Formal Models of SysML Blocks

Alvaro Miyazawa[1], Lucas Lima[2], and Ana Cavalcanti[1]

[1] Department of Computer Science, University of York, York, UK
{alvaro.miyazawa,ana.cavalcanti}@york.ac.uk
[2] Centro de Informática, Universidade Federal de Pernambuco, Recife, Brazil
lal2@cin.ufpe.br

Abstract. In this paper, we propose a formalisation of SysML blocks based on a state-rich process algebra that supports refinement, namely, CML. We first establish a set of guidelines of usage of SysML block definition and internal block diagrams. Next, we propose a formal semantics of SysML blocks described by diagrams that conform to our guidelines. The semantics is specified by inductive functions over the structure of SysML models. These functions can be mechanised to support automatic generation of the CML models.

Keywords: CML, SysML, process algebra, refinement, semantics.

1 Introduction

SysML is an extension of UML 2.0 to support modelling for systems engineering. In recent years, it has increasingly been supported by a number of tool vendors such as IBM [13], Atego [1] and Sparx Systems [19].

Our aim is to support the application of formal analysis tools and techniques at the level of the graphical notations used in current industrial practice. In particular, in this paper, we present our results on formalising the notion of SysML blocks including their related elements such as associations, compositions, generalisations, ports, interfaces and connectors. This is achieved by a denotational semantics of SysML blocks in the COMPASS modelling language (CML) [22], a formal specification language that supports a variety of analysis techniques [4].

Whilst SysML is an informal graphical notation, CML builds on well known and widely used formal specification languages: VDM [9] and CSP [12]. Its approach to modelling reactive behaviour and its semantic model are those adopted in the *Circus* [3] family of refinement languages.

The semantics of both CML and *Circus* use the Unifying Theories of Programming to cater for object-orientation [17], time [18], and synchronicity [2], for instance.The distinguishing feature of CML and *Circus* is the support for modelling at various levels of abstraction, and compositional refinement, including formal derivation (or verification) of code.

We present a denotational semantics for blocks in SysML models using CML. Its main distinctive feature is the fact that it can be used as an integration context for formal models of other SysML elements such as state machine, activity and sequence diagrams. The semantic function is formalised via translation rules, which can be used to generate CML models of blocks automatically; they are presented in [14]; here we illustrate the modelling approach via examples. Currently, our translation rules are being used as a basis for an implementation of a CML semantics of SysML based on the Atego's Artisan Studio [1]. As far as we know, there are no formal accounts in the literature of the behavioural semantics of SysML blocks that support integration with other diagrams.

The CML semantics enables a variety of refinement-based analysis of SysML models. CML tools [4] include an Eclipse-based development environment (parser and type-checker) with links to Artisan Studio [1] to support design using SysML and RT-Tester [20] for test automation, and plug-ins that support the generation of proof obligations, simulation, theorem proving based on Isabelle/HOL [16], model checking, and the application of a refinement calculus. The use of CML to reason about systems of systems is discussed in [22], and compositional refinement-based reasoning techniques are described and formalised in [15].

In SysML, the behaviour of blocks may be specified by state machine diagrams, and operations may be specified by activity diagrams, which describe a form of flowchart. Sequence diagrams may be used to model particular scenarios of interaction between elements of the model. Our approach considers process models for state machine, activity and sequence diagrams. An approach to the construction of these models is described in [14].

This paper is structured as follows. Sections 2 and 3 briefly present CML and SysML. Section 4 describes our guidelines of usage of SysML blocks and the formal model of SysML blocks by means of a simple example. Section 5 discusses related work and Section 6 summarises our results and discusses future work.

2 CML

A CML specification consists of a number of paragraphs, which at the top level can declare types, classes, functions, values (i.e., constants), channels, channel sets, and processes. Both classes and processes declare state components, and may contain paragraphs declaring types, values, functions and operations. Processes also have actions, which provide a behavioural specification including data operations (using VDM) and interaction patterns (using CSP).

Processes are the main elements of a CML specification; systems and their components are both specified by processes that encapsulate some state and communicate with each other and the external environment via channels. A process may declare any number of actions, and must contain an anonymous main action, which specifies the behaviour of the process.

Other features of CML used in this paper are explained as necessary. For further details on CML, we refer to [21,22]. An example is presented in Figure 1.

This specification declares a type `Item` of natural numbers, a constant `MAX`, two channels `put` and `get` that communicate values of type `Item`, and four processes. The first process, `Producer`, has a state component `i` whose initial value is 0; it records the number of items sent through `put`. Its behaviour is defined by a recursive action that increments `i`, sends `i` through `put` (`put.i`) and waits one time

```
types
  Item = nat
values
  MAX = 5
channels
  put, get: Item
process Producer = begin
  state i: nat := 0
  @ mu X @ i := i+1; put!i -> Wait(1); X
end
process Consumer = begin
  @ mu X @ get?x -> Wait(2); X
end
process Buffer = begin
  state b: seq Item
  @ mu X @ ([len b > 0] & get!(hd b) -> b := tl b
             [] [len b < MAX] & put?x -> b := b^[x]
  ); X
process System = (Buffer [|{|put,get|}|]
                  (Producer ||| Consumer))\{|put,get|}
```

Fig. 1. CML excerpt

unit (`Wait(1)`) before recursing (`X`). `Consumer` reads a value `x` from `get` (`get?x`) and waits two time units before recursing. `Buffer` maintains a sequence `b` of values of type `Item`, and recursively allows a choice (`[]`) of communications on `put` and `get` depending on whether `b` is not empty (`len b > 0`) or not full (`len b < MAX`). The value output via `get` is the first element of `b` (`hd b`), and the value `x` input via `put` is appended to the end of `b` (`b := b^[x]`).

The overall specification is given by the process `System`, which composes in parallel the three previous processes. `Producer` and `Consumer` are composed in interleaving (`|||`), that is, without communication, and their composition is put in parallel (`[|{|put,get|}|]`) with `Buffer` synchronising on the channels `put` and `get`. Finally, these channels are made internal using the hiding operator (\).

Another example of a CML model is sketched in Section 4.

3 SysML

SysML is built as a UML profile, that is, it reuses part of the UML metamodel and extends it with some specific features from system engineering. The classic software-centric focus of UML, through class and composite structure diagrams, has been moved to the system level in SysML by the introduction of the block definition diagram (bdd) and internal block diagram (ibd). The UML notion of interfaces has been focused in SysML on system-level interfaces by the introduction of ports, which are located in the boundary of a block and may communicate

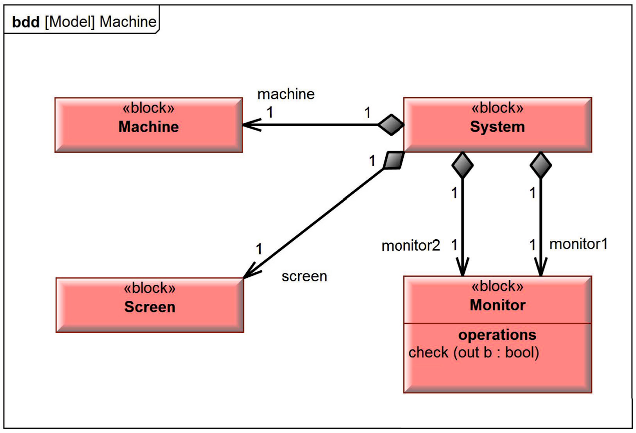

Fig. 2. Block Definition Diagram

service-based data and flow-based items; we identify two sides of a port with respect to the block that contains it: internal and external. Blocks are based on UML classes and composite structures with some changes and extensions. A block is defined in terms of a structural part, which can include constraints, properties (simple attributes), and parts (that may be typed by another block), and a behavioural part, which is defined in terms of operations and signals. Blocks can communicate with each other by sending events, which correspond to sending a signal or an operation call from one block to another. Whilst signals and operation calls are elements of the model, signal events and operation call events are occurrences of the model elements in a particular time point.

A bdd defines a structure of blocks and their relationships such as associations, generalisations, and dependencies. It is based on UML class diagrams with restrictions and extensions. Figure 2 shows an example where the block System is linked by a part-whole association (known in UML as composition) to three other blocks: Machine, Screen and Monitor. This diagram provides a view of the main components of the example we use here to illustrate our semantics. The actual configuration of the parts of System is described by the ibd in Figure 3.

An ibd is a modified version of a UML composite structure diagram. It captures the internal organization of a block in terms of its parts and the connection between them. Whilst in a bdd the blocks can be compared to classes, in an ibd the connected parts resemble instances of classes. Usually, these parts are typed by other blocks, hence, the diagram explains how the instances of blocks communicate with each other. Such communications can be represented by a direct link between the parts or by connected ports. For example, the ibd in Figure 3 shows that the blocks Monitor and Screen have each one port (p and p2, respectively), Machine has three ports (m1, m2 and p1), and System has one port (p3). Ports may restrict the kind of communication that can happen between blocks by specifying interfaces, which define the operations and signals that the

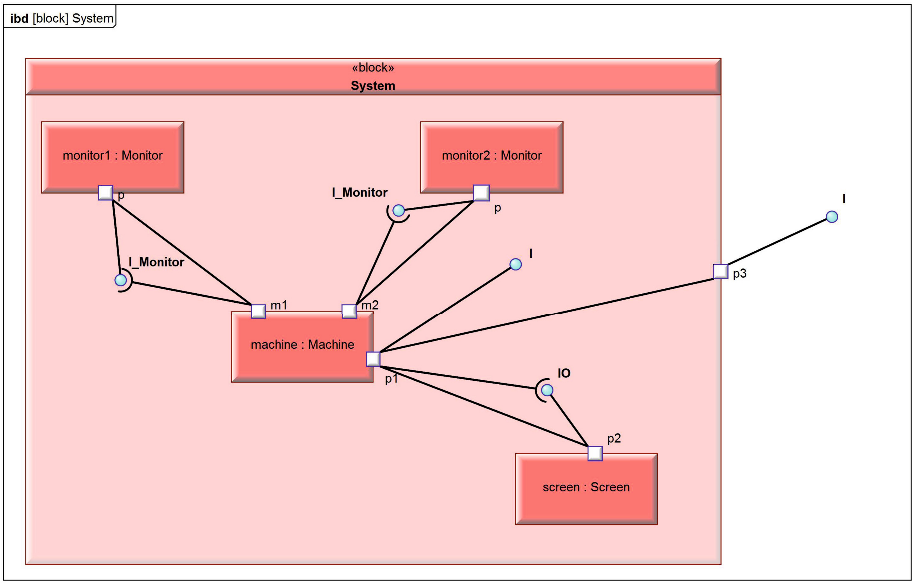

Fig. 3. Internal Block Diagram

block provides and requires. For instance, the port p1 in Figure 3 provides the interface I, which contains signals that control the machine, and requires the interface IO, which contains an operation that allows printing. Further details of our running example are presented where necessary.

In the next section, to explain our approach to define CML models of SysML blocks, we present the formal model of our running example.

4 Formal Models of SysML Blocks

We assume that the SysML model is sufficiently complete to allow the derivation of a well formed CML model. This assumption is decomposed in a number of guidelines that can be divided into three groups:

Entity Definition. These guidelines require that elements such as operations, blocks and associations are defined somewhere in the model. For instance, it requires that operations are defined either via the action language, a state machine or an activity diagram.

Instance Definition. These guidelines require that enough information about the instances of composite blocks is available. For instance, it requires that the parts of a composite block and their interconnections are specified.

Simplification Assumptions. These guidelines provide alternatives to the use of certain elements, where they have an equivalent counterpart, or define how they can be used. An example of such guidelines is the requirement that asynchronous operations are modelled as signals. This requirement stems from the fact that the meaning in SysML of an asynchronous operation with return value is unclear and that asynchronous operations without return values and signals can be considered equivalent. In this case, the guidelines propose the use of signals as an alternative to asynchronous operations.

The full list of guidelines can be found in [14]. A SysML model that respects our guidelines has the following structure. It is formed by blocks that may have properties and offer services defined by operations and signals. A simple block contains properties, operations, signals and ports, whilst a composite block contains parts that are typed by blocks and ports. A block is defined in the SysML metamodel and its content is obtained from the diagrams that refer to it.

Our CML definition of a SysML block is specified by a semantic function that calculates the model of a block in terms of the models of its parts. This semantic function is formalised by translation rules, whose compositional nature makes traceability viable as they allow us to identify which parts of the CML model that they define correspond to particular elements of a given SysML model. Moreover, as opposed to *ad hoc* rules for model transformation, the semantic function formalised by our translation rules can be encoded in a theorem prover to support both the validation of the semantics and the analysis of the model using techniques based on theorem proving.

4.1 Structure of Models

We model a block as a CML process, where the state characterised by its properties is encapsulated (not accessible externally). For this reason, access to properties as well as interaction via operations and signals are modelled as communications through channels. The CML process that models a block can receive requests to read and write to the block's properties, as well as signals and operation calls. Each of these requests are received through CML channels whose names are the names of the blocks properties prefixed by `set_` and `get_`, or the name of the block appended with `_op` and `_sig`. Finally, a channel `_addevent` is used to delegate the treatment of event to the environment, which, for instance, can be a process that models an activity or a state machine diagram.

In general, a SysML model may contain a number of blocks that are not related to each other. In this case, our CML model consists of a number processes, one for each block, that are also not related to each other. An analysis needs therefore to focus on a particular process.

The model of a simple block is formed by the parallel composition of two or more basic processes: one specifying the behaviour of the block's parent, another called `simple_` process, modelling the behaviour of the block itself, and the remaining modelling the block's ports. This allows the reuse of the model of the parent block to reflect the structure of the SysML model in CML. A CML process that models a composite block is defined in terms of the processes that model its parts and ports.

Table 1. SysML-CML correspondence

SysML element	CML element
Simple block	Process
Composite block	Process
Port	Process
Connector	Channel
Interface	Class
Operation call	Record type
Signal	Record Type
Event	Communication

A port is modelled by a process that uses four channels: `ext_op`, `ext_sig`, `int_op` and `int_sig`. The first two allow the port to interact with a component external to the block by sending and receiving signals and operation calls as well as responses to operation calls. The last two are used to communicate with the model of the block. The behaviour of a port is to restrict which values can be received at each channel, and relay the accepted values to the equivalent channel on the other side of the port.

Table 1 shows the correspondence between elements of a SysML model and elements of CML. In general, a SysML element that exhibits some form of behaviour, namely, blocks and ports, are modelled by CML processes; connectors, which specify communication links, are modelled by channels; static elements (i.e, without intrinsic behaviour), namely operational calls and signals, are modelled by record types, and interfaces, which are collections of static elements are modelled by classes. Operation calls are considered static because they specify the message that is sent to blocks, not the behaviour of the operation itself, which is usually specified by a state machine or activity diagram. Events, like the communication of signals, are modelled by CML communications.

4.2 Integration with other SysML Model Elements

In our approach, the processes that model state machine and activity diagrams accept events through a channel `addevent`. These events are added to an event pool and are processed according to the semantics of the element (state machine or activity diagram). The processing of events may lead to the generation of new events as well as to changes to the state of the block that is associated with state machine or activity diagram. The models of sequence diagrams use the channels of the blocks included in the diagram.

To obtain the integrated model of a block whose behaviour is specified by a state machine diagram or whose operations are specified by activity diagrams, the processes that model the state machine and activity diagrams are composed in parallel with the process that models the block. Each of these processes synchronise on the events associated with a channel `_addevent` whose parameters include the operations and signals treated by the activity or state machine diagrams. For instance, if a state machine diagram treats the operations check,

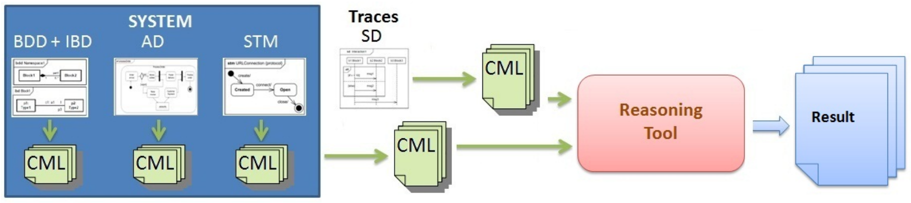

Fig. 4. Integrated Model Analysis Approach

and an activity diagram responds to a signal off, the synchronisation sets should be as follows. The first synchronisation set includes all events of the channel `addevent` where the first three parameters (representing the instance, source and target of a signal or operation call) are unrestricted and the fourth is limited to values whose type is the input record type of the operation check. Similarly, the second synchronisation set includes all events of the channel `addevent` where the first three parameters are unrestricted and the fourth is limited to values of the record type of the signal off.

Figure 4 illustrates an approach to the analysis of SysML models by establishing consistency between a sequence diagram, which describes possible valid traces, and the SysML model described by the blocks, and activity and state machine diagrams. The integrated CML model can be validated by reasoning tools, like a model checker, in order to check whether the system model is compatible with the flows of execution specified by the sequence diagram.

In the following sections, we present in more detail the model of blocks.

4.3 Structure of the CML Specification

Figure 5 gives an overview of the formal model of our example; it consists of a number of type declarations that encode the operations and signals found in the model, and classes that group some of these types. Additionally, channels and processes are declared to model SysML connectors and blocks.

A global type `ID` is used to identify instances of blocks as well as instances of model elements such as ports and states. It is declared as sequences of type token, which is the most unspecified type in CML, supporting only comparison. The use of sequences to identify instances allows us to produce unique identifiers based on the hierarchical structure of the models.

Next, a number of types are declared: two for each operation and one for each signal. These types encapsulate the parameter of the operations and signals. All the operation types are gathered in the type `OPS`, and similarly all signal types are gathered in the type `S`. These two types are then joined to form the type of all messages `MSG`. Next, classes are defined to declare the types of operations, signals and messages that correspond to the operations and signals of a block, port or interface. In the case of a port, the class is defined in terms of the interface classes and further distinguishes operations and signals according to the type of interface: provided or required.

```
types
    ID = seq of token
    check_I = <check_I> ... fix = <fix> ...
    OPS = check_I | check_O | print_I | print_O
    S = fix | on | off
    MSG = OPS | S
class I_types = ...
channels
    c_m1_p.ops: nat*ID*ID*OPS
    ...
    c_p1_p2.sig: nat*ID*ID*S
process Machine = ...
process Screen = ...
class Monitor_types ...
channels
  Monitor_op: nat*ID*ID*OPS Monitor_sig: nat*ID*ID*S
  Monitor_addevent: nat*ID*ID*MSG
process simple_Monitor = ...
process bare_Monitor = id: ID @ simple_Monitor(id)
process Monitor = ...
process System = ...
```

Fig. 5. Overview of the formal model of the example in Figure 2 and 3

Next, for each connector, two channels are declared. The values communicated by these channels correspond to instances of operation calls and signals dispatch; they identify the instance of the call or dispatch (using a natural number), the source and destination of the message (using `ID` values), and the message itself including any parameters (a value of type `OPS` or `S`).

Finally, the models of each of the blocks and ports are declared. These include new types and channel declarations as well as processes and channel sets.

4.4 Simple Blocks

The model of a simple block comprises a class, a number of channels and three processes. Figure 5 shows an overview of the declarations associated with the block Monitor in Figure 2. First, the class containing the types of signals and operations of the block is declared, and then the three channels previously described: `_op`, `_sig`, `_addevent`. Finally, the three processes that specify the behaviour of the block are declared.

The details of the process `simple_Monitor` are shown in Figure 6. This process is parametrised by a value that identifies an instance of the block. For example, the process that models the part monitor1 in Figure 3 is an instantiation of the process `Monitor` with an identifier `id^[mk_token(monitor1)]`, where `id` is the identifier of an instance of the block System.

```
process simple_Monitor = id: ID @ begin
  state enabled: Bag := empty_bag
  actions
    Monitor_state = Skip
    Monitor_requests = mu X @ (
      Monitor_op?n?o!id?x:(is_Monitor_types‘check_I(x)) -> (
        [is_Monitor_types‘check_I(x)] &
          Monitor_addevent!n!o!id!x -> Skip;
          enabled := bunion(enabled, Monitor_types‘check_O)
      ) [] Monitor_op?n?o!id?x:(in_bag(x,enabled)) -> (
        [is_Monitor_types‘check_O(x)] &
          enabled := bdiff(enabled, Monitor_types‘check_O)
      ) [] ...
    ); X
  @ Monitor_state [||{}|{enabled}||] Monitor_requests
end
```

Fig. 6. Process `simple_Monitor`

The process `simple_Monitor` declares a state containing all the properties of the block, and an extra component `enabled` whose type is a bag of elements of the type `OPS`. This component models the responses to operation calls that the block can communicate, and since multiple calls to the same operation may occur, this state component must be able to hold any number of identical values, thus the use of a bag. The component `enabled` is initialised with the empty bag. Next, two actions are declared: the first controls access to the state components that model block properties, and the second controls the communication of operation and signal messages. The two actions are composed in interleaving to specify the overall behaviour of the process (main action).

Since Monitor does not have properties, the state of the process declares a single component, namely `enabled`, and the action that controls the access to the block properties is declared as `Skip`, that is, the action that terminates immediately. If a block has properties, the state declares corresponding components, and this action recursively offers a choice between reading or writing to the state components through the channels `set_` and `get_`.

The second action `Monitor_requests` controls which signals and operation calls can be accepted by the block, and which values can be communicated as a response to an operation call. This action is recursive; each cycle corresponds to the handling of a signal or operation call, or the response to an operation call. Each cycle offers a choice between receiving values of an input record type or sending values of an output record type on the channel `Monitor_op`, and receiving values of a signal record type on the channel `Monitor_sig`. The input types that can be received are restricted to those in the class `Monitor_types`, which contains the signal, input and output record types of the signals and operations in the block

Monitor. The restriction of the communication is achieved by constraining the parameter `x` of the communication `Monitor_op?n?o!id?x`, which corresponds to a signal or operation call, with the predicate `is_Monitor_types‘check_I(x)`. The function `is_Monitor_types‘check_I` becomes available in the CML specification when the type `Monitor_types‘check_I` is declared.

After receiving any value `x` of the input record type of one of the operations in the block, the event is communicated through the channel `Monitor_addevent` (and can then be treated by the environment, perhaps characterised by state machine or activity diagrams, as explained in Section 4.2). Since values of output record types can only be communicated after a corresponding input record type value has been received (that is, an operation can only terminate after it has been started), after sending the event through `Monitor_addevent`, all the possible values of the output record type of the operation are added to the bag `enabled`.

The second possible communication in a cycle of the recursion restricts the possible values of output record types that can be communicated. Only values that are in the component `enabled` can be communicated as a response to an operation call. Once the communication is completed, one instance of each value of the appropriate output record type is removed from the bag. The third choice treats signals in a way similar to the treatment of operation calls just explained.

The two actions `Monitor_state` and `Monitor_requests` are composed in interleaving to define the main action of the process. The interleaving operator `[{}||{enabled}]` partitions the state between the actions to avoid race conditions. In the process `simple_Monitor`, the first action has no write access to the state, and the second action has write access to the state component `enabled`.

Processes prefixed by `bare_` model SysML's generalisation (inheritance) relation as interleaving. If a block A has parents B and C, the process `bare_A` is the interleaving of `simple_A`, `bare_B` and `bare_C`, renaming the channels of the last two processes to match those of `simple_A`. This is necessary to allow signals and operations defined in the parents' models to be communicated through the channel of A. The process `bare_Monitor` in Figure 7 is just the instantiation of `simple_Monitor` because Monitor does not have parents.

`Monitor`, depicted in Figure 7, is the parallel composition of `bare_Monitor` and the process modelling the port p (`port_p`) with its `p_int_op` and `p_int_sig` channels (see Section 4.6) renamed to the corresponding channels in the block Monitor. The renaming is necessary to synchronise these channels and the channels of the block to model the fact that values received in the port of a simple block are relayed to the block. This parallel composition synchronises on the channels `Monitor_op` and `Monitor_sig` where the source and destination of the communication are the block and the port, and vice-versa. That is, the events on those channels that communicate values between the port and the block. These events are then made internal through the hiding operator (\) because the communication between a block and its ports is implicit in the SysML model.

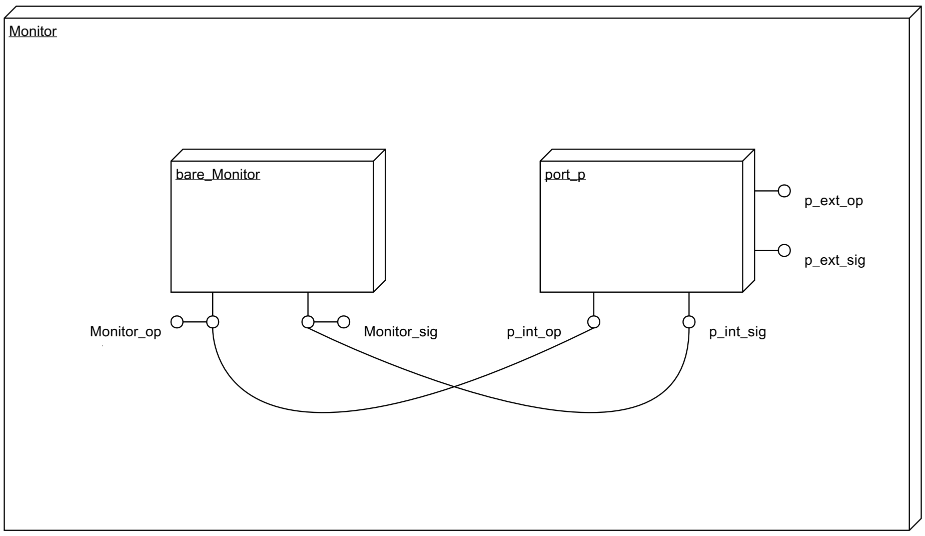

Fig. 7. Structure of the process `Monitor`

4.5 Composite Blocks

The model of a composite block is a process that composes in parallel the models of its parts and ports. The pattern of communication within the parallelism is determined by the connectors between the parts and ports. The channels associated with a connector are used to rename the channels of processes that model the blocks or ports to which the connector is attached. The renamed processes are then composed in parallel synchronising on the channels associated with the connectors. Figure 8 gives an overview of the structure of System.

The body of the process `System` is defined as the parallel composition of five processes (four parts and one port), whose channels have been renamed with the channels associated with any connectors reaching the block or its ports, synchronising on the channels associated with the connectors. For instance, the process `Machine(...)` has the external channels of the ports m1, m2 and p1 renamed to the channels of the connectors from m1 to p, m2 to p, p1 to p3, and p1 to p2, and similarly the channels of the processes that model the ports p, p2 and p3 are renamed with the channels of these connectors. The synchronisation between these channels is depicted in Figure 8 by the curves lines connecting the processes (represented by boxes). Whilst in the case of ports in the parts of the block the renaming occurs on the external channels of the port, in the case of ports in the block itself, the renaming takes place on the channel associated with the internal side of the port. This can be observed in Figure 8 by the connection between the channels (straight lines with black dot) `p3_int_op` and `p1_ext_op`, and the channels `p3_int_sig` and `p1_ext_sig`. The renaming of port channels by connector channels is necessary because the ports of a block may be linked to

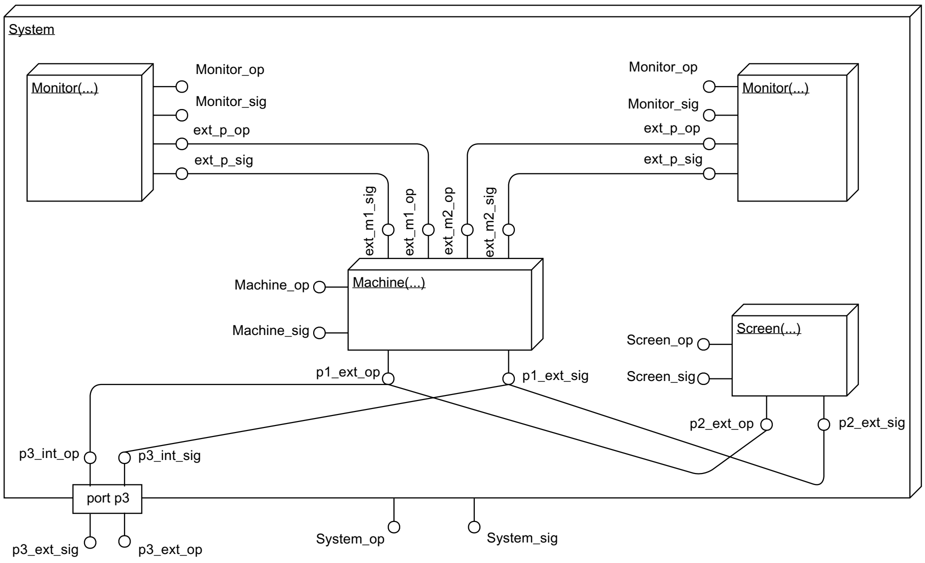

Fig. 8. Overview of the model of the block System

different connectors in different contexts, and thus the link must be established at the level of the port process instantiation, and not in its declaration.

4.6 Ports

The models of ports consist of a class, four channels and one process. The class differs from that of blocks and interfaces in that it distinguishes between operations and signals contributed by provided and required interfaces; it defines six types, three for provided interfaces and three for required interfaces. These types identify the input, output and signal record types that are defined in terms of the interfaces. For port p1, the type of provided signals is defined as the type `S` of the provided interface I.

The four channels declared for a port allow the communication of operation and signal records from both sides of the port (internal and external). These channels are used in the process `port_p1` to restrict the kind of message that can be received in the port depending on the direction, and also to relay the message to the appropriate destination. For instance, calls for provided operations (values of a provided input record type) can only be received on the external channel `p1_ext_op`. In this case, the port relays the message through the internal channel `p1_int_op`. Notice that whilst the message is received by the port (third parameter of the communication is `id`), it relays the message from the port to some unknown target (`?y`). This reflects the fact that the target of the message is not determined until the port is associated with a block. This is achieved through renaming as shown in Figure 8 by the connection between the channels.

```
process port_p1 = id: ID @ begin
  @ mu X @ (
    p1_ext_sig?i?o!id?x:(is_p1_types‘P_S) ->
      p1_int_sig.i.id?y.x -> Skip
    [] p1_int_sig?i?o!id?x:(is_p1_types‘R_S) ->
      p1_ext_sig.i.id?y.x -> Skip
    [] p1_ext_op?i?o!id?x:(is_p1_types‘P_I) ->
      p1_int_op.i.id?y.x -> Skip
    [] p1_int_op?i?o!id?x:(is_p1_types‘P_O) ->
      p1_ext_op.i.id?y.x -> Skip
    [] p1_ext_op?i?o!id?x:(is_p1_types‘R_O) ->
      p1_int_op.i.id?y.x -> Skip
    [] p1_int_op?i?o!id?x:(is_p1_types‘R_I) ->
      p1_ext_op.i.id?y.x -> Skip
  ); X
end
```

Fig. 9. Overview of the communication patterns of a port

The complete formalisation of the semantics of SysML blocks as well as state machine, activity and sequence diagrams can be found in [14].

5 Related Work

Graves [10] proposes a representation of a restricted subset of SysML block diagrams in OWL2, which is a language for knowledge representation based on a description logic. Ding and Tang [5] proposes a representation of SysML block diagrams directly in a description logic. In both cases, block diagrams are restricted to include only associations and simple blocks.

Graves and Bijan [11] extend [10] by encoding SysML diagrams into a type theory that axiomatises block diagram notions of types, properties and operators. Both bdds and ibds are covered, but dynamic aspects of SysML diagrams, such as the treatment of operation calls, are not.

All these works focus on generating a set of axioms that specify a system based on a SysML diagram, and then using techniques for the underlying logic to check properties. Although Graves and Bijan [11] describe model refinement as theory refinement (that is, modification of the knowledge base aiming at achieving consistency), they do not elaborate on the topic, and it is not clear what properties are preserved by this notion of refinement.

Evans and Kent [6] describes the pUML approach, which aims at strengthening the meta-model semantics of UML via a precise semantics. This work focusses on the semantics of generalisation and packages, and provides a number of extra well-formedness conditions. Dynamic aspects of UML models are not discussed, and it is not clear how the pUML approach tackles such aspects.

The work presented in [8] is the closest to ours. It formalises UML-RT structure diagrams in CSP-OZ [7], and whilst the treatment of composition and connectors is similar to ours, the semantics does not cover issues related to operation calls and integration with other diagrams.

6 Conclusions

In this paper, we have presented a behavioural model of SysML blocks that includes simple and composite blocks, generalisation, association and composition relations, standard ports and connectors, interfaces, operations, properties and signals. To the best of our knowledge, this is the first formalisation of the behavioural semantics of a comprehensive subset of the block notation.

The main characteristics of our approach are the compositionality of the generated models, the use of parallelism to compose different aspects of the system and the support for refinement. These aspects make it possible to apply compositional analysis techniques [15], and refinement strategies to obtain equivalent models better suited to alternative analysis techniques (e.g., model checking).

The most interesting aspects of our models are the treatment of operation calls, the use of interleaving to model inheritance, and the use of parallelism to model block composition. Finally, whilst the models of ports are simple, the use of interface classes in the specification of a port's communication protocol proved important to preserve the compositionality of our models.

The functions that characterise our semantics are specified by translation rules, which take elements of the SysML abstract syntax and produce the corresponding CML elements. The complete set of translation rules for SysML blocks can be found in [14]. These rules are currently being implemented in Artisan Studio to support the automatic generation of CML from SysML models. This work is being carried out by our industrial partners at Atego, and the revision of the rules by a SysML expert and the process of mechanising the rules have helped us partially validate our semantics.

As future work, we plan to further validate our semantics by completing the automation of the translation rules, simulating the models using the CML tools, applying refinement strategies for model simplification, and analysing models via model-checking. Finally, we plan to encode the translation rules in Isabelle/HOL and prove general properties of our semantics.

Acknowledgement. This work is supported by the EU FP7 Project COMPASS (`http://www.compass-research.eu`).

References

1. Artisan Studio, `http://atego.com/products/artisan-studio/` (accessed: April 11, 2013)
2. Gancarski, P., Butterfield, A.: The Denotational Semantics of slotted-*Circus*. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 451–466. Springer, Heidelberg (2009)

3. Cavalcanti, A.L.C., Sampaio, A.C.A., Woodcock, J.C.P.: A Refinement Strategy for *Circus*. Form. Asp. Comp. 15(2-3), 146–181 (2003)
4. Coleman, J.W., Malmos, A.K., Larsen, P.G., Peleska, J., Hains, R., Andrews, Z., Payne, R., Foster, S., Miyazawa, A., Bertolini, C., Didier, A.: COMPASS Tool Vision for a System of Systems Collaborative Development Environment. In: 7th International Conference on System of Systems Engineering, pp. 451–456 (2012)
5. Ding, S., Tang, S.Q.: An approach for formal representation of SysML block diagram with description logic SHIOQ(D). Proceedings of the 2nd ICIIS 2, 259–261 (2010)
6. Evans, A., Caskurlu, B.: Core Meta-Modelling Semantics of UML: The pUML Approach. In: France, R.B. (ed.) UML 1999. LNCS, vol. 1723, pp. 140–155. Springer, Heidelberg (1999)
7. Fischer, C.: CSP-OZ: A combination of Object-Z and CSP. In: Bowmann, H., Derrick, J. (eds.) FormalMethods for Open Object-Based Distributed Systems (FMOODS 1997), vol. 2, pp. 423–438. Chapman & Hall, Ltd. (1997)
8. Fischer, C., Olderog, E.-R., Wehrheim, H.: A CSP View on UML-RT Structure Diagrams. In: Hussmann, H. (ed.) FASE 2001. LNCS, vol. 2029, pp. 91–108. Springer, Heidelberg (2001)
9. Fitzgerald, J., Larsen, P.G.: Modelling Systems – Practical Tools and Techniques in Software Development, 2nd edn. Cambridge University Press (2009)
10. Graves, H.: Integrating SysML and OWL. In: Proceedings of OWL: Experiences and Directions (2009)
11. Graves, H., Bijan, Y.: Using formal methods with SysML in aerospace design and engineering. Ann. Math. Artif. Intel., 1–50 (2011)
12. Hoare, C.A.R.: Communicating sequential processes. Prentice-Hall, Inc. (1985)
13. Rational Rhapsody Architect for Systems Engineers, `http://www-142.ibm.com/software/products/us/en/ratirhaparchforsystengi` (accessed: April 11, 2013)
14. Miyazawa, A., Albertins, L., Iyoda, J., Cornélio, M., Payne, R., Cavalcanti, A.: Final report on combining SysML and CML. Technical report, COMPASS (2013)
15. Oliveira, M., Sampaio, A., Antonino, P., Ramos, R., Cavalcanti, A., Woodcock, J.: Compositional analysis and design of CML models. Technical report, COMPASS (2013)
16. Paulson, L.C.: Isabelle: A Generic Theorem Prover. LNCS, vol. 828. Springer, Heidelberg (1994)
17. Santos, T., Cavalcanti, A., Sampaio, A.: Object-Orientation in the UTP. In: Dunne, S., Stoddart, B. (eds.) UTP 2006. LNCS, vol. 4010, pp. 18–37. Springer, Heidelberg (2006)
18. Sherif, A., Cavalcanti, A., Jifeng, H., Sampaio, A.: A Process Algebraic Framework for Specification and Validation of Real-time Systems. Form. Asp. Comp. 22, 153–191 (2010)
19. Sparx Systems' Enterprise Architect supports the Systems Modeling Language, `http://sparxsystems.com/products/mdg/tech/sysml/` (accessed: April 11, 2013)
20. RT-Tester, `http://verified.de/en/products/rt-tester` (accessed: April 11, 2013)
21. Woodcock, J., Cavalcanti, A., Coleman, J., Didier, A., Larsen, P.G., Miyazawa, A., Oliveira, M.: CML Definition 0. Technical Report D23.1, COMPASS (2012)
22. Woodcock, J., Cavalcanti, A., Fitzgerald, J., Larsen, P., Miyazawa, A., Perry, S.: Features of CML: A formal modelling language for Systems of Systems. In: 7th International Conference on System of Systems Engineering, pp. 1–6 (2012)