On the Formal Interpretation of SysML Blocks using a Safety Critical Case Study

Jaco Jacobs & Andrew Simpson
Department of Computer Science, University of Oxford
Wolfson Building, Parks Road
Oxford OX1 3QD
Email: {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), can be used in conjunction with SysML in a formal top-down approach to systems engineering. Typically, a system is composed from constituent systems or components using the concept of blocks. SysML allows two alternative interpretations with regards to the behaviour of the resulting composition. By making use of a process-algebraic formalism we are able to explore these interpretations more rigorously. A case study is used throughout to illuminate the concepts in an informal manner.**

I. Introduction

The *Systems Modeling Language* (SysML) [1], 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. These, in turn, might be simple structural elements, or might themselves be viewed as systems comprised of various components working together.

Modelling a system with SysML relies on the concept of blocks — each with an associated set of states — that communicate 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 allow replacing an abstract block with a composition of parts, but the big drawback of this decomposition is that it is at best semi-formal and cannot guarantee consistency between a block and its parts. However, there are two alternative interpretations with regards to the combined behaviour [2].

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.

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

Communicating Sequential Processes (CSP) [3] is a process algebra 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 these differing notions of composition by making use of an illustrative case study. Moreover, the associated refinement checker, *Failures Divergences Refinement* (FDR) [4], gives rise to a practical approach that enables us to reason about these interactions in a formal setting.

The structure of the remainder of this paper is as follows. In Section II, we provide a brief introduction to CSP. In Section III we employ a small case study to show how CSP can be employed to analyse expositions composed of multiple, communicating state machine and activity constructs. Section IV formalises the different interpretations that can be assigned to combined block behaviour using the case study as exemplar. In Section V we review related literature. Section VI summarises the contributions of this paper.

II. Background

In this section, we give a necessarily brief introduction to CSP. We assume familiarity with SysML, although the necessary concepts are explained throughout Section III.

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 either be a primitive event, or can take a more structured, message-passing form, utilising channels. The message-passing mechanism is fundamentally based on the principle of a rendezvous between a sending and a receiving process: if the communication takes place on channel c, and a sending process wants to output a value e, 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 $c?x$.

CSP is compositional in the sense that it provides operators that allow us to define a process in terms of other, constituent processes. The CSP syntax utilised in this paper can be defined thus:

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

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

Stop is the deadlocked CSP process: it will refuse all and any event and never communicates. *Skip* is the process that communicates the special internal event $\checkmark$, before behaving like *Stop*; it is used to model successful termination.

The process $e \to P$, modelled using the *prefixing* operator, performs the event e and subsequently behaves as 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 process. Indexed versions exist for both operators.

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

The process $P_1 \mathbin{;} P_2$ represents the *sequential composition* of 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. The process $P_1 \,[|\, X \,|]\, P_2$ uses the *generalised parallel* operator to define an interface 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 $X \cap Y$. The *interleaving* operator, $|||$, expresses the unsynchronised concurrent interleaving of the events of its constituent processes. Indexed forms exist for these parallel operators.

A conditional choice construct is available in the form *if b* then P_1 else P_2, where a process behaves as P_1 if b is true and P_2 otherwise. The *let within* construct allows us to use local definitions (of the form of $P_1, .., P_n$) in the definition of a complex process.

CSP (and FDR) allows us to compare the behaviour of one process against that of another. Our concern here is in terms of *traces*. The traces of a process P, written $\mathit{traces}\,[\![P]\!]$, is the set of all finite sequences of observable events. As an example, $\mathit{traces}\,[\![\, a \to b \to \mathit{Stop} \sqcap c \to d \to \mathit{Stop} \,]\!]$ is the set

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

In the following, P/t represents the process P after the trace t; $\mathit{refusals}\,[\![P]\!]$ represent the initial set of events that P may refuse, no matter how long they are offered. The $\mathit{failures}\,[\![P]\!]$ are the pairs of the form (t, X) such that, for all $t \in \mathit{traces}\,[\![P]\!]$, $X = \mathit{refusals}\,[\![P/t]\!]$.

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

$$P_1 \sqsubseteq_T P_2 \Leftrightarrow \mathit{traces}\,[\![P_2]\!] \subseteq \mathit{traces}\,[\![P_1]\!]$$

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

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

The refinement checker Failures-Divergence Refinement (FDR) — which utilises the machine-readable dialect of CSP, CSP_M [4] — uses this theory of refinement to investigate whether a potential design meets its specification. A pleasing feature of FDR is that if such a test fails, a counter-example is returned to indicate why this is so.

Similarly, we define *failures refinement* as

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

III. 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 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 [5]. 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.

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. Conversely, *sensors*, as the name suggests, are sensory components that send inputs to the system controller. Examples of actuators are bidirectional motors and electromagnets. A *bidirectional motor* can operate in two opposing directions. An *electromagnet* 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 slightly different angles, related to the different interpretations which can be attributed to the composition of the composing block. The Arm

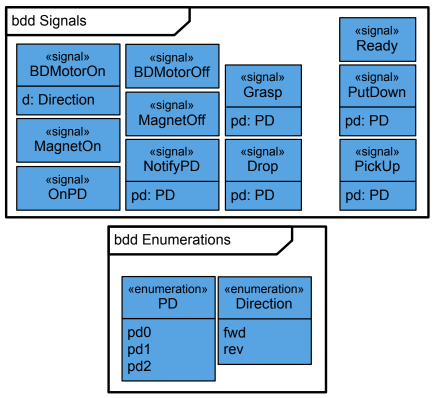

Fig. 1. The block definition diagram introducing Signals and Enumerations.

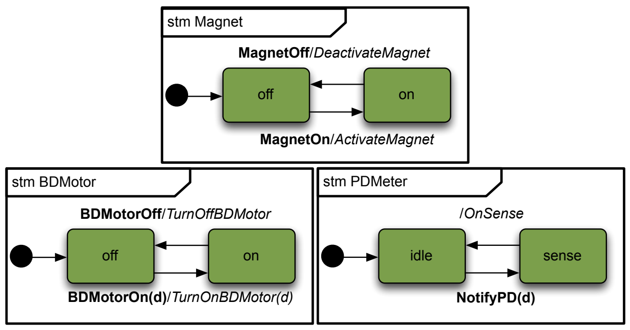

Fig. 2. The state machine diagrams of the Magnet, BDMotor and PDMeter blocks.

is the block of interest for the purposes of the case study. In this section 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 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 like state machines and activities.

A. Enumerations and Signals

Refer to Figure 1. 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 Figure 1 can be represented with CSP datatypes.

$$\text{datatype } \mathit{Direction} = \mathit{fwd} \mid \mathit{rev}$$
$$\text{datatype } \mathit{PD} = \mathit{pd}_0 \mid \mathit{pd}_1 \mid \mathit{pd}_2$$

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

The signals used by the communicating state machines are similarly defined. 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.

$$\begin{aligned}&\text{datatype } \mathit{BDMotorSignal} = \\ &\quad \mathit{BDMotorOn.Direction} \mid \mathit{BDMotorOff} \\ &\text{datatype } \mathit{MagnetSignal} = \mathit{MagnetOff} \mid \mathit{MagnetOn} \\ &\text{datatype } \mathit{PDMeterSignal} = \mathit{NotifyPD.PD}\end{aligned}$$

SysML blocks 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. 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.

$$\begin{aligned}&\text{channel } \mathit{bdmotor} : \mathit{BDMotorSignal} \\ &\text{channel } \mathit{bdmotorlocal} : \mathit{Dispatched.BDMotorSignal} \\ &\text{channel } \mathit{magnet} : \mathit{MagnetSignal} \\ &\text{channel } \mathit{magnetlocal} : \mathit{Dispatched.MagnetSignal} \\ &\text{channel } \mathit{pdmeter} : \mathit{PDMeterSignal} \\ &\text{channel } \mathit{pdmeterlocal} : \mathit{Dispatched.PDMeterSignal}\end{aligned}$$

For example, the channel *bdmotor* is used 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 for processing. The datatype *Dispacthed* 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 assumption above, any block connected to *BDMotor* via a connector uses the channel *bdmotor* to send signal events destined for *BDMotor*.

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

B. State Machines

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.

Figure 2 shows the state machines of the magnet, bidirectional motor and potentiometer. Activities that execute on the transitions as the effect component are italicised after the trigger, set in bold typeface. Activities are shown in Figure 3 and are discussed in Section III-C. For now, it is sufficient to assume that these are modelled using CSP processes.

The CSP processes modelling the state machines for the BDMotor, Magnet and PDMeter blocks follow. Local process definitions model each state in the associated state machine. The deterministic choice between permissible triggers are offered to the external environment. If a CSP event corresponding to a permitted SysML triggering event is received:

- the process modelling the exit behaviour of the source state execute;
- the process modelling the effect of the transition execute;
- the process modelling the entry behaviour of the target state execute; and
- the target state is entered.

The above is modelled using the sequential composition operator of CSP. 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[2]. 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 a 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*.

```
$BDMotor(queue, local) =$
  let
    $I_0 = OFF$
    $OFF =$
      $local.proc.BDMotorOn?d \rightarrow$
        $TurnOnBDMotor(d) \fatsemi ON$
      $\Box$
      $local.disc?e : \{| BDMotorOff |\} \rightarrow OFF$
    $ON =$
      $local.proc.BDMotorOff?d \rightarrow$
        $TurnOffBDMotor(d) \fatsemi OFF$
      $\Box$
      $local.disc?e : \{| BDMotorOn |\} \rightarrow ON$
    $EQ = queue?e \rightarrow local?p!e \rightarrow EQ$
  within
    $I_0 \; [| \{| local |\} |] \; EQ$

$BDMOTOR = BDMotor(bdmotor, bdmotorlocal)$

$\alpha BDMOTOR =$
  $Union(\{\{| bdmotor, bdmotorlocal |\},$
    $\alpha TurnOnBDMotor, \alpha TurnOffBDMotor\})$

$Magnet(queue, local) =$
  let
    $I_0 = OFF$
    $OFF =$
      $local.proc.MagnetOn \rightarrow$
        $ActivateMagnet \fatsemi ON$
      $\Box$
      $local.disc?e : \{| MagnetOff |\} \rightarrow OFF$
    $ON =$
      $local.proc.MagnetOff \rightarrow$
        $DeactivateMagnet \fatsemi OFF$
      $\Box$
      $local.disc?e : \{| MagnetOn |\} \rightarrow ON$
    $EQ = queue?e \rightarrow local?p!e \rightarrow EQ$
  within
    $I_0 \; [| \{| local |\} |] \; EQ$

$MAGNET = Magnet(magnet, magnetlocal)$

$\alpha MAGNET =$
  $Union(\{\{| magnet, magnetlocal |\},$
    $\alpha ActivateMagnet, \alpha DeactivateMagnet\})$

$PDMeter(queue, local) =$
  let
    $I_0 = IDLE$
    $IDLE =$
      $local.proc.NotifyPD?pd \rightarrow SENSE$
    $SENSE =$
      $OnSense \fatsemi IDLE$
      $\Box$
      $local.disc?e : \{| NotifyPD |\} \rightarrow SENSE$
    $EQ = queue?e \rightarrow local?p!e \rightarrow EQ$
  within
    $I_0 \; [| \{| local |\} |] \; EQ$

$PDMETER = PDMeter(pdmeter, pdmeterlocal)$

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

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

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.

C. Activities

The activities that serve to augment the classifier behaviour of the blocks introduced in Section III-B are formalised here.

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 — TurnOnBDMotor and TurnOffBDMotor — can be modelled thus.

```
$TurnOnBDMotor(d) =$
  let
    $DEC_0 =$
      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
    $DEC_0$

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

$TurnOffBDMotor =$
```

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

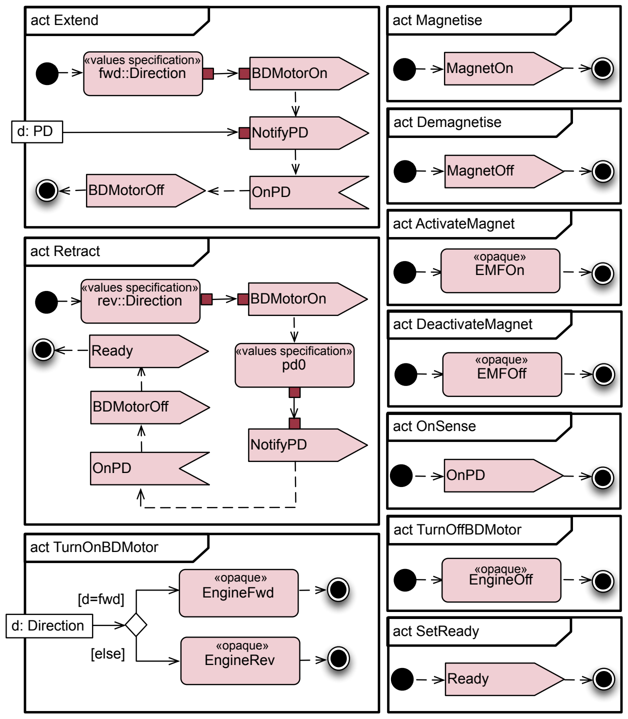

Fig. 3. Activity diagrams modelling additional behaviours executed within the context of state machines.

$$
\begin{aligned}
&\text{let}\\
&\quad I_0 = OA_0\\
&\quad OA_0 = opaque.engineoff \rightarrow F_0\\
&\quad F_0 = Skip\\
&\text{within}\\
&\quad I_0
\end{aligned}
$$

$$\alpha TurnOffBDMotor = \{\mid opaque.engineoff \mid\}$$

The rest of the activities of Figure 3 — ActivateMagnet, DeactivateMagnet and OnSense — can be similarly defined. We omit the definitions here in the interest of brevity.

IV. Interpretations

This section explores the different notions that can be attributed to the behavioural composition of a collection of blocks using a process algebraic approach.

A. Abstraction

We explore the behavioural composition of the Arm block with the first interpretation, mentioned in Section I, in mind.

The block definition diagram showing the composition of the Arm is shown in Figure 4; the interconnection amongst the parts are depicted with the internal block definition diagram of Figure 5. The structural aspects of the system are modelled using blocks for the controller, bidirectional motor, 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

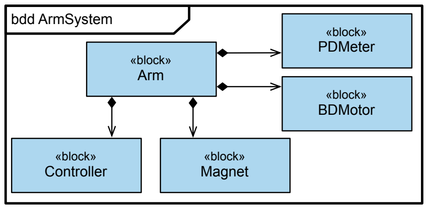

Fig. 4. The block definition diagram of the Arm System.

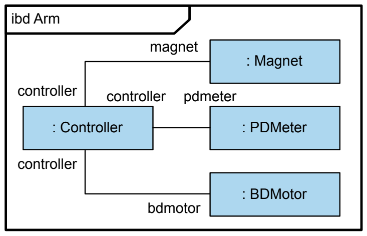

Fig. 5. The internal block definition diagram of the Arm block.

BDMotor, Magnet and PDMeter; the controller is introduced below.

The channel and datatype definitions are similar to those defined in Section III.

$$
\begin{aligned}
&\text{datatype } ArmSignal = PickUp.PD \mid PutDown.PD\\
&\text{datatype } ControllerSignal =\\
&\quad Grasp.PD \mid Drop.PD \mid OnPD\\
&\\
&\text{channel } controller : ControllerSignal\\
&\text{channel } controllerlocal : Dispatched.ControllerSignal
\end{aligned}
$$

The provided and required receptions of the Controller and Arm blocks are shown below[3]. 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 a triggers in their classifying state machines.

The CSP process describing modelling the characteristic behaviour of the controller's state machine 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. Activities are shown in Figure 3.

$$
\begin{aligned}
&Controller(queue, local) =\\
&\quad \text{let}\\
&\quad\quad I_0 = IDLE\\
&\quad\quad IDLE =\\
&\quad\quad\quad local.proc.Grasp?e \rightarrow\\
&\quad\quad\quad\quad Extend(local, e) \fatsemi Magnetise \fatsemi GRASP\\
&\quad\quad\quad \Box\\
&\quad\quad\quad local.proc.Drop?e \rightarrow\\
&\quad\quad\quad\quad Extend(local, e) \fatsemi Demagnetise \fatsemi DROP\\
&\quad\quad\quad \Box
\end{aligned}
$$

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

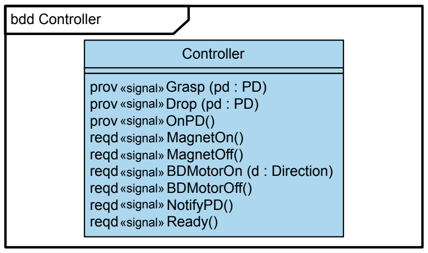
bdd Controller
Controller
prov «signal» Grasp (pd : PD)
prov «signal» Drop (pd : PD)
prov «signal» OnPD()
reqd «signal» MagnetOn()
reqd «signal» MagnetOff()
reqd «signal» BDMotorOn (d : Direction)
reqd «signal» BDMotorOff()
reqd «signal» NotifyPD()
reqd «signal» Ready()

Fig. 6. The block definition diagram of the Controller block.

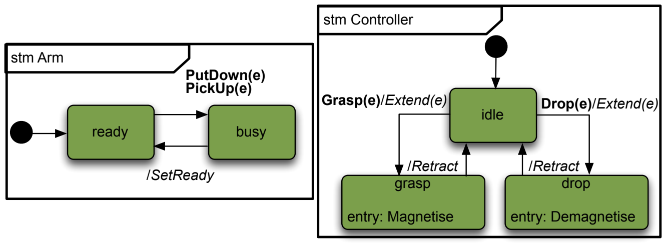

Fig. 7. The state machine diagrams of the classifier behaviours of the Arm and Controller blocks.

```
        $local.disc?e : \{\mid OnPD \mid\} \rightarrow IDLE$
    $GRASP =$
        $Retract(local) \fatsemi IDLE$
        $\Box$
        $local.disc?e : \{\mid Grasp, Drop, OnPD \mid\} \rightarrow GRASP$
    $DROP =$
        $Retract(local) \fatsemi IDLE$
        $\Box$
        $local.disc?e : \{\mid Grasp, Drop, OnPD \mid\} \rightarrow DROP$
    $EQ = queue?e \rightarrow local?p!e \rightarrow EQ$
  within
    $I_0 \; [\![ \{\mid local \mid\} ]\!] \; EQ$
```

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

```
$\alpha CONTROLLER =$
  $Union(\{\{\mid controller, controllerlocal \mid\},$
    $\alpha Magnetise, \alpha Demagnetise, \alpha Extend, \alpha Retract\})$
```

The blocks above — Controller, BDMotor, Magnet and PDMeter are all concrete implementation blocks in SysML. The abstract block, Arm, which serve as an implementation which the parts must realise, is modelled below.

```
$Arm(queue, local) =$
  let
    $I_0 = READY$
    $READY =$
        $local.proc.PickUp?e \rightarrow BUSY$
        $\Box$
        $local.proc.PutDown?e \rightarrow BUSY$
    $BUSY =$
        $SetReady \fatsemi READY$
        $\Box$
        $local.disc?e : \{\mid PickUp, PutDown \mid\} \rightarrow BUSY$
    $EQ = queue?e \rightarrow local?p!e \rightarrow EQ$
  within
    $I_0 \; [\![ \{\mid local \mid\} ]\!] \; EQ$
```

$$ARM = Arm(arm, armlocal)$$

```
$\alpha ARM =$
  $Union(\{\{\mid arm, armlocal \mid\}, \alpha SetReady\})$
```

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
    $I_0$
```

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

The processes — *Extend*, *Retract*, *Magnetise* and *Demagnetise* — modelling the activities used in the *CONTROLLER* process follow. All activities in this paper execute within the context of their owing state machine. An activity can take parameters, passed from the arguments of the triggering event of the owing 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
    $I_0 = VS_0$
    $VS_0 = SS_0(fwd)$
    $SS_0(o) = bdmotor.BDMotorOn.o \rightarrow SS_1$
    $SS_1 = pdmeter.NotifyPD.pd \rightarrow RS_0$
    $RS_0 =$
        $local.proc.OnPD \rightarrow SS_2$
        $\Box$
        $local.disc?ev : \{\mid Grasp, Drop \mid\} \rightarrow RS_0$
    $SS_2 = bdmotor.BDMotorOff \rightarrow F_0$
    $F_0 = Skip$
  within
    $I_0$
```

```
$\alpha Extend =$
  $\{\mid bdmotor.BDMotorOn.fwd, bdmotor.BDMotorOff,$
    $pdmeter.NotifyPD \mid\}$
```

```
$Retract(local) =$
  let
    $I_0 = VS_0$
    $VS_0 = SS_0(rev)$
    $SS_0(o) = bdmotor.BDMotorOn.o \rightarrow VS_1$
    $VS_1 = SS_1(pd_0)$
    $SS_1(o) = pdmeter.NotifyPD.0 \rightarrow RS_0$
    $RS_0 =$
        $local.proc.OnPD \rightarrow SS_2$
        $\Box$
```

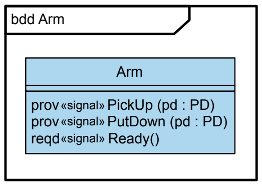

Fig. 8. The block definition diagram of the Arm block.

$$
\begin{array}{l}
\qquad\quad local.disc?ev : \{\!|\, Grasp, Drop \,|\!\} \rightarrow RS_0 \\
\qquad SS_2 = bdmotor.BDMotorOff \rightarrow SS_3 \\
SS_3 = client.Ready \rightarrow F_0 \\
F_0 = Skip \\
\quad \text{within} \\
\qquad I_0
\end{array}
$$

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

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

$$\alpha Magnetise = \{\!|\, magnet.MagnetOn \,|\!\}$$

$$
\begin{array}{l}
Demagnetise = \\
\quad \text{let} \\
\qquad I_0 = SS_0 \\
\qquad SS_0 = magnet.magnetOff \rightarrow F_0 \\
\qquad F_0 = Skip \\
\quad \text{within} \\
\qquad I_0 \\
\alpha Demagnetise = \{\!|\, magnet.MagnetOff \,|\!\}
\end{array}
$$

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[4] 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[5] are sent using the channel with the same name as the target block, similar to the approach taken for state machines[6].

Assuming that

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

we then have

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

[4]As an example, the action outputting the forward direction in Extend.

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

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

In the above, αp 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.

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

The set *Hidden* are 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

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

FDR verifies the assertion

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

Given that the trace refinement holds, *ARM* can be substituted for its parts in the complete system: the behaviour of the concrete implementation processes, denoted 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.

B. Controller

We explore the behavioural composition of the Arm block with the second interpretation, mentioned in Section I, 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.

Figure 9 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 Figure 10: 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, potentiometer and bidirectional motor).

The behaviour of the state machine for the arm in the second interpretation is exactly the behaviour of the controller in

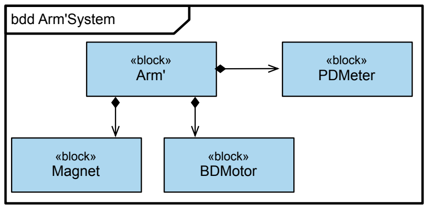

Fig. 9. The block definition diagram of the Arm' System.

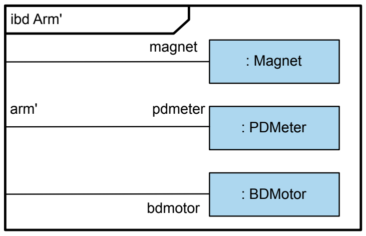

Fig. 10. The internal block definition diagram of the Arm' block.

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
      $I_0 = IDLE$
      $IDLE =$
         $local.proc.Grasp?e \rightarrow$
            $Extend(local, e) ⨾ Magnetise ⨾ GRASP$
         $\square$
         $local.proc.Drop?e \rightarrow$
            $Extend(local, e) ⨾ Demagnetise ⨾ DROP$
         $\square$
         $local.disc?e : \{| OnPD |\} \rightarrow IDLE$
      $GRASP =$
         $Retract(local) ⨾ IDLE$
         $\square$
         $local.disc?e : \{| Grasp, Drop, OnPD |\} \rightarrow GRASP$
      $DROP =$
         $Retract(local) ⨾ IDLE$
         $\square$
         $local.disc?e : \{| Grasp, Drop, OnPD |\} \rightarrow DROP$
      $EQ = queue?e \rightarrow local?p!e \rightarrow EQ$
   within
      $I_0 [| \{| local |\} |] EQ$

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

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

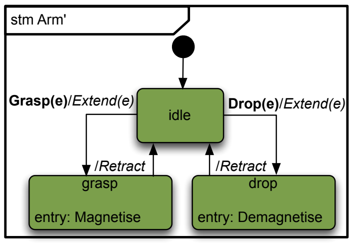

Fig. 11. The state machine diagram of the classifier behaviour of the Arm' block.

The complete behaviour of the arm and all its parts can be expressed in CSP thus. Assuming that

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

we then have

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

In the above, αp denotes the set of events communicable by P. The set of processes P represent the arm, that acts as a controller, and its constituent parts: the magnet, bidirectional motor and potentiometer. The behaviour is a combination of the arm and that of the magnet, bidirectional motor and 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 [6]. Figure 12 graphically 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[7] in the system of interest – the travelling crance — 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 [6].

V. Related Work

In this paper we have defined the semantics of state machines and activities that execute within this context informally. For a formal treatment of the semantics of state machines, see [6]; the formal semantics of activities is forthcoming.

[7]Presuming the refinement holds.

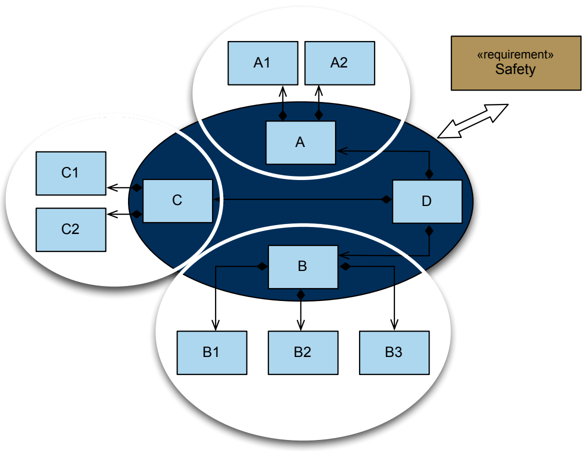

Fig. 12. The compositional approach afforded by CSP. The white ellipses denote the behavioural interpretation of blocks using the abstraction approach. The system of interest, block *D*, is composed of abstract blocks and serves as a controller that orchestrates the behaviour. The second interpretation, shown inside the dark blue 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 A_1 and A_2. 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 [6].

Interactions, within the context of SysML, is formalised in [7].

A formal semantics for some of the SysML diagrams have been given in terms of the *COMPASS Modelling Language* (CML) [8]. A set of translation rules are given that maps SysML diagrams to their counterparts in CML. The work presented here is different in that CML integrates state-based as well as process algebraic description techniques. Our work [9], [6], [7] is solely concerned with defining a process algebraic approach to ensure behavioural conformance amongst the behaviour diagrams of SysML. A formalisation of state machines and activities as used in this context can be found in [10].

Ng and Butler [11] proposed the formalisation of UML state machine diagrams using CSP as the semantic domain [11]. They define the translation in terms of a mapping function 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 and colleagues [12] built 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 [11]. Zang and Liu [13] mapped state machine diagrams to CSP using the model checker PAT [14]. A state machine is modelled by a single CSP process; translation rules are presented that map state machine constructs to CSP# [14], the input language of the *Process Analysis Toolkit* (PAT) [14]. Refinement checking as well as *linear temporal logic* (LTL) [15] 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. [16], [17] 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 [16], [17] 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 the primary focus.

VI. Conclusions

In this paper we demonstrated how the refinement checker FDR can be used in a practical setting to ensure that different behavioural formalisms — activities and state machines — are consistent. Moreover, we have demonstrated how the concept of 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. To the best of our knowledge this is the first contribution that 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 behavioural formalisms explored in this paper — state machines and activities — using CSP.

The contributions of this paper can be summarised thus.

- We presented a formal model of SysML blocks using CSP. In particular, we demonstrated two possible interpretations of SysML blocks for modelling and integrating system behaviour in a formal setting.
- We 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 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 is well suited to underline the principles of systems engineering.

References

[1] *Systems Modeling Language Specification, version 1.3*, Object Management Group, 2012, available at: http://www.omg.org/spec/SysML/1.3, [2014, March].

[2] S. Friedenthal, A. Moore, and R. Steiner, *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann Publishers, 2008.

[3] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.

[4] *Failures Divergences Refinement User Manual, version 2.94*, Department of Computer Science, University of Oxford, 2012, available at: http://www.cs.ox.ac.uk/projects/concurrency-tools/download/fdr2manual-2.94.pdf, [2014, March].

[5] C. Lewerentz and T. Lindner, "Case study Production Cell," in *Formal Development of Reactive Systems*, ser. Lecture Notes in Computer Science. Springer, 1995, vol. 891.

[6] J. Jacobs and A. C. Simpson, "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)*, ser. Lecture Notes in Computer Science. Springer, 2013, vol. 8144, pp. 266–281.

[7] ——, "On a process algebraic representation of sequence diagrams," in *Proceedings of the 1st International Workshop on Safety and Formal Methods (SaFoMe 2014)*, ser. Lecture Notes in Computer Science. Springer, 2014.

[8] J. C. P. Woodcock, A. L. C. Cavalcanti, J. Fitzgerald, P. Larsen, A. Miyazawa, and S. Perry, "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)*. IEEE, 2012, pp. 1–6.

[9] J. Jacobs and A. C. Simpson, "A process algebraic approach to decomposition of communicating SysML blocks," in *Proceedings of the 2nd International Conference on System Engineering and Modeling (ICSEM 2013)*. IACSIT, 2013.

[10] ——, "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)*, ser. Communications in Computer and Information Science. Springer, 2014.

[11] M. Y. Ng and M. Butler, "Towards formalizing UML state diagrams in CSP," in *Proceedings of the 1st International Conference on Software Engineering and Formal Methods (SEFM 2003)*. IEEE, 2003, pp. 138–147.

[12] W. L. Yeung, K. R. P. H. Leung, W. Dong, and J. Wang, "Improvements towards formalizing UML state diagrams in CSP," in *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC 2005)*. IEEE, 2005, pp. 176–182.

[13] S. J. Zhang and Y. Liu, "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)*. IEEE, 2010, pp. 1–6.

[14] J. Sun, Y. Liu, and J. S. Dong, "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)*, ser. Communications in Computer and Information Science. Springer, 2008, vol. 17, pp. 307–322.

[15] C. Baier and J.-P. Katoen, *Principles Of Model Checking*. MIT Press, 2008.

[16] D. Xu, N. Philbert, Z. Liu, and W. Liu, "Towards formalizing UML activity diagrams in CSP," in *Proceedings of the 2008 International Symposium on Computer Science and Computational Technology (ISCSCT 2008)*. IEEE, 2008, pp. 450–453.

[17] D. Xu, H. Miao, and N. Philbert, "Model checking UML activity diagrams in FDR," in *Proceedings of the 8th International Conference on Computer and Information Science (ICIS 2009)*. IEEE, 2009, pp. 1035–1040.