcal{F}(A, n) = \mathit{name}(n) \rightarrow \mathcal{F}(A, \mathit{cf})$$

Alternatively, the receive signal event may be passed a parameter as part of the event. In this case it is conceivable that an object flow will exit the action. The formalisation follows.

$$\mathcal{F}(A, n) = \mathit{name}(n)?p \rightarrow \mathcal{F}(A, \mathit{outgoing}_{\mathit{of}}(n))[p]$$

[2] A *value specification action*, rather than an activity parameter node, connected via an object flow, can be used for constants.

[3] Note that an incoming control flow is still present and also terminates on $n_1$.

The input $p$ on the CSP channel corresponds to the parameter passed as part of the receive signal event.

*Final node.* A final node $n \in N_A^F$ has no outgoing edges. It is trivially modelled as the CSP *Skip* process.

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

*Fork node.* A fork node $n \in N_A^{FK}$ splits the control flow in $k$ parallel flows $cf_0 \mathinner{\ldotp\ldotp} cf_{k-1}$.

$$\mathcal{F}(A, n) = [|\mathit{join}\,|]\; j : \mathit{outgoing}_{cf}(n) \bullet \mathcal{F}(A, j)$$

The above alphabetised indexed parallel construction ensures that all the different threads of control only synchronise on the *join* event; all other events are interleaved.

*Join node.* A join node $n \in N_A^{JN}$ synchronises $k$ parallel control flows and has a single outgoing control flow $cf = \mathit{outgoing}_c f(n)$.

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

**State machines** This paper is a companion of sorts to the work presented in [4]: it extends the formalisation of state machines to encompass entry, exit, and do behaviours modelled via activities. This hybrid approach is typical of most systems engineering methodologies used in practice today. In addition, as the activities execute within the context of an owing state machine, the run to completion execution semantics of state machines are applicable. We briefly reprise the necessary mathematical structures and CSP descriptions of [4] to ensure this paper is self-contained. We restrict ourselves to non-hierarchical state machines and ignore guard conditions on transitions in order to simplify the presentation here. The interested reader can refer to [4] for an account of more complex state machines.

A *state machine* $M \in \mathcal{M}$ consists of a finite set of *states*, denoted $S_M$, and *transitions* between those states, denoted $T_M$. We partition $S_M$ such that $S_M^I$ represents the set of *initial states*, $S_M^F$ the set of *final states*, $S_M^S$ the set of *simple states.* A function $\mathit{outgoing} : S_M \to \mathbb{P}\, T_M$ returns the set of outgoing transitions for a given state.

We define the following functions, to return for a transition $t \in T_M$: the source state, $\mathit{source} : T_M \to S_M$; the target state, $\mathit{target} : T_M \to S_M$; the trigger, $\mathit{trigger} : T_M \to \mathcal{S}$; and the effect, given by $\mathit{effect} : T_M \to \mathcal{A}$. $\mathcal{S}$ is the set of signals.

The entry and exit behaviours of a particular state are given by the following functions: $\mathit{entry} : S_M \to \mathcal{A}$; and $\mathit{exit} : S_M \to \mathcal{A}$. In each case, an activity modelling the behaviour is returned.

A mapping function $\mathcal{F}$ is used to formalise the behaviour; $\mathcal{F}(M, s)$ is a process that describes the behaviour of $M$ in state $s$.

*Initial state.* An initial state $s \in S_M^I$ has a single outgoing transition $t$ that defines its unique starting point. Optionally, an effect component can be specified for the transition using an activity $A \in \mathcal{A}$. In the following: $\mathit{effect}(t)$ returns a behaviour specified via an activity; similarly, $\mathit{entry}(\mathit{target}(t))$ returns the entry behaviour of the target state specified via an activity.

$$\mathcal{F}(M, s) = \mathit{effect}(t) \fatsemi \mathit{entry}(\mathit{target}(t)) \fatsemi \mathcal{F}(M, \mathit{target}(t))$$

*Simple state.* The CSP channel *local* is used for communicating with the event queue of the state machine $M$. The arrival of a SysML signal event serves as the trigger; consequently this is made available as a CSP event. If the signal signature has a data component associated with it, this is made available as an input along with the channel modelling the event[4].

We need to consider the eventuality where the state machine receives a signal event not expected in the current state $s$. Here, the state machine discards the unexpected event. In the following, assume that $\mathit{unexpected}(s)$ returns the set of unexpected events for state $s$ (receive signal events that are valid in other states of $S_M$ but not in $s$). The components *proc* and *disc* denote the event being processed and discarded, respectively. In both cases, it is removed from the event queue.

$$\begin{array}{l}
\mathcal{F}(M, s) = \\
\quad \Box\, t : \mathit{outgoing}(s) \bullet \mathit{local.proc.trigger}(t) \rightarrow \\
\qquad \mathit{exit}(s) \fatsemi \mathit{effect}(t) \fatsemi \mathit{entry}(\mathit{target}(t)) \fatsemi \mathcal{F}(M, \mathit{target}(t)) \\
\quad \Box \\
\quad \Box\, t : \mathit{unexpected}(s) \bullet \mathit{local.disc.trigger}(t) \rightarrow \mathcal{F}(M, s)
\end{array}$$

*Final state.* Consider a final state $s \in S_M^F$. A final state has no outgoing transitions and is trivially modelled as the deadlocked process.

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

*Event queue.* The state machine as a whole is modelled with a single process that contains all the localised process descriptions defined above. The overall structure is similar to that given by Davies and Crichton [5]. The state machine receives all communications through an event queue, 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 overall process $M(\mathit{queue}, \mathit{local})$ initially behaves as the process associated with the initial state $\mathcal{F}(M, s_0)$. Throughout, the state machine behaves like the various processes until it possibly reaches a final state, after which it behaves as $\mathcal{F}(M, s_f)$. The local process $EQ$ models the event queue. Here, we assume a queue with a maximum capacity of 1; the queue blocks when full. The datatype *Dispatcthed*, communicated along with the event on channel

[4] Next, the guard (if it exists) is evaluated and if false the event is discarded without effect. Conversely, if the guard evaluates to true the behavioural construct specified for the effect are executed before behaving as the process associated with the destination state. Guards are omitted in this paper due to space restrictions.

*local*, models the dispatching of an event: an event can either be processed, *proc* or, if the state machine is in a state where the dispatched event is not expected, discarded, *disc*.

```
$M(queue, local) =$
  let
    $\mathcal{F}(M, s_0) = \ldots$
    $\ldots$
    $\mathcal{F}(M, s_f) = Stop$
    $EQ = queue?e \rightarrow local?p!e \rightarrow EQ$
  within
    $\mathcal{F}(M, s_0) \; [| \; \{| \; in \; |\} \; |] \; EQ$
```

The state machine of a block $B_i$ only receives (through its event queue) the provided receptions. The required features are communicated across the connectors linking parts. In our formalisation, the name of the part is used as the channel name.

**Blocks** The formalisation above additionally allows us to showcase how CSP can be used in a compositional approach to specification and refinement within the context of systems engineering.

Assume a block $B_i \in \mathcal{B}$ composed of $K$ constituent blocks $B_0 \mathinner{..} B_{K-1}$, where $i \geq K$. We known 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.

$$\mathit{classifier}(B_i) \sqsubseteq \big\| \; P : \{B_0 \mathinner{..} B_{K-1}\} \bullet \mathit{classifier}(P)$$

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.

## 4 A robotic arm

In this section we apply the concepts central to our methodology to an illustrative case study. We study a single component, a robotic arm, of a fully fledged case

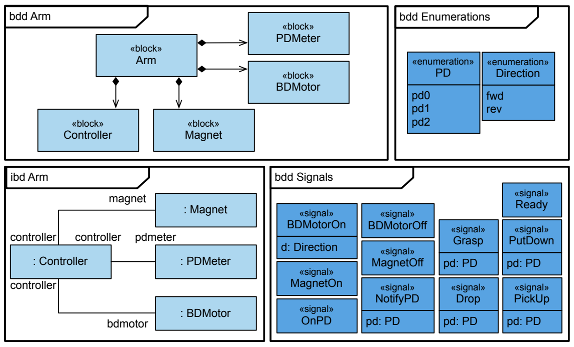

**Fig. 1.** The block definition and internal block diagrams of the arm system.

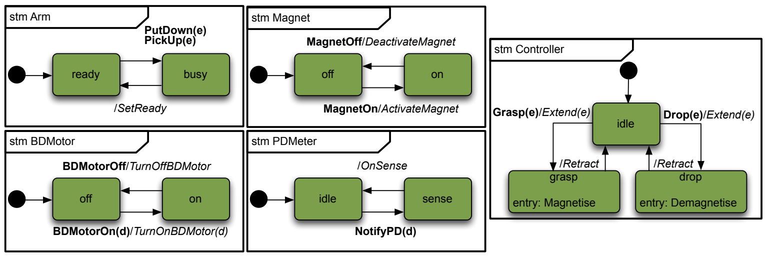

**Fig. 2.** The state machine diagrams of the arm system.

study that is well known in the formal methods community. The production cell is an industrial installation of a metal processing plant located in Karlsruhe, Germany [6]. 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.

A *bidirectional motor* can operate in two opposing directions. An *electromagnet* can activate or deactivate a magnetic field using an electric current. A *potentiometer* 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.

Refer to Figure 1. The structural aspects of the system are modelled using blocks for the controller, bidirectional motor, electromagnet, and the potentiometer; signals and enumeration definitions further illuminate the design by introducing the messages and associated parameters communicated between state machines and activities.

Figures 2 and 3 show the state machines and activities of the arm system.

The channels used by the state machine of the bidirectional motor can be defined thus. The Direction enumeration of Figure 1 can be represented with a CSP datatype. Channel and datatype definitions for other state machines are similar.

```
datatype $Dispatched = proc \mid disc$
datatype $Direction = fwd \mid rev$
datatype $BDMotorSignal =$
  $BDMotorOn.Direction \mid BDMotorOff$
channel $bdmotor : BDMotorSignal$
channel $bdmotorlocal : Dispatched.BDMotorSignal$
```

In the above, the channel *bdmotor* is used by other state machines to communicate with the state machine of the bidirectional motor via its associated event queue; the channel *bdmotorlocal* is used by the event queue of the bidirectional motor to dispatch events (to the bidirectional motor's state machine) for processing.

The CSP process modelling the characteristic behaviour of the Controller follows. The activity 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. CSP datatype definitions are used to type the provided receptions of the Controller block; these serve as triggers for the classifying state machine. The name of the instance is used as the channel name when communicating with a state machine; a channel with the same name and the suffix local is used to model the internal event queue of the corresponding state machine.

```
$Controller(queue, local) =$
  let
    $I_0 = IDLE$
    $IDLE =$
      $local.proc.Grasp?e \rightarrow$
        $Extend(local, e) \fatsemi Magnetise \fatsemi GRASP$
      $\Box$
      $local.proc.Drop?e \rightarrow$
        $Extend(local, e) \fatsemi Demagnetise \fatsemi DROP$
      $\Box$
      $local.disc?e : \{\mid OnPD \mid\} \rightarrow IDLE$
    $GRASP =$
      $Retract(local) \fatsemi IDLE$
      $\Box$
      $local.disc?e : \{\mid Grasp, Drop, OnPD \mid\} \rightarrow GRASP$
    $DROP = \ldots$
    $EQ = queue?e \rightarrow local?p!e \rightarrow EQ$
  within
    $I_0 \; [\mid \{\mid local \mid\} \mid] \; EQ$
$CONTROLLER = Controller(controller, controllerlocal)$
```

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

The processes *Magnetise* and *Extend*, modelling the activities used in the *CONTROLLER* process, follows. The event queue is passed in as the activity executes within the context of its owing state machine.

$$
\begin{array}{l}
Magnetise = \\
\quad \mathsf{let} \\
\qquad I_0 = SS_0 \\
\qquad SS_0 = magnet.magnetOn \rightarrow F_0 \\
\qquad F_0 = Skip \\
\quad \mathsf{within} \\
\qquad I_0 \\
\alpha Magnetise = \{|\ magnet.MagnetOn\ |\}
\end{array}
$$

$$
\begin{array}{l}
Extend(local, pd) = \\
\quad \mathsf{let} \\
\qquad I_0 = VS_0 \\
\qquad VS_0 = SS_0(fwd) \\
\qquad SS_0(o) = bdmotor.BDMotorOn.o \rightarrow SS_1 \\
\qquad SS_1 = pdmeter.NotifyPD.pd \rightarrow RS_0 \\
\qquad RS_0 = \\
\qquad\quad local.proc.OnPD \rightarrow SS_2 \\
\qquad\quad \Box \\
\qquad\quad local.disc?ev : \{|\ Grasp, Drop\ |\} \rightarrow RS_0 \\
\qquad SS_2 = bdmotor.BDMotorOff \rightarrow F_0 \\
F_0 = Skip \\
\quad \mathsf{within} \\
\qquad I_0 \\
\alpha Extend = \\
\quad \{|\ bdmotor.BDMotorOn.fwd, bdmotor.BDMotorOff, \\
\qquad pdmeter.NotifyPD\ |\}
\end{array}
$$

The processes, along with their respective alphabets, denoting concrete parts for the magnet, bidirectional motor and potentiometer can be similarly defined, but are excluded here due to space constraints. Activities and alphabets used within these state machines can also be similarly defined.

$$
\begin{array}{l}
MAGNET = Magnet(magnet, magnetlocal) \\
BDMOTOR = BDMotor(bdmotor, bdmotorlocal) \\
PDMETER = PDMeter(pdmeter, pdmeterlocal)
\end{array}
$$

The definition of the process *ARM*, modelling the abstract block that serves as the specification that the parts must realise, follows.

$$
Arm(queue, local) =
$$

```
let
  $I_0 = READY$
  $READY = \ldots$
  $BUSY =$
    $SetReady \fatsemi READY$
    $\Box$
    $local.disc?e : \{| \, PickUp, PutDown \, |\} \rightarrow BUSY$
  $EQ = queue?e \rightarrow local?p!e \rightarrow EQ$
within
  $I_0 \, [| \, \{| \, local \, |\} \, |] \, EQ$
$ARM = Arm(arm, armlocal)$
$\alpha ARM =$
  $Union(\{\{| \, arm, armlocal \, |\}, \alpha SetReady\})$
```

Assuming that $P = \{CONTROLLER, MAGNET, BDMOTOR, PDMETER\}$ we then have $CONCRETE = \parallel p : P \bullet [\alpha p]p$. In the aforementioned, $\alpha p$ denotes the set of events communicable by $P$. 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.

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

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

The set $Hidden$ are those events not present in the alphabet of the abstract specification process $ARM$; $\Sigma$ denotes the set of all CSP events within the context of the specification. Thus

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

FDR verifies the assertion

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

Given that the refinement holds, $ARM$ can be substituted for its parts in the complete system: the behaviour of the concrete implementation processes, denoted by $CONCRETE$, can neither refuse nor accept an event that $ARM$ can. Stated another way, the characteristic behaviour of $CONCRETE$ 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.

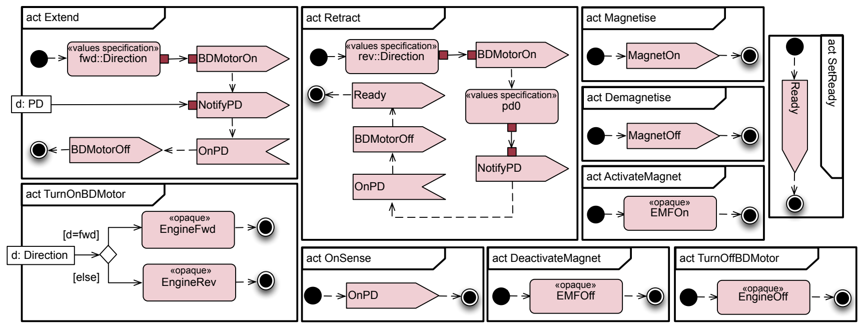

**Fig. 3.** The activity diagrams of the arm system.

## 5 Conclusions

There is a wealth of literature on the formalisation of activity and state machine diagrams, primarily within the context of UML. In order to limit the scope we only report on approaches that utilise CSP.

Ng and Butler [7] proposed the formalisation of UML state machine diagrams using CSP as the semantic domain [7]. They define the translation in terms of a mapping function from structural diagrammatic constructs to their CSP counterparts. The work of Yeung and colleagues [8] built on that of Ng and Butler by generalising inter-level transitions.

Xu et al. [9] formalised activity diagrams in CSP. A transformation function is defined that maps the mathematical representation of an activity to the semantic domain of CSP. The goal in [9] 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 is different than the aforementioned contributions in a number of ways. This paper presents 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 their primary focus.

The choice of CSP is due to a number of factors. The behavioural aspects of SysML can be modelled naturally by a process-algebraic formalism such as CSP, resulting in a formal framework where assertions about requirements can be proved or refuted with relative ease [4]. CSP's approach to process composition, combined with the fact that refinement is preserved within context, would allow us to decompose a complex design of a system (or system of systems) in such a way that the automated analysis is computationally feasible. In particular, the decompositional approach to specification, as illuminated by the case study in Section 4, allows us to substitute a collection of blocks with a single block that depicts the intended behaviour of the whole. Furthermore, CSP's approach to establish refinement — by comparing the behaviour of a characteristic specification process to that of a concrete implementation process — coincides with SysML's compositional outlook to specification and the notion that a block can act as a specification of constituent blocks. In contrast, in conventional model checking approaches where there is no concept of refinement, this distinction is less clear.

## References

1. Leveson, N.G.: Engineering a Safer World: Systems Thinking Applied to Safety. MIT Press (2012)
2. Object Management Group: Systems Modeling Language Specification, version 1.3. (2012) Available at: http://www.omg.org/spec/SysML/1.3, [2014, March].
3. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall (1985)
4. Jacobs, J., Simpson, A.: Towards a process algebra framework for supporting behavioural consistency and requirements traceability in SysML. In: Proceedings of the 15th International Conference on Formal Engineering Methods (ICFEM 2013). Volume 8144 of Lecture Notes in Computer Science. Springer (2013) 266–281
5. Davies, J.W.M., Crichton, C.R.: Concurrency and refinement in the Unified Modeling Language. Electronic Notes in Theoretical Computer Science **70**(3) (2002) 217–243
6. Lewerentz, C., Lindner, T.: Case study Production Cell. In: Formal Development of Reactive Systems. Volume 891 of Lecture Notes in Computer Science. Springer (1995)
7. Ng, M.Y., Butler, M.: Towards formalizing UML state diagrams in CSP. In: Proceedings of the 1st International Conference on Software Engineering and Formal Methods (SEFM 2003), IEEE (2003) 138–147
8. 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), IEEE (2005) 176–182
9. 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), IEEE (2008) 450–453