Towards a Process Algebra Framework for Supporting Behavioural Consistency and Requirements Traceability in SysML

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), an extension of a subset of the Unified Modeling Language (UML), is a visual modelling language for systems engineering applications. At present, the semi-formal SysML, which is widely utilised for the design of complex heterogeneous systems, lacks integration with other more formal approaches. In this paper, we describe how Communicating Sequential Processes (CSP) and its associated refinement checker, Failures Divergence Refinement (FDR), may be used to underpin an approach that facilitates the refinement checking of behavioural consistency of SysML diagrams; we also show how the proposed approach supports requirements traceability. We illustrate our contribution by means of a small case study.

Keywords: CSP, SysML, Requirements Traceability

1 Introduction

State-of-the-art systems are typically organic, multi-disciplinary compositions of interconnecting components or systems, functioning as a whole in order to achieve a shared goal.

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.

One of the biggest challenges in the specification and design of a complex system, potentially composed of several subsidiary systems, is to ensure that the proposed design is logically consistent. There are, of course, various aspects to take into account when looking at the notion of consistency. For example, it is possible to consider whether the non-behavioural aspects — embodied by the structural constructs of the modelling language — are specified such that they are logically sound, and that the constraints imposed upon them, hold. It

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

is also possible to consider more closely the behavioural aspects of a proposed design, and this is our focus in this paper. For this purpose, we utilise Communicating Sequential Processes (CSP) [2, 3]. In contrast to SysML, which one might categorise as a semi-formal notation, CSP has a rigorous mathematical underpinning with an elegant means of specifying and reasoning about complex behaviour. Moreover, the Failures Divergence Refinement (FDR) [4] tool — a refinement checker for CSP — readily allows for the reasoning about correctness of designs and verification that asserted properties hold. Specifically, by translating SysML into CSP we can precisely define the intended behaviour of a given SysML model, by making use of the underlying formal semantics of CSP. Consequently, this allows refinement checking of the SysML model. However, when attempting to formalise behaviour, we still need to consider structural aspects: if we are to define a sensible behavioural semantics, we cannot be agnostic with respect to the static composition of the SysML model. Moreover, the resulting CSP process network and its associated communication configuration need to reflect the structural specification of the model. It is worth noting at this point that the intention here is not to propose a means of replacing SysML modelling tools, but, rather, to develop a formal framework that can be used in conjunction with SysML, with a view to complementing the modelling activity. The subsequent integration of graphical and formal notations would have the potential to allow the modeller to reap the benefits of both methods.

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. Process algebras, like CSP, can help in this respect.

Requirements traceability plays an important role as part of any model-based systems engineering methodology. In SysML, requirements can be related to other requirements, as well as to other model elements via one or more relationships: a behavioural construct can be allocated to a particular requirement, and we can subsequently use FDR to ensure that the model satisfies it.

In Section 2, we define the necessary mathematical structures, based on the syntax of SysML, required to formulate behavioural CSP descriptions. (We assume that the reader is familiar with CSP.) Section 3 presents a bird's eye view of how CSP can be used in combination with SysML in order to create well-formed models and ensure that requirements are satisfied. To this end, we provide a formal semantics for state machines, within the context of SysML. In Section 4, we introduce a small case study which we use as a means of illustrating and validating the contribution. Finally, Section 5 summarises the contribution of this paper, places it in context with respect to other research, and outlines possible avenues of further work.

2 Syntactical structures

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. Our purpose is not to formulate a complete syntactical specification of the considered constructs, but rather to employ these expositions to assist us in defining the formal behavioural semantics in a comprehensible and sympathetic fashion.

2.1 Model and signals

Let the SysML model be a quintuple $(\mathcal{B}, \mathcal{P}, \mathcal{M}, \mathcal{R}, \mathcal{S})$, where $\mathcal{B}$ is the set of blocks, $\mathcal{P}$ is the set of parts, $\mathcal{M}$ is the set of state machines, and $\mathcal{R}$ is the set containing all requirements. $\mathcal{S}$ contains the set of all signals; these are used as a means of communication between state machines.

A *signal* $S_i \in \mathcal{S}$ is uniquely identified by a name, $N_{S_i} \in \mathcal{N}_{\mathcal{S}}$, and contains a sequence of parameters. Let $\mathcal{N}_{\mathcal{S}}$ contain the names of all signals. The function $name : \mathcal{S} \to \mathcal{N}_{\mathcal{S}}$ returns the unique name of a signal. The sequence of parameters P_{S_i} of a signal S_i is given by $params(S_i) = P_{S_i}$.

2.2 Blocks and parts

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 (for reactive, event-driven blocks) or an activity (for transformational blocks that take inputs to outputs via a sequence of actions).

A block $B_i \in \mathcal{B}$ 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 *operations*, for method calls, or *receptions*, for signals, 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*. In this paper we consider only asynchronous communication using signal events, therefore all behavioural features are receptions. We define the functions $prov : \mathcal{B} \to \mathbb{P}\,\mathcal{S}$ and $reqd : \mathcal{B} \to \mathbb{P}\,\mathcal{S}$ to return provided and required receptions.

The *internal block diagram*, as per Figure 1, graphically sets out the internal structure of a block from its parts. In contrast, a *block definition diagram*, also presented in Figure 1, 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

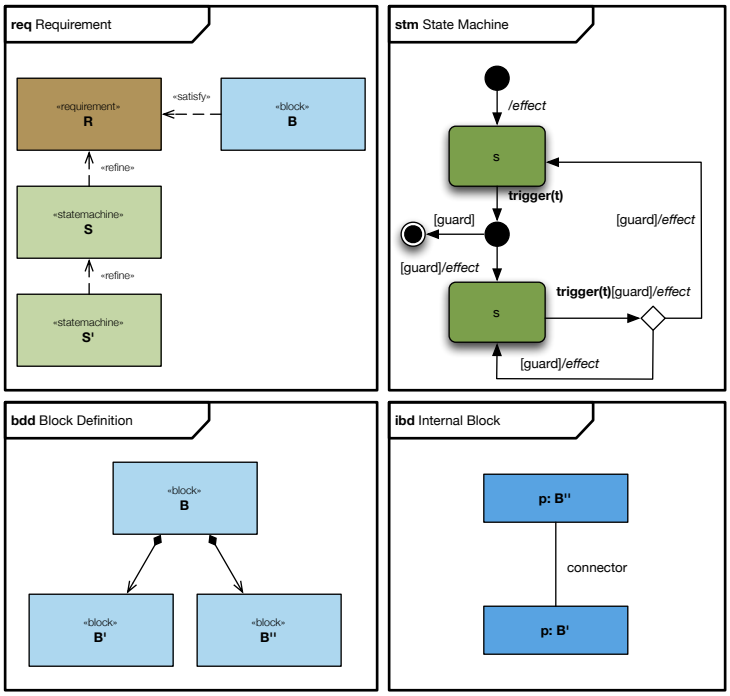

Fig. 1. Examples of a *requirement*, a *state machine*, a *block definition*, and an *internal block diagram*. Block definition and internal block diagrams are used to model static, structural aspects. In contrast, the state machine diagram models dynamic behaviour. Requirements and their relationship to other modelling constructs are shown on a requirement diagram.

classifying block within the context of its owning block. Each part $P_i \in \mathcal{P}$ is typed by a block $B_j \in \mathcal{B}$; 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 send between communicating block instances.

2.3 State machines

We now define a structural model for non-hierarchical state machines, our main concern in this paper. Figure 1 presents an example of a *state machine diagram* with the modelling constructs we consider here.

A *state machine* $M_i \in \mathcal{M}$ consists of a finite set of *states*, denoted S_{M_i}, and *transitions* between those states, denoted T_{M_i}. We partition S_{M_i} such that $S^I_{M_i}$ represents the set of *initial states*, $S^F_{M_i}$ the set of *final states*, $S^S_{M_i}$ the set of *simple states*, $S^J_{M_i}$ the set of *junction states*, and finally $S^C_{M_i}$ the set of *choice states*. Assume that the aforementioned sets are mutually disjoint: we have $\forall\, s_j, s_k : \{S^I_{M_i}, S^F_{M_i}, S^S_{M_i}, S^J_{M_i}, S^C_{M_i}\} \bullet s_j \neq s_k \Rightarrow s_j \cap s_k = \emptyset$. Furthermore, assume that

$S^I_{M_i}$ is the singleton set, that is, there is a unique initial state. In addition, we assume that a final state is optional: $S^F_{M_i}$ can be the empty set.

Refer to Figure 1. Diagrammatically, initial and final states are indicated by a solid circle and encircled solid circle, respectively, whereas junction and choice states are indicated using a solid circle and diamond shaped node. When labelling transitions, we use the following convention: **trigger**[guard]/*effect*.

A function $outgoing : S_{M_i} \to \mathbb{P}\, T_{M_i}$ returns the set of outgoing transitions for a state. In order to make the formalisation easier, assume the existence of $outgoing'$ and $outgoing''$, where the former returns, for a particular state, all guarded outgoing transitions, and the latter returns all those that are not guarded. Thus,

$$\begin{aligned}\forall\, s : S_{M_i} \bullet\ & outgoing(s) = outgoing'(s) \cup outgoing''(s) \\ & \land \\ & outgoing'(s) \cap outgoing''(s) = \emptyset\end{aligned}$$

A transition consists of a *trigger*, a *guard*, and an *effect*. Additionally, a transition is defined to exist between a source and target state. We define the following functions, to return for a transition t:

- the source state, given by $source : T_{M_i} \to S_{M_i}$;
- the target state, given by $target : T_{M_i} \to S_{M_i}$;
- the trigger, given by $trigger : T_{M_i} \to \mathcal{S}$;
- the guard, given by $guard : T_{M_i} \to E_{M_i}$; and
- the effect, given by $effect : T_{M_i} \to \mathrm{seq}\,\mathcal{S}$.

In the above, E_{M_i} represents a set of expressions. Let $eval(e, p)$ represent the evaluation of an expression e by considering the values of the instantiated parameter sequence p, returning a value from the set $\mathbb{B} = \{true, false\}$. Specifically, for a transition t, we substitute[2] the parameters passed on the receive signal event, $params(trigger(t))$, in the guard expression, $guard(t) \in E_{M_i}$. We write $eval(guard(t), params(trigger(t)))$ to denote this.

One can take an alternative stance with respect to the guards of transitions. Specifically, the CSP non-deterministic choice operator can be used instead of conditional choice. We opt for conditional choice here, as we assume the parameters of the receive signal event to be of relevance when evaluating the conditions.

We distinguish between two types of transitions: *simple transitions* and *complex transitions* via junction and choice pseudo states, respectively. A simple transition exists between two simple states. A complex transition is treated as two separate transitions between a simple state and a pseudo state, and a pseudo state and a simple state, where the pseudo state is either a junction or choice state. Thus it allows for the specification of multiple paths between states, although only a single path can be taken in response to any event. Again, Figure 1 presents an example of this.

[2] In SysML, parameters are matched based on their names and types.

3 A CSP view of SysML

This section outlines an approach to integrate the semi-formal SysML notation with the process algebra CSP. Throughout, we make use of the structures defined in Section 2. First, we provide a formal process semantics for a set of communicating SysML blocks — a central aspect to any SysML model. We then consider how CSP and refinement can be utilised to support requirements traceability — a core SysML concept supported via the requirement diagram.

3.1 Signals and events

In CSP, for a signal S_i, the name of the signal $name(S_i)$ is used as a component of the CSP event. In our formalisation, an effect corresponds to a sequence of *send signal events*, whereas a trigger corresponds to a *receive signal event*. We write $snd(S_i)$ for a send signal event, where the parameters of the event are communicated as outputs on the corresponding CSP channel. Conversely, for a receive signal event, written $rcv(S_i)$ the parameters are modelled as inputs. The signal types the corresponding send and receive signal events and this is reflected in the CSP channel definition. For a signal S_i, with parameter component given by $params(S_i) = \langle p_0, p_1, .., p_n \rangle$, we have (for a CSP channel c)

- $snd(S_i) = c.name(S_i)!p_0!p_1...!p_n$
- $rcv(S_i) = c.name(S_i)?p_0?p_1...?p_n$

For the base case where $params(S_i) = \langle\rangle$, the corresponding input and output components are simply omitted.

3.2 State machines

Consider a state machine M_i. We provide mapping rules, starting from an initial state, using a translation function $\mathcal{T}$. Every rule, $\mathcal{T}(m, s)$, is defined such that it describes the behaviour of state machine m at state s. These rules define local process definitions, where each state and pseudo state is represented by a CSP process. This approach is similar to that taken by Ng and Butler [5].

Initial state. We start with the unique initial state $s_0 \in S^I_{M_i}$. In order for the state machine to be well-defined, the initial state must have a single outgoing transition that defines its unique starting point. The lone outgoing transition, $t \in outgoing(s_0)$, may optionally include a sequence of effects.

$$\mathcal{T}(M_i, s_0) = \mathit{Effect}(effect(t)) \fatsemi \mathcal{T}(M_i, target(t))$$

In the above, the CSP process *Effect* takes a sequence of events and communicates them in order before successfully terminating.

$$
\begin{array}{l}
\mathit{Effect}(s) = \\
\quad \text{if } null(s) \text{ then} \\
\quad\quad \mathit{Skip} \\
\quad \text{else} \\
\quad\quad snd(head(s)) \rightarrow \mathit{Effect}(tail(s))
\end{array}
$$

Simple state. Consider a simple state $s \in S^S_{M_i}$. We define the following functions:

- $\forall\, s : S^S_{M_i} \bullet \mathit{simple}'(s) = \{t : \mathit{outgoing}'(s) \mid \mathit{target}(t) \in S^S_{M_i}\}$, that returns guarded transitions to simple states;
- $\forall\, s : S^S_{M_i} \bullet \mathit{simple}''(s) = \{t : \mathit{outgoing}''(s) \mid \mathit{target}(t) \in S^S_{M_i}\}$, that returns transitions with no guard condition that lead to simple states;
- $\forall\, s : S^S_{M_i} \bullet \mathit{junction}'(s) = \{t : \mathit{outgoing}'(s) \mid \mathit{target}(t) \in S^J_{M_i}\}$, that returns guarded transitions to junction states;
- $\forall\, s : S^S_{M_i} \bullet \mathit{junction}''(s) = \{t : \mathit{outgoing}''(s) \mid \mathit{target}(t) \in S^J_{M_i}\}$, that returns transitions with no guard condition that lead to junction states;
- $\forall\, s : S^S_{M_i} \bullet \mathit{choice}'(s) = \{t : \mathit{outgoing}'(s) \mid \mathit{target}(t) \in S^C_{M_i}\}$, that returns guarded transitions to choice states;
- $\forall\, s : S^S_{M_i} \bullet \mathit{choice}''(s) = \{t : \mathit{outgoing}''(s) \mid \mathit{target}(t) \in S^C_{M_i}\}$, that returns transitions with no guard condition that lead to choice states.

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. 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 effects are executed in order before behaving as the process associated with the destination state. Recall that in our formalisation, an effect corresponds to the sending of a series of send signal events, in the order prescribed by the sequence modelling the effect. In addition, we need to consider the eventuality where the state machine receives a signal event not expected in the current state s: that is, an instance of a signal event S_j such that $S_j \notin \{t : \mathit{outgoing}(s) \bullet \mathit{trigger}(t)\}$. 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).

Junction or choice states are modelled as a parametrised CSP processes: we assume that the data component, i.e. the parameters, of the receive signal event that served as the trigger, will be used in the guard of the next leg of the compound transition. A choice state is distinct from a junction state in that a junction state only allows for a trigger and optional guard on the first leg of the compound transition, whereas a choice state allows a trigger, guard and effect.

The CSP channel *in* is used for communicating with the event queue of the state machine M_i.

$$
\begin{array}{l}
\mathcal{T}(M_i, s) = \\
\quad \Box\, t : \mathit{simple}'(s) \bullet \mathit{in.rcv}(\mathit{trigger}(t)) \rightarrow \\
\qquad (\text{if } \mathit{eval}(\mathit{guard}(t), \mathit{params}(\mathit{trigger}(t))) \text{ then} \\
\qquad\quad \mathit{Effect}(\mathit{effect}(t)) \fatsemi \mathcal{T}(M_i, \mathit{target}(t)) \\
\qquad \text{else} \\
\qquad\quad \mathcal{T}(M_i, s)) \\
\quad \Box \\
\quad \Box\, t : \mathit{simple}''(s) \bullet \mathit{in.rcv}(\mathit{trigger}(t)) \rightarrow \\
\qquad \mathit{Effect}(\mathit{effect}(t)) \fatsemi \mathcal{T}(M_i, \mathit{target}(t))
\end{array}
$$

$$
\begin{array}{l}
\Box \\
\Box\ t : \mathit{junction}'(s) \bullet \mathit{in.rcv}(\mathit{trigger}(t)) \rightarrow \\
\quad (\text{if } \mathit{eval}(\mathit{guard}(t), \mathit{params}(\mathit{trigger}(t))) \text{ then} \\
\quad\quad \mathcal{T}(M_i, \mathit{target}(t))[\mathit{params}(\mathit{trigger}(t))] \\
\quad \text{else} \\
\quad\quad \mathcal{T}(M_i, s)) \\
\Box \\
\Box\ t : \mathit{junction}''(s) \bullet \mathit{in.rcv}(\mathit{trigger}(t)) \rightarrow \\
\quad \mathcal{T}(M_i, \mathit{target}(t))[\mathit{params}(\mathit{trigger}(t))] \\
\Box \\
\Box\ t : \mathit{choice}'(s) \bullet \mathit{in.rcv}(\mathit{trigger}(t)) \rightarrow \\
\quad (\text{if } \mathit{eval}(\mathit{guard}(t), \mathit{params}(\mathit{trigger}(t))) \text{ then} \\
\quad\quad \mathit{Effect}(\mathit{effect}(t)) \fatsemi \mathcal{T}(M_i, \mathit{target}(t))[\mathit{params}(\mathit{trigger}(t))] \\
\quad \text{else} \\
\quad\quad \mathcal{T}(M_i, s)) \\
\Box \\
\Box\ t : \mathit{choice}''(s) \bullet \mathit{in.rcv}(\mathit{trigger}(t)) \rightarrow \\
\quad \mathit{Effect}(\mathit{effect}(t)) \fatsemi \mathcal{T}(M_i, \mathit{target}(t))[\mathit{params}(\mathit{trigger}(t))] \\
\Box \\
\Box\ t : \mathit{unexpected}(s) \bullet \mathit{in.rcv}(\mathit{trigger}(t)) \rightarrow \mathcal{T}(M_i, s)
\end{array}
$$

Junction or choice state. Consider a junction state $s \in S^J_{M_i}$, or alternatively a choice state $s \in S^C_{M_i}$. Furthermore, assume a set of outgoing transitions such that $\forall\, t : \mathit{outgoing}(s) \bullet \mathit{target}(t) \in S^S_{M_i}$. The second leg of the complex transition (emanating from the choice or junction state) consists of a guard and optional sequence of effects. In order for the state machine to be well-defined, we assume that all guards must be mutually exclusive and that one of the guards always evaluates to true. The assumption here sits well with the notion that a state machine cannot stay indefinitely within a pseudo state and that it is merely a temporary point along a transition, designed to determine the next simple state. As such, one of the available transitions must be selected based on the guards, and the subsequent effects executed. In addition, note that there are no triggering events possible in a junction or choice state, because the triggering event is assumed to have occurred on the previous leg of the compound transition.

$$
\begin{array}{l}
\mathcal{T}(M_i, s)[\mathit{params}(\mathit{trigger}('t))]^3 = \\
\quad \Box\ t : \mathit{outgoing}'(s) \bullet \\
\quad\quad (\text{if } \mathit{eval}(\mathit{guard}(t), \mathit{params}(\mathit{trigger}('t))) \text{ then} \\
\quad\quad\quad \mathit{Effect}(\mathit{effect}(t)) \fatsemi \mathcal{T}(M_i, \mathit{target}(t)) \\
\quad\quad \text{else} \\
\quad\quad\quad \mathit{Stop})
\end{array}
$$

[3] Formal parameters corresponding to the signal (parameter component thereof) that typed the send signal event that served as the trigger. The transition $'t$ is that of the first leg of the compound transition.

SysML allows a junction or choice state to have multiple incoming transitions, but these can always be recast into separate pseudo states, each with a single incoming transition. As such, our formalisation assumes a single incoming transition per pseudo state, as the definition of parametrised process is dependent on the triggering component of the transition.

Final state. Consider a final state $s_f \in S^F_{M_i}$. A final state has no outgoing transitions and is trivially modelled as the deadlocked process.

$$\mathcal{T}(M_i, s_f) = Stop$$

Process. 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 [6]. 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, *in*. Each of the localised processes have access to this channel in order to receive communications from the event queue. The overall process $\mathcal{T}(M_i)(queue, in)$ initially behaves as the process associated with the initial state $\mathcal{T}(M_i, s_0)$. The local process EQ models the event queue. Here, we assume a queue with a maximum capacity of 1; the queue blocks when full. Non-blocking 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 FDR.

$$\begin{array}{l}\mathcal{T}(M_i)(queue, in) = \\ \quad \mathsf{let} \\ \quad\quad \mathcal{T}(M_i, s_0) = \ldots \\ \quad\quad \vdots \\ \quad\quad \mathcal{T}(M_i, s_f) = Stop \\ \quad\quad EQ = queue?a \rightarrow in.a \rightarrow EQ \\ \quad \mathsf{within} \\ \quad\quad \mathcal{T}(M_i, s_0) \; [|\, \{|\, in \,|\} \,|] \; EQ\end{array}$$

The state machine of a block B_i only receives (through its event queue) the provided receptions, $prov(B_i)$. The required features, $reqd(B_i)$, are communicated across the connectors linking parts. In our formalisation, the name of the part is used as the channel name. For example, if another part P_i provides a feature S_j that a part P_k requires, the state machine of P_k uses $name(P_i).snd(S_j)$ to model the event.

3.3 Blocks and parts

The structure of the SysML model is described by a block $B_i \in \mathcal{B}$, composed from N constituent block instances, the parts $\{P_0 \,..\, P_{N-1}\} \subseteq \mathcal{P}$. The classifier behaviour of each part P_j is modelled via a state machine M_j, given by $classifier(type(P_j))$.

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

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

3.4 Requirements

A *requirement* $R_i \in \mathcal{R}$ is a SysML-specific modelling construct represented explicitly in the syntax of the language via the *requirement diagram*. Requirements, in their most basic form, are typically text-based, and allow for the description of conditions that must be satisfied by a particular system. Requirements can be related to other requirements and to modelling constructs via several relationships. For the purposes of this paper, we concern ourselves with the *satisfy* and *refine* relationship, which are defined as follows.

> "The satisfy relationship describes how a design or implementation model satisfies one or more requirements" [1]

This relationship is used to state that a particular model element meets the associated requirement. This is merely an assertion and not a proof of fact.

> "The refine requirement relationship can be used to describe how a model element or set of elements can be used to further refine a requirement" [1]

A text-based requirement can be captured in a SysML model, with the danger being that a textual description can often be ambiguous. Furthermore, such a description is, in general (and for obvious reasons), not well-suited to automated reasoning. A more precise definition is possible in SysML, which allows any behavioural formalism — for example, a state machine — to be assigned to a requirement using the *refine* relationship. This results in a more formal representation of the corresponding desired behaviour. If this behavioural requirement is subsequently mapped to CSP — as a corresponding characteristic process — we can utilise the refinement checker to assist in reasoning about whether this refinement holds for a given model. The characteristic process is therefore a CSP process that describes the patterns of behaviour of an associated textual requirement.

Consider Figure 1. Here, we assume that B, S and S' all denote behavioural constructs of SysML, and R represents a text-based requirement. In the case of a static modelling construct like a block, we assume the corresponding classifier behaviour. We base our treatment of the UML satisfy and refine relationships on that of the satisfaction and refinement relations of [3]. In the following, $CHAR_R$ denotes the characteristic process of a textual requirement R, and B, S, and S' are the CSP processes for B, S and S', respectively. For the satisfy relationship between a behavioural formalism B and a textual requirement R, we define:

$$\mathsf{B}\ \textsf{satisfy}\ \mathsf{R} \Leftrightarrow CHAR_R \sqsubseteq B$$

The definition of the refine relationship is dependent on the model elements involved in the relationship. In the case where the relationship is between two behavioural formalisms S and S', we define:

$$\mathsf{S}' \text{ refine } \mathsf{S} \Leftrightarrow S \sqsubseteq S'$$

Alternatively, in the case where the refine relationship is expressed between a behavioural formalism S and a textual requirement R, we define the corresponding process S of the behavioural formalism S to be the characteristic process $CHAR_R$ of R:

$$\mathsf{S} \text{ refine } \mathsf{R} \Leftrightarrow CHAR_R = S$$

Therefore, using the definitions above (and substitution) we can check whether S holds in B by executing a refinement check (where the set *Hidden* consists of those events not present in S):

$$S \sqsubseteq B \setminus Hidden$$

If the refinement does not hold, FDR generates a *counterexample* that demonstrates where the behaviour of B deviates from S, prompting the designer to correct the design. Furthermore, we can check the refinement relation between S and S':

$$S \sqsubseteq S' \setminus Hidden$$

These successive transformations (assuming the refinements hold) are behavioural formalisms that we can think of as getting more specific as we move along the refinement chain: S' is more refined than S, which means that (if we only consider traces) $traces[\![S']\!] \subseteq traces[\![S]\!]$ due to the reverse inclusion characteristic of refinement. The direction of the arrowhead in the requirement diagram, as per Figure 1, and its treatment in CSP is also of significance. In UML, the direction of the arrow is from the dependent model element to the independent model element. The state machine B is dependent on the requirement R: any change in R could impact B; CSP refinement honours this dependency.

The Requirement diagram is introduced in SysML and is a core part of the language. Requirements traceability is undoubtedly a good thing; a formal representation of requirement diagrams in CSP gives rise to a formal means of requirements traceability. This provides a formal foundation for this otherwise informal technique that serves as justification for the development of a formal framework. It should be noted that the above approach hinges on the characteristic process being specified correctly.

4 Case study: an access control system

Our case study is a small control component for a physical access control system that can be used, for example, to manage admittance to a toll road. Each authorised user is assigned a *tag*; in order to gain access, the tag is placed on a *reader*. If the tag is valid, a *barrier* moves to the open position; otherwise, it remains closed.

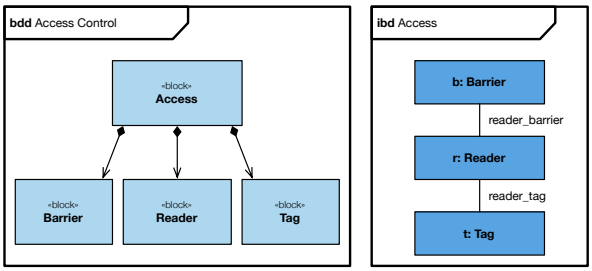

Fig. 2. The block definition diagram modelling the case study as an Access block, composed of aggregate blocks Barrier, Reader and Tag. The internal block diagram of Access shows the internal decomposition into constituent parts, with connectors indicating the links between block instances b, r and t.

4.1 Behavioural consistency of communicating blocks

We now describe how state-based behaviour of blocks may be modelled in CSP, using the formalisms and translation approach defined in Section 2. We also consider how their combined behaviour may be verified using FDR.

The block Access is decomposed into constituent parts: the Barrier, Reader, and Tag block instances (see Figure 2). These parts need to communicate in an orchestrated fashion so as to incorporate the desired behaviour of the composing block, Access. Figure 2 shows a snapshot of the structural constructs used in the design: an instance of the block Reader, r, communicates with instances of the blocks Tag, t, and Barrier, b, using events, typed by signals, via the various connectors. The classifier behaviour of each block is modelled using a corresponding state machine, as per Figure 3.

$$
\begin{array}{l}
\mathsf{datatype}\ BarrierSignal = open \mid close \\
\mathsf{channel}\ barrier, blocal : BarrierSignal \\
\\
Barrier(queue, in) = \\
\quad \mathsf{let} \\
\qquad INIT = CLOSE \\
\qquad CLOSE = \\
\qquad\quad in.open \rightarrow OPEN \\
\qquad\quad \Box \\
\qquad\quad in?discard : \{close\} \rightarrow CLOSE \\
\qquad OPEN = \\
\qquad\quad in.close \rightarrow CLOSE \\
\qquad\quad \Box \\
\qquad\quad in?discard : \{open\} \rightarrow OPEN \\
\qquad EQ = queue?a \rightarrow in.a \rightarrow EQ \\
\quad \mathsf{within} \\
\qquad INIT\ [|\ \{|\ in\ |\}\ |]\ EQ
\end{array}
$$

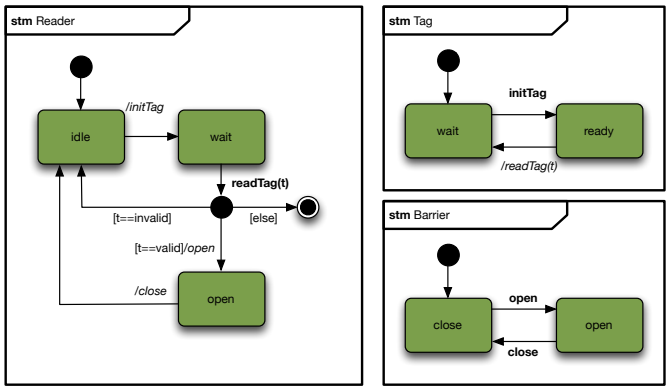

Fig. 3. The state machines modelling the classifier behaviour of the Barrier, Reader and Tag blocks.

$$BARRIER = Barrier(barrier, blocal)$$
$$\alpha BARRIER = \{| \ barrier, blocal \ |\}$$

The definition of process *Barrier*, the CSP counterpart of the classifier behaviour of Barrier, is presented above. A datatype definition is used to type the provided receptions of the Barrier block; these serve as triggers for the classifying state machine.

$$\begin{array}{l}
\mathsf{datatype}\ TagValue = valid \mid invalid \\
\mathsf{datatype}\ ReaderSignal = readTag.TagValue \\
\mathsf{channel}\ reader, rlocal : ReaderSignal \\
\\
Reader(queue, in) = \\
\quad \mathsf{let} \\
\qquad INIT = IDLE \\
\qquad IDLE = tag.initTag \rightarrow WAIT \\
\qquad WAIT = in.readTag?t \rightarrow JUNCTION(t) \\
\qquad JUNCTION(t) = \\
\qquad\quad \text{if } (t == valid) \text{ then } barrier.open \rightarrow OPEN \\
\qquad\quad \text{elseif } (t == invalid) \text{ then } IDLE \\
\qquad\quad \text{else } ERROR \\
\qquad OPEN = barrier.close \rightarrow IDLE \\
\qquad ERROR = STOP \\
\qquad EQ = queue?a \rightarrow in.a \rightarrow EQ \\
\quad \mathsf{within} \\
\qquad INIT \ [| \ \{| \ in \ |\} \ |] \ EQ \\
\\
READER = Reader(reader, rlocal)
\end{array}$$

$$\alpha READER = \{| \ reader, rlocal, tag.initTag, barrier.close, barrier.open \ |\}$$

Similarly, the process *Reader* models the classifier behaviour of the Reader block. In this case, an additional datatype definition is used to model the parameters passed with the signal. Definitions for Tag, TAG and αTAG are omitted: these are defined in a similar fashion to the above.

We use parallel composition to construct the system in terms of its constituent processes. The process *Access* is defined thus:

$$ACCESS = \|(p, a) : Parts \bullet [a]p$$

The set *Parts*, containing (*process*, *alphabet*) pairs, is given by:

$$Parts = \{(BARRIER, \alpha Barrier), (READER, \alpha Reader), (TAG, \alpha Tag)\}$$

These recursive decompositions sit extremely well with the process algebraic formalism CSP. Moreover, by utilising CSP in conjunction with SysML, we are able to describe, and reason about, complex patterns of interaction to an extent not currently possible using just SysML.

4.2 Requirements traceability and refinement

A text-based requirement can be formalised by allocating another SysML model element to the requirement in order to clarify its description and limit ambiguity. In addition, we can test whether a design satisfies the requirement. As an example, consider the following text-based requirement:

"The barrier will only open if a valid tag is presented"

This requirement can be refined by a behavioural modelling construct, e.g., a state machine, as shown in Figure 4, that enriches the textual description.

The associated process, *AccessRequirement*, is defined thus:

$AccessRequirement =$
  let
    $INIT = STATE$
    $STATE = reader.readTag?t \rightarrow JUNCTION(t)$
    $JUNCTION(t) =$
      if $(t == valid)$ then $barrier.open \rightarrow STATE$
      elseif $(t == invalid)$ then $STATE$
  within
    $INIT$

Using the hiding operator and refinement we can test if the design satisfies this requirement, which FDR confirms:

$$AccessRequirement \sqsubseteq_T ACCESS \setminus (\Sigma \setminus NotHidden) \qquad \text{[refinement holds]}$$

In the above, Σ denotes the set of all CSP events within the context of the specification. The set *NotHidden* is given by

$$NotHidden = \{reader.readTag.valid, reader.readTag.invalid, barrier.open\}$$

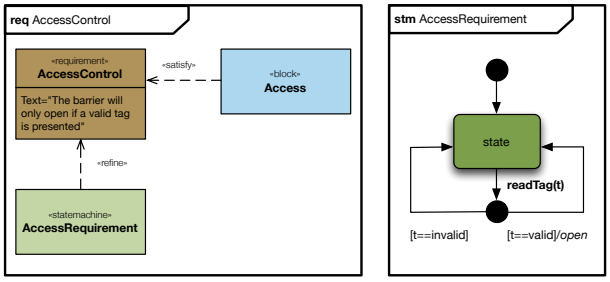

Fig. 4. The state machine AccessRequirement refining the textual description of the requirement AccessControl, which is satisfied by the Access block.

5 Conclusions

SysML is a standard that has only recently matured, and, as such, efforts to integrate it with formal approaches are ongoing. Jacobs and Simpson [10] presented an approach to decompose an abstract SysML block into constituent, concrete blocks. Activity diagrams [7] and state machine diagrams [5] have both been given a formal semantics (considering a subset of the available modelling constructs) in CSP. Abdelhalim *et al.* presented an approach based on checking behavioural consistency between a UML state machine diagram and a corresponding fUML activity diagram [8]. The diagrams are given a formal semantics in CSP; FDR is then employed in order to ensure that a trace refinement holds between the abstract state machine diagram and the concrete activity diagram. Other noteworthy contributions include that of Davies and Crichton [6], which provides a behavioural semantics to combinations of class, object, and state machine diagrams using CSP. Behavioural conformance is formalised within the context of traces and failures refinement of CSP. As another example, Graves and Bijan integrated SysML with a higher order type theory logic in an incremental, top-down approach that aims to maintain the logical consistency of the design, with a case study from the aerospace industry [9].

We have presented a bird's eye view of how CSP and the associated refinement checker FDR can be used in conjunction with a model-based systems engineering approach to systems modelling. Our focus is on the high level integration of blocks, where the behavioural characteristics are modelled using SysML state machines. We cannot view these in isolation, but need to consider them in their context of use as to ensure that the complete system functions as intended. We showed how refinement can be utilised to facilitate requirements traceability and bestow a sense of confidence in the validity of the design.

The choice of CSP is due to a number of factors. First, 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 relatively straightforwardedly. Second, CSP's approach to process composition, combined with the fact that refinement is preserved within

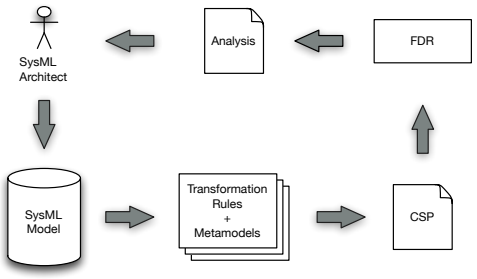

Fig. 5. High level overview of the approach.

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.

Future avenues of research will include the implementation of a formal refinement framework, using a model based approach similar to [8], where SysML and CSP are each defined using a meta-model, and transformations are subsequently used in order to translate from SysML to CSP, as per Figure 5. The resulting framework will then be used to establish whether the design is logically consistent, and if requirements are satisfied by the proposed design.

References

1. Object Management Group: Systems Modelling Language Specification, version 1.3. (2012)
2. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall (1985)
3. Roscoe, A.W.: Understanding Concurrent Systems. Springer (2010)
4. Department of Computer Science, University of Oxford & Formal Systems Europe: Failures Divergence Refinement User Manual, version 2.94. (2012)
5. 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
6. Davies, J.W.M., Crichton, C.R.: Concurrency and refinement in the Unified Modelling Language. Electronic Notes in Theoretical Computer Science **70**(3) (2002) 217–243
7. Dong, X., Philbert, N., Zongtian, L., Wei, L.: Towards formalizing UML activity diagrams in CSP. In: Proceedings of the International Symposium on Computer Science and Computational Technology (ISCSCT 2008), IEEE (2008) 450–453
8. Abdelhalim, I., Schneider, S.A., Treharne, H.: Towards a practical approach to check UML/fUML models consistency using CSP. In: Proceedings of the 13th International Conference on Formal Engineering Methods (ICFEM 2011). Volume 6991 of Lecture Notes in Computer Science. Springer (2011) 33–48
9. Graves, H., Bijan, Y.: Using formal methods with SysML in aerospace design and engineering. Annals of Mathematics and Artificial Intelligence **63**(1) (2011) 53–102
10. Jacobs, J., Simpson, A.C.: A process algebraic approach to decomposition of communicating SysML blocks. International Journal of Modeling and Optimization **3**(2) (2013) 153–157