Software and Systems Modeling manuscript No.
(will be inserted by the editor)

An integrated semantics for reasoning about SysML design models using refinement

Lucas Lima · Alvaro Miyazawa · Ana Cavalcanti · Márcio Cornélio · Juliano Iyoda · Augusto Sampaio · Ralph Hains · Adrian Larkham · Vaughan Lewis

July 26, 2016

Abstract SysML is a variant of UML for systems design. Several formalisations of SysML (and UML) are available. Our work is distinctive in two ways: a semantics for refinement and for a representative collection of elements from the UML4SysML profile (blocks, state machines, activities, and interactions) used in combination. We provide a means to analyse and refine design models specified using SysML. This facilitates the discovery of problems earlier in the system development lifecycle, reducing time and costs of production. Here, we describe our semantics, which is defined using a state-rich process algebra and implemented in a tool for automatic generation of formal models. We also show how the semantics can be used for refinement-based analysis and development. Our case study is a leadership-election protocol, a critical component of an industrial application. Our major contribution is a framework for reasoning using refinement about systems specified by collections of SysML diagrams.

L. Lima · M. Cornélio · J. Iyoda · A. Sampaio
Centro de Informática, Universidade Federal de Pernambuco, Recife, Brazil

L. Lima
E-mail: lal2@cin.ufpe.br

A. Miyazawa · A. Cavalcanti
Department of Computer Science, University of York, Heslington, York, UK

A. Miyazawa
E-mail: Alvaro.Miyazawa@york.ac.uk

R. Hains · A. Larkham · V. Lewis
Atego, 701 Eagle Tower, Montpellier Drive, Cheltenham, UK

1 Introduction

The increasing complexity of systems have led to increasing difficulty in design. The standard approach to development, based on trial and error, with testing used at later stages to identify errors, is costly and leads to unpredictable delivery times. In addition, for critical systems, for which safety is a major concern, early verification and validation (V&V) is recognised as a valuable approach to promote dependability. In this context, we identify three important and desirable features of a V&V technique: (i) a graphical modelling language; (ii) formal and rigorous reasoning, and (iii) automated support for modelling and reasoning.

Our goal is to address these points with a refinement technique for SysML [1] supported by tools. SysML is a UML-based language. There is wide availability of literature on SysML [2,3], tool support from vendors like IBM [4], Atego [5], and Sparx Systems [6].

Refinement in our context is a technique to assess behaviour preservation. If a model M_1 is refined by a model M_2, then M_2 preserves the behaviour of M_1: the traces of interactions, deadlocks, and divergences of M_2 are also possible for M_1. With refinement, we enable reasoning at various levels of abstraction, and about model transformations that may take place between an abstract and a concrete model capturing particular architectural designs, data representations, and (distributed) algorithms. For example, if state machines are built as part of the conceptual and low-level design models, we can make sure that the state machine in the low-level design respects the requirements embedded in that of the conceptual diagrams.

SysML provides flexibility and generality for model development and evolution. Its semantics is described in natural language. To apply automated techniques for analysis and verification of SysML models, however, we require a precise well-defined semantics.

Our main contributions are: (i) guidelines of usage for construction of meaningful SysML models; (ii) a state-rich process algebraic semantics for SysML models, in particular, a CML semantics; and (iii) applications of the CML model in reasoning at the diagrammatic level. Our focus is on SysML due to our interest in system engineering, particularly design of systems, but most of our results are also relevant for UML.

We support refinement-based analysis and verification by providing a semantics for SysML elements based on CML (COMPASS Modelling Language) [7]. Many have pursued formalisation of SysML and UML: there are works on individual diagrams [8–11], and on combinations of diagrams [12]. Typically, approaches for collections of elements represented in different diagrams consider constrained subsets of the abstract syntax.

The fUML (Semantics of a Foundational Subset for Executable UML Models) [13], for example, proposes a standard semantics for a subset of UML limited to classes and activities. It has an executable semantics described in Java, and an axiomatic semantics. The PSCS (Precise Semantics of UML Composite Structures) [14] is an extension of fUML and was developed to enable the usage of UML composite structures. Various works provide support for reasoning about fUML models via transformation to a target language for which formal verification techniques are available [15, 16]. Our main contribution is support for refinement-based reasoning. We support automatic generation (via translation rules) of refinement models from the diagrammatic notation and automated analysis of the generated models.

We focus on diagrams that facilitate design, namely the block-definition, internal block, state machine, activity, and sequence diagrams. Our work is distinctive in its coverage of the abstract syntax of the model elements, and, most importantly, it adopts an integrated approach. We define a single formal model that captures the data and behavioural aspects of a SysML model that uses a collection of diagrams. It is an integrated semantics in that the semantics of the individual elements are combined to support reasoning about complete models that include information about related blocks, activities, state machines, and interactions.

To support reasoning at the level of the diagrams, so that developers and verifiers are not required to have expertise on CML or formal techniques, our semantics can be used for automatic generation of CML models from SysML models. It takes the form of a function that maps SysML to CML and is defined by transformation rules, which are implemented in a model-generation tool based on Atego's Artisan Studio [5].

To enable the construction of meaningful CML models, we define usage guidelines for SysML. Basically, block diagrams are used to define the system and its components, internal block diagrams to define the relationship between them, each operation is implemented as a state machine or an activity, but not both, and sequence diagrams define scenarios of the system. Our guidelines ensure that we can produce useful CML models, but are not restrictive. Using an industrial case study, namely, a leadership-election protocol used in an industrial multimedia system of systems, we illustrate the guidelines and the use of refinement.

CML semantics for block, internal block, activity, and sequence diagrams have been presented in [17–19]. Here, we provide a revised and integrated version of these works. The CML semantics of each of the SysML elements has been changed to accommodate the integration. To establish its relevance, we discuss and illustrate the kind of analysis enabled by these comprehensive models. We note, however, that due to the compositionality of the CML constructs with respect to refinement, independent analysis of the individual SysML elements and of their components is still possible.

Using our CML semantics, we can define notions of refinement for complete SysML models as well as individual elements and their constructs. We show how these notions can be used to underpin refinement laws that support the sound transformation of diagrammatic models. We also show how we can use refinement to analyse properties of diagrammatic models.

Next, we give an overview of SysML and CML. Section 3 presents the SysML modelling pattern that characterises our guidelines of usage. Sections 4 and 5 present our CML semantics: we give an overview of our approach to integrated modelling and discuss the model of the individual SysML elements. Section 6 is concerned with the analysis of models. In Section 7 we present our efforts to implement and validate our technique. Section 8 presents related work, and in Section 9 we summarise our results and indicate future work.

2 Preliminaries

Here, we briefly introduce SysML (structural and behavioural diagrams) and CML, including the key concept of refinement in CML.

2.1 SysML

Like UML, SysML provides several diagrammatic modelling constructs. Here, we describe the structural and behavioural model elements covered by our semantics;

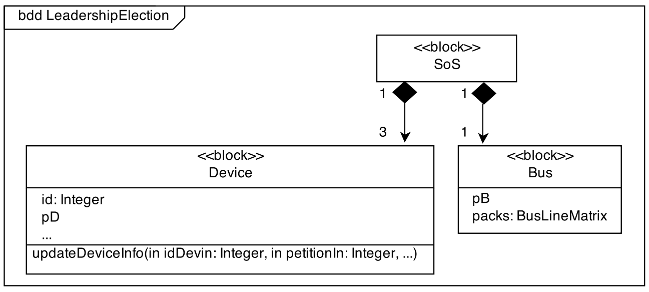


Fig. 1: A block-definition diagram.

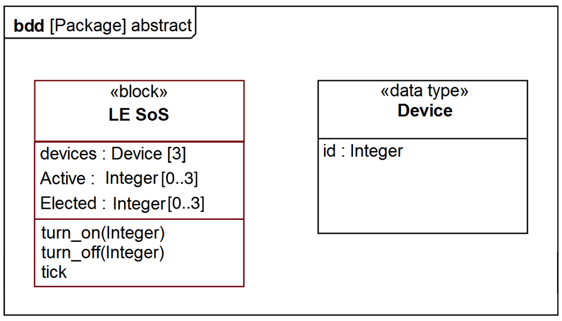


Fig. 2: Abstract model: block-definition diagram

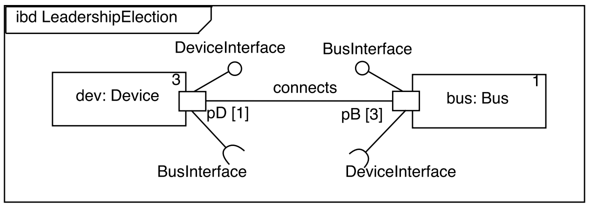


Fig. 3: An example of an internal block diagram.

a comprehensive account is in [1–3]. To illustrate the SysML notation, we use diagrams of our case study, the leadership-election protocol. It is a system (of systems) including a number of devices that automatically elect a leader among them. The main requirement is that there is exactly one leader. This is captured in a conceptual design specified by an abstract SysML model. The implementation uses a distributed architecture with no centralised control; it is specified by an alternative, more concrete, SysML model.

2.1.1 Structural diagrams

These include the block-definition and internal block diagrams, which we describe next.

Block-definition diagrams depict blocks and their relationships. A block can represent any abstract or real entity: a piece of hardware, a software, or a physical object, for instance. The whole system is also represented by a block. Figure 1 shows a diagram for the concrete model of our case study. There are three blocks: SoS represents the complete system (of systems), Device, three devices, and Bus, one bus.

Blocks can have attributes and operations. Attributes are properties of a block. For instance, in Figure 1, id is an integer attribute of the block Device, and packs is an attribute of Bus. The attributes pD and pB are ports, which are used for connecting blocks.

An operation captures functional behaviour provided by the block. For instance, the state of a Device can be updated via the operation updateDeviceInfo(). Operations are triggered by synchronous or asynchronous requests. A signal, on the other hand, does not have a specific behaviour, but may trigger behaviours in state machines and activities. It is used for asynchronous communication between blocks or with the environment.

A block can be related to another by an association, which indicates a potential relationship between parts of the blocks. Blocks can also be related by composition, indicated by an arrow with a filled diamond at one end. A composition establishes a whole-part relationship: the block at the diamond end is the whole composite block and those at the other end are the part blocks. A part owned by composition defines a local usage of its defining block within the specific context to which the part belongs. In our example, SoS is a composite block; its parts are three Devices and one Bus.

In the abstract model of the leadership-election protocol, the system is modelled by the block LE SoS in Figure 2, which has three properties, devices, Active and Elected, the operations turn_on and turn_off, and the signal tick. The attributes record the devices, those that are Active, and the current Elected leader. The type of Elected is a set because the system can be in an undesired state where no devices claim to be the leader. The type Device has a single component, id, that records the identifier of a device. The operations of LE SoS take as a parameter an identifier in devices and model the activation and deactivation of the corresponding device. The signal tick models the discrete passage of time.

Internal block diagrams are similar to block-definition diagrams, but typically show the internal connections between parts of a block. Figure 3 shows the parts of the block SoS defined in the diagram in Figure 1.

In an internal block diagram, a connector links two or more parts of a block, either directly or via ports, to allow them to communicate. For example, in Figure 3, three Device parts named dev are connected to one Bus part named bus. This is defined via the connector be-

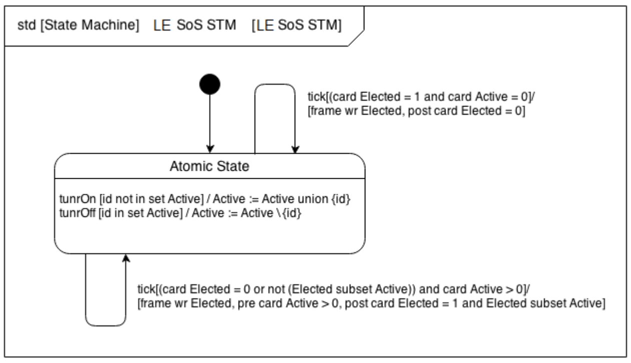

std [State Machine] LE SoS STM [LE SoS STM]
tick[(card Elected = 1 and card Active = 0]/
[frame wr Elected, post card Elected = 0]
Atomic State
tunrOn [id not in set Active] / Active := Active union {id}
tunrOff [id in set Active] / Active := Active \ {id}
tick[(card Elected = 0 or not (Elected subset Active)) and card Active > 0]/
[frame wr Elected, pre card Active > 0, post card Elected = 1 and Elected subset Active]

Fig. 4: Abstract model: state machine diagram

tween the ports pD and pB, represented by a solid line. This is in contrast with an association, which specifies that there can be links between any parts of the associated blocks, rather than that there is a link between parts owned by the same other part.

Ports can define interfaces. For example, the ports pD and pB define provided and required interfaces. A provided interface is depicted as a circle and identifies a port that produces outputs to its client. A required interface is depicted as a semicircle and identifies a port that takes inputs from its client.

Finally, the number in a port defines its multiplicity. For instance, each device has a port pD with multiplicity one, whereas the port pB of the single instance of the bus has multiplicity 3. This specifies the connection of one bus with three devices.

2.1.2 Behavioural diagrams

We now give a brief overview of the notation for state machine, activity and sequence diagrams.

state machine diagrams in SysML are described in a notation that is more restrictive than that of UML, but compatible [20, pp. 541]. A state machine model reacts to events from the environment stored in an event pool. The order in which events in the pool are processed is unspecified. Figures 4 and 5 show examples of state machines from our case study. The state machine in Figure 4 is from the abstract model. The state machine in Figure 5 is for a Device block of the concrete model.

States can be simple or composite. Simple states do not have substates. For instance, Off in Figure 5 is simple, while On is composite. Initial states are depicted as filled circles. Figure 5 shows two initial states: that of the entire state machine and that of the state On.

A state may have three types of behaviour: entry and exit actions, and do activities. Entry actions are executed when the state is activated, exit actions, when the state is exited, and do activities, when the entry action finishes its behaviour. Do activities may either stop or continue indefinitely until a transition interrupts it and exits the state. Figure 5 shows the do activities of the states Undefined, Follower and Leader. These activities store in currentState the new value currentState$+1$ modulo the number nDevices of devices.

A transition connects a source to a target state; it can be triggered by a signal and have a guard. For example, in Figure 5, the transition from the state Off to the state On is triggered by a signal turnOn. The transition from Undefined to Leader takes place whenever the state Undefined is active, its do activity has finished, and the guard currentState == id and nLeaders == 0 and petitionAccepted is true. (This means that a device becomes a leader whenever its petition has been accepted and there are no leaders.)

The state machine in Figure 4 contains a single state with four transitions: two represented implicitly inside the state, triggered by turn_on and turn_off, and two shown explicitly, both triggered by tick. The former model the activation and deactivation of devices by adding and removing their identifiers from Active.

Two transitions are triggered by the signal tick. The first is executed when there are active devices, but either there is no leader or it is no longer active; it specifies that Elected must be updated to contain exactly one of the identifiers of the active devices. The second transition is executed if there is a leader but no active devices, and specifies that Elected must be emptied.

The state machine depicted in Figure 5 shows how a device decides whether it is a leader. Since there is no centralised control, each device maintains information about all others. A device communicates with the others in cycles, in which it receives information from the others and broadcasts its data. At the end of each cycle a device knows the roles of all devices.

Activity diagrams are based on classic flow charts and are used, for example, for low-level modelling of the detailed behaviour of an operation, or high-level modelling of workflows or processes.

An activity diagram has three basic elements: activity nodes, edges, and regions. An activity node represents an action, a control or an object. An action node denotes a behaviour in the activity. In Figure 6, updateDeviceInfo, «Value Specification Action» and updateCurrentState are action nodes. The node named «Value Specification Action» computes the value of an expression (++currentState) mod nDevices that is then sent to updateCurrentState via the output pin s.

Control nodes manipulate the flow of actions, for example, via decisions, forks and joins. A decision is,

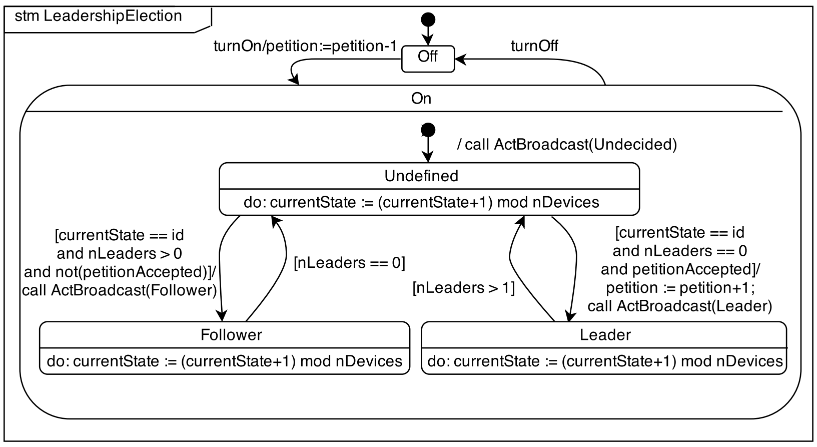

stm LeadershipElection
turnOn/petition:=petition-1
Off
turnOff
On
/ call ActBroadcast(Undecided)
Undefined
do: currentState := (currentState+1) mod nDevices
[currentState == id and nLeaders > 0 and not(petitionAccepted)]/ call ActBroadcast(Follower)
[nLeaders == 0]
[nLeaders > 1]
[currentState == id and nLeaders == 0 and petitionAccepted]/ petition := petition+1; call ActBroadcast(Leader)
Follower
do: currentState := (currentState+1) mod nDevices
Leader
do: currentState := (currentState+1) mod nDevices

Fig. 5: A state machine diagram.

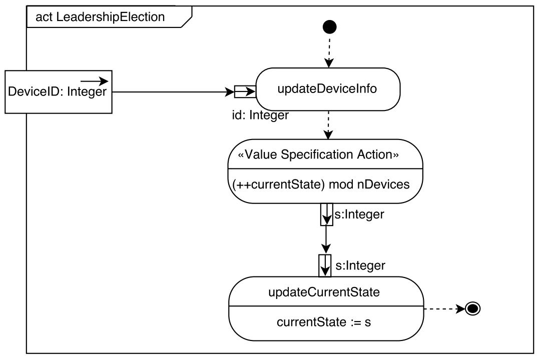


Fig. 6: Example of an activity diagram.

typically, an if-then-else choice. A fork creates parallel flows, and a join defines a point where they unite.

Object nodes represent data used by an activity: inputs and outputs to the activity or to its nodes. For example, in Figure 6, DeviceID is an object node taken as input by the node updateDeviceInfo via the pin id.

Edges can be of two types: control flow or object flow. A control flow defines when and in which order the actions run. Control flows are shown as dashed arrows. An object flow describes how inputs and outputs flow between actions. Object flows are depicted as solid arrows; in Figure 6 the DeviceID object is passed to updateDeviceInfo and ((++currentState) mod nDevices) is sent to the action updateCurrentState.

Activities use a token semantics to control the flow of execution. A node can be executed only when all its inputs have received tokens, and, once it finishes its behaviour, it provides tokens on its outputs. Some nodes create tokens (for instance, an initial node provides tokens on its outputs), while others remove tokens (for instance, a flow final node consumes tokens that arrive on it). An activity finishes its execution when there are no tokens flowing through it.

Sequence diagrams are used to define and present UML/SysML Interactions [21,1]. They describe operational scenarios of a system with an emphasis on sequences of interactions between objects. This is achieved through the use of lifelines. Each participant of the diagram, typically, an instance of a block, possesses a lifeline, so that we can represent a message-exchange order.

The sequence diagram in Figure 7 presents a scenario of our example where a user turns on three devices and each of them notifies the bus that the leader is undefined. The user is depicted as an actor, while the three devices and the bus are instances Dev1, Dev2, Dev3, and bus of a block. A lifeline is represented by a dotted vertical line under each participant.

Participants communicate via messages. For example, Figure 7 shows the actor sending messages to turn on devices. Messages are sent in sequence along a participant lifeline. So, the first message sent by the actor is to Dev1, the second to Dev2, and the third to Dev3.

Messages can be of three types: asynchronous (open arrow), synchronous call (closed arrow), or reply from a synchronous call (dashed arrow). All messages shown in Figure 7 are asynchronous.

A lifeline can include a state invariant: a constraint on the blocks. If the constraint holds, any message-exchange order just established is a valid scenario of the system, otherwise, it is forbidden. State invariants define properties of the system in terms of the attributes of a block. At the bottom of the lifeline Dev3 in Figure 7, there is an invariant Dev3.claim == Undecided, which verifies that the value of the attribute claim of

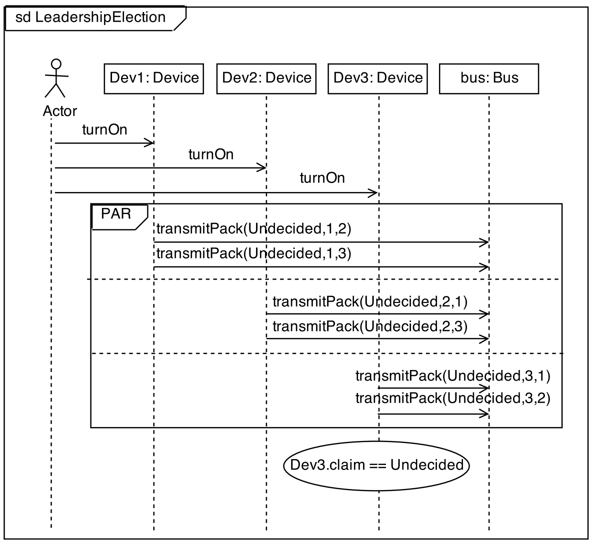

sd LeadershipElection
Actor
Dev1: Device
Dev2: Device
Dev3: Device
bus: Bus
turnOn
turnOn
turnOn
PAR
transmitPack(Undecided,1,2)
transmitPack(Undecided,1,3)
transmitPack(Undecided,2,1)
transmitPack(Undecided,2,3)
transmitPack(Undecided,3,1)
transmitPack(Undecided,3,2)
Dev3.claim == Undecided

Fig. 7: Example of a sequence diagram.

Dev3 is Undecided after it has been turned on and all its packages have been transmitted.

A sequence diagram can include an InteractionUse element to refer to another sequence diagram, so that part of its definition is provided by the referenced diagram. In this way, a sequence diagram can be used several times in the definition of others.

Message exchanges can be grouped inside combined fragments that describe operators like parallel composition, conditional, and loops. Figure 7 shows a parallel combined fragment PAR.

A complete SysML model contains several elements modelled and visualised through diagrams. Each diagram provides a different point of view of a system that must complement and be consistent with the others. State machine and activity diagrams describe behaviours using the attributes and calling the operations of blocks. A state machine diagram may call an activity. state machine and activity diagrams may call an operation and send a signal to a block. Finally, a sequence diagram describes a scenario where exchanged messages between blocks are used to represent either operation calls or transmission of signals.

Our semantics defines an integrated view of the various elements defined by these diagrams.

2.2 CML

CML is a formal language based on VDM [22] (Vienna Development Method), for data modelling, and CSP [23] (Communicating Sequential Processes), for interaction modelling, with support for refinement. We introduce CML through excerpts from the semantics of the SysML model of the leadership-election protocol. A comprehensive description is in [24], and we explain any extra notation as needed later on.

Like in CSP, a system and its components are modelled in CML as processes that communicate via channels. A core concept is refinement, a behaviour-preserving relation between processes. If a CML process `P` is refined by another process `Q`, written $P \sqsubseteq_P Q$, then `Q` can only engage in interactions that are possible for `P`, and can only deadlock or livelock, when `P` can. Refinement is the main reasoning technique available for CML, and we exploit it in our work to reason about SysML models via their CML semantics defined here.

A CML specification is formed of a sequence of paragraphs that specify types, values, functions, channels, and channel sets to support the specification of processes. For example, in our semantics, each block of a SysML model is identified by a value of the type `ID` defined below as a sequence of tokens.

```
types
    ID = seq of token ...
```

The primitive type `token` supports only equality. Record types, like `turnOnS` below, are declared for operations and signals. In this example, the records of the type `turnOnS` have a single field `$id`.

```
    turnOnS :: $id: token ...
```

Union types like `S` below are used to gather these records.

```
    S = ... | turnOnS | turnOffS
```

Values are constants used in the model. For instance, the value `empty_bag` in our example is defined below as the empty map. We represent bags as functions that associate a value with its multiplicity.

```
values
    public empty_bag = {|->}
```

Similarly, functions may be declared to manipulate values. For instance, the function `in_bag`, which checks if a value is in a bag, is declared as follows.

```
functions
  public in_bag: token * Bag -> bool
  in_bag(o,b) == (o in set dom b) and b(o) > 0
```

For a token `o` and a bag `b`, we have that `in_bag(o,b)` is true if, and only if, `o` is in the domain of `b` and is associated with a value `b(o)` greater than zero.

Communication in CML takes place through channels that may communicate zero or more values. In our example, interaction with a device takes place via operation calls and signal transmissions. These interactions are modelled by channels like Device_sig declared below.

```
channels
    Device_sig: nat*ID*ID*S ...
```

It communicates a natural number that identifies a signal transmission, the identifier of the block transmitting the signal, the identifier of the target of the transmission, and the signal, represented by an element of S.

A set of communications can be grouped in channel sets, which can be used to make communications internal or to define synchronisation requirements in a parallelism. In our example, the communications allowed by the model of the part device(1) of the block SoS are specified by the following channel set.

```
chansets
cs_Device1 =
  {|Device_sig.n.o.([mk_token("device(1)")])
                 | n: nat, o: ID|}   ...
```

This set includes the events associated with the channel Device_sig, where the first parameter n is any natural number, the second is any identifier o, and the third is the identifier [mk_token("device(1)")] of device(1).

A process declares a state, auxiliary actions, and a main action at the end that specifies its behaviour. For example, the process simple_Device below models the basic aspects of the block Device to provide access to attributes, signals and operations.

```
process simple_Device = $id: ID @ begin
    state devices: DeviceSequence
        id: int
        ...
        enabled: Bag := empty_bag
```

The state of a process is encapsulated, that is, not visible outside the process. Above, the state devices of simple_Device contains one component for each property (id,...) of the block Device, plus the extra component enabled, which records the operations that have been called but not completed yet.

Actions are defined using a combination of VDM data operations and CSP operators. For example, the auxiliary action Device_state of simple_Device shown below models access to attributes. It is a recursive action.

```
actions Device_state = mu X @ (
    Device_get_id?o!$id!id-> Skip
    []
    Device_set_id?o!$id?x -> id := x
    []
    ...
    ); X
```

Recursion is introduced by the construct mu X @ F(X), which defines the name X for use in the action F(X) to stand for recursive calls. In each iteration of the recursion in Device_state, an external choice ([]) of communications, representing reads or writes to the state components, is offered to the environment of the process. Communications are specified using CSP-like prefixings, in which a communication like Device_get_id?o!$id!id is followed by an action. In communications, the parameters of the channel are decorated with ? or ! to denote inputs or outputs. The action Skip describes immediate termination; id := x assigns the input value x to id.

In the definition of a process, auxiliary actions, if any, are composed to specify a main action that comes at the end to define the overall behaviour of the process. For simple_Device, the main action is as shown below.

```
@ Device_state
    [||{devices, id, ...} | {enabled}||]
  Device_requests
end
```

It is specified as the parallel composition of auxiliary actions Device_state and Device_requests without synchronisation, that is, in interleaving, as defined by the operator [||...||]. Any parallel composition must partition the state between the parallel actions to avoid data races. In this case, Device_state has write access to all the components corresponding to attributes of the block Device, and Device_requests has access to enabled. They can both read the initial value of all components.

There is also a notion of refinement for actions; we write $A \sqsubseteq_A B$ when an action A is refined by an action B. Like for processes, $A \sqsubseteq_P B$ means that B can only engage in interactions that are possible for A, and can only deadlock or livelock when A can. Overall, refinement corresponds to reduction in non-determinism, but in the case of processes, the data model is hidden.

A process may also be defined in terms of other processes. For instance, in the CML model of the leadership-election protocol, simple_Device is composed with another process controller_Device to define a new process with the added capability of storing received events in a pool and managing the pool.

Models evolve during development, and sometimes, practitioners need to verify that a new model conforms to an earlier, perhaps more abstract, version. Refinement allows the verification that the behaviour of the new model conforms to that of the abstract model in the sense previously described, where the notion of behaviour includes interactions, deadlocks, and livelocks. In addition, we can use refinement to analyse a model by comparing it to a description of properties, that is, a validation model, rather than another system model.

Refinement can be used to reason about SysML models of a system once the SysML elements have an integrated formal semantics for refinement.

3 Modelling patterns

One of the main features of SysML, namely its flexibility, hinders the task of enriching it with a formal semantics. Certain uses of the notations are not addressed in the informal semantics, and even explicitly allowed omissions (like operation definitions) can lead to incomplete SysML models that do not have meaningful CML models. To avoid these problems, we propose guidelines of usage for SysML. In what follows, we describe these guidelines and illustrate them using the leadership-election case study.

Overall, the guidelines maximise the definedness of a SysML model and are similar to constraints imposed by most tools to enable automatic code generation. We have four types of guidelines: entity-definition, instance-level, action-language assumption, and simplification assumptions. The action-language assumptions fill in an omission regarding the notation in which SysML actions (for instance, entry actions in state machine diagrams) are expressed, and the simplification assumptions identify a manageable, yet comprehensive, subset of SysML models to characterise our approach.

We now describe each type of guideline.

A. *Entity-definition*:
   1. operations are defined;
   2. composite blocks contain only parts, ports and connectors (that is, do not have attributes, operations, or signals);
   3. associations are realised by connectors between parts;
   4. aggregations are not used (since the distinction to an association seems rather ambiguous and weak);
   5. connectors between ports, as opposed to those that realise associations, are not typed.

B. *Instance-level*:
   1. composite blocks have their structure specified by internal block diagrams;
   2. multiplicities with the * character are not allowed;
   3. all blocks in a composition appear in the associated internal block diagram in numbers compatible with their multiplicities;
   4. ports are connected only to other ports.

C. *Action-language assumption*: actions are expressed in a subset of CML used to define data operations, enriched with statements for sending a signal or calling an operation of another block through a port or association, and for calling an activity;

D. *Simplification assumptions*:
   1. the behaviour of a block is specified by a state machine (or not at all);
   2. activities are used to model operations and auxiliary behaviours;
   3. sequence diagrams are used to model scenarios;
   4. activity and sequence diagrams must not be recursive (mutually or otherwise);
   5. operation calls are synchronous.

While our entity-definition guidelines disallow composite blocks that contain state machines and activities, it is possible to model such blocks by adding a non-composite part block that holds the state machines and activities. As for the restriction on connection types, since a connection between ports is specified by the provided and required interfaces of the connected ports, typing information on the connector is unnecessary.

The instance-level guidelines disallow multiplicities with upper bound of *, because they add the possibility of dynamic creation of blocks, which leads to intractable CML models due to state explosion. A port can be connected directly to a part, but this feature is not treated in this version of the formalisation.

Regarding the action-language assumption, it does not allow channel-based communication to model interactions as in CML. Instead, attributes, signals and operations define the services provided by an application. SysML actions are, therefore, data operations that explain how the state embedded in blocks and other operations can be used to specify the services. We could use any data language to specify the actions, and we choose the CML data language for convenience. The really important point is that the language is defined.

These guidelines capture the way in which the relevant diagrams are commonly used. More detailed descriptions and examples can be found in [25].

Besides these guidelines, the designer must use the subset of the UML/SysML abstract syntax covered by our semantics. Figure 8 indicates the constructs covered for the different model elements. We note that, together, the CML model generator and the CML parser enforce the use of our guidelines.

The abstract and concrete models of the leadership-election protocol in Section 2.1 respect the guidelines presented. The block-definition and internal block diagrams satisfy the entity-definition (A) and instance-level (B) guidelines. For instance, as required by guideline A1, all the operations of the block LE SoS in Figure 2 are defined using the state machine diagram in Figure 4. Moreover, as required by B2, we assume a fixed number of devices: three.

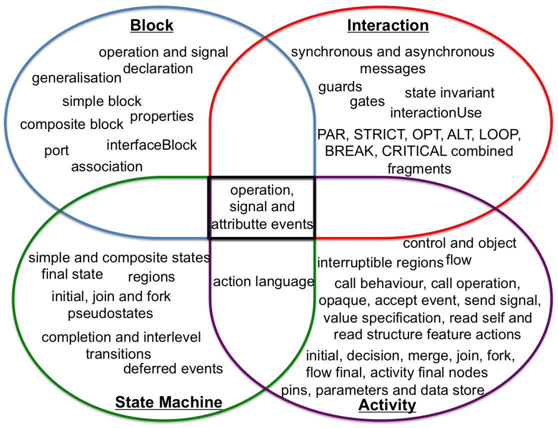

Block
operation and signal declaration
generalisation
simple block
composite block
properties
port
interfaceBlock
association
Interaction
synchronous and asynchronous messages
guards
gates
state invariant
interactionUse
PAR, STRICT, OPT, ALT, LOOP, BREAK, CRITICAL combined fragments
operation, signal and attributte events
simple and composite states
final state
regions
initial, join and fork pseudostates
completion and interlevel transitions
deferred events
action language
control and object flow
interruptible regions
call behaviour, call operation, opaque, accept event, send signal, value specification, read self and read structure feature actions
initial, decision, merge, join, fork, flow final, activity final nodes
pins, parameters and data store
State Machine
Activity

Fig. 8: Constructs covered by our semantics.

Regarding the action-language assumption, the do activities actions of the states Undefined, Leader and Follower of the state machine in Figure 5 are specified as CML assignments. In the activity shown in Figure 6, the action updateCurrentState is opaque: its behaviour is not defined using diagrams. Instead, we use the CML-based data language also following our guidelines.

Finally, we have sequence diagrams that specify scenarios (guideline D3). Figure 7 shows a possible initialisation of the system where three devices are turned on and they send their data in parallel to the network.

For didactic reasons, most of the diagrams presented are simplified. The complete model is available in [26].

4 Model integration

While a SysML model can be visualised via diagrams, its actual representation is a set of interconnected elements conforming to the metamodel specified in [1,21]. For instance, a block-definition diagram is not recorded explicitly in the SysML model; it is just a means of declaring blocks of the model. Furthermore, different diagrams may contribute to the same model element. For example, if a block-definition diagram introduces a block `B` with a property `n`, and an internal block diagram adds a port `p` to the block `B`, in the model, `B` contains both the property `n` and the port `p`.

Accordingly, a SysML model that follows our guidelines is, essentially, a collection of blocks, state machines, activities, and interactions. The parts of composite blocks are structured using connectors, simple blocks have state machines specifying their behaviours, operations are specified either by the block's state machine or by an associated activity, and activities and state machines use CML data operations.

Typically, a SysML model contains several simple and composite blocks as components, with the system as a whole modelled by a composite block. Since the CML construct used to represent systems and their components is a process, blocks, state machines, activities and ports are all modelled as CML processes.

The services defined by a SysML model are the operations, signals, and public attributes of its blocks. These are all represented in CML by channels that are used by the CML process that models the system for communication with the environment, and that, therefore, characterise the interface of this CML process.

In this section, we present our approach to defining the CML model corresponding to a SysML model integrating several diagrams as described above. We first address in Section 4.1 the case of a system defined just by a simple block; we call such basic models non-hierarchical. An example is the abstract model of the leadership-election protocol. Models containing several simple blocks can be handled by considering each of the blocks (and associated diagrams) in isolation. Next, we explain how our approach extends to consider the integration of elements in hierarchical models, which include composite (as well as simple) blocks. The modelling approach in this case builds on that for non-hierarchical models. Finally, in Section 4.3, we discuss the models of interactions, and how they define properties of the overall model of a system.

4.1 Non-hierarchical models

Figure 9 gives an overview of the architecture of the CML models that we use to capture the semantics of SysML models containing a single simple block. They are defined by a composition of interacting processes, each of which models the individual elements (state machines, activities, ports, and so on) of the SysML model.

Figure 9 depicts these processes as nodes. Five nodes are shown: a block B, a state machine S1, a port p1, and activities A1 and A2. The required cooperation between them is indicated as arrows annotated with the set of channels on which the processes synchronise. The modelling strategy described allows elements to be modelled and analysed independently. The compositionality of parallelism (and all other CML constructs) with respect to refinement ensures that established properties are preserved by the whole model.

The services of a SysML block are characterised by its operations, signals, and attributes. As in the case of a SysML model, this interface is represented in CML by channels, which define the interface of the process that models the block. We have channels to represent operation calls, signal transmissions, and get-and-set operations that allow access to attributes, and channels that

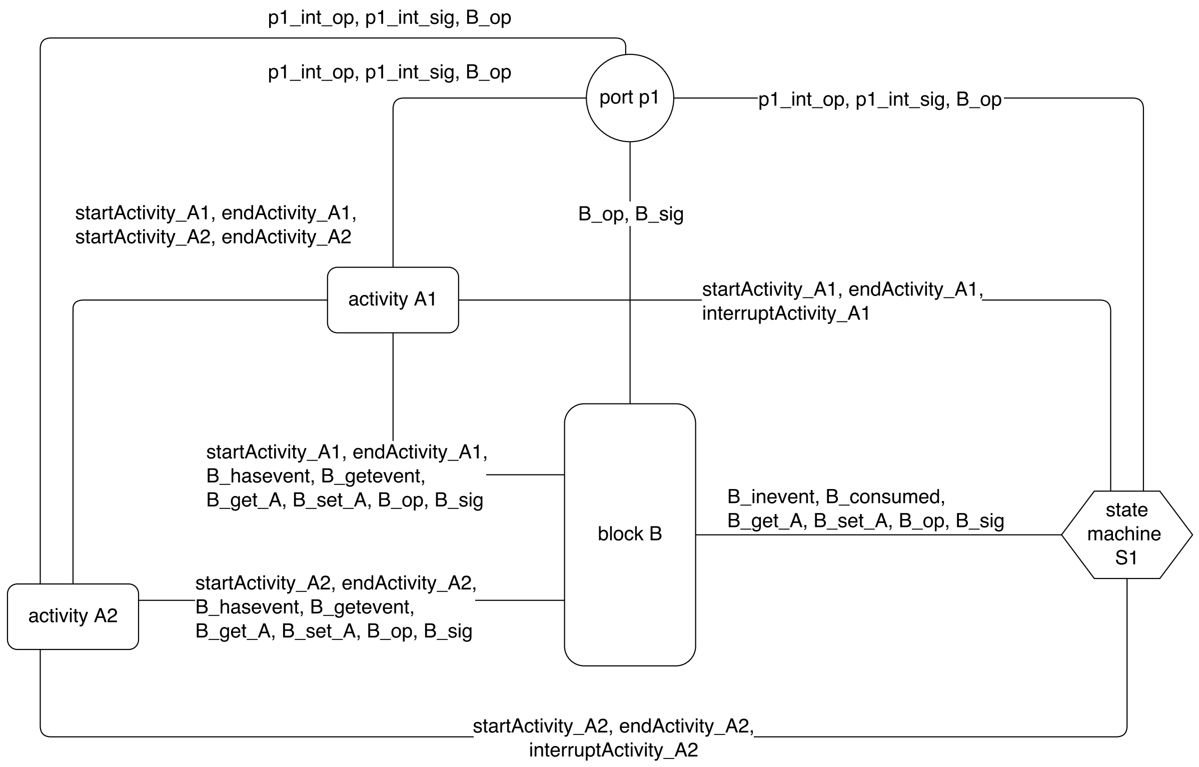

p1_int_op, p1_int_sig, B_op
p1_int_op, p1_int_sig, B_op
port p1
p1_int_op, p1_int_sig, B_op
startActivity_A1, endActivity_A1, startActivity_A2, endActivity_A2
B_op, B_sig
activity A1
startActivity_A1, endActivity_A1, interruptActivity_A1
startActivity_A1, endActivity_A1, B_hasevent, B_getevent, B_get_A, B_set_A, B_op, B_sig
block B
B_inevent, B_consumed, B_get_A, B_set_A, B_op, B_sig
state machine S1
startActivity_A2, endActivity_A2, B_hasevent, B_getevent, B_get_A, B_set_A, B_op, B_sig
activity A2
startActivity_A2, endActivity_A2, interruptActivity_A2

Fig. 9: Overview of the integration points of the semantics of a SysML model.

support the evaluation by state machines and activities of a SysML event, which can be related to an operation call or a signal reception.

A state machine accepts requests to process operations and signals of its block. So, the interface of the CML process corresponding to a state machine includes a channel that accepts a request to react to events corresponding to an operation call or a signal reception, and another to provide responses regarding the realisation of the event. For instance, in the CML model of the example displayed in Figure 5, the communications between the Device block and its state machine happen through the channels `Device_inevent` and `Device_consumed`. In general, as shown in Figure 9, a channel `B_inevent` is used by the block process `B` to send (a representation of) a SysML event for treatment by the state machine, and `B_consumed` is used by the state machine process to indicate the result of evaluating such an event.

Additionally, in order to handle these SysML events, the state machine may access the block's attributes. This interaction between a block B and a state machine S1 is represented by the channels `B_get_A` and `B_set_A` shown in Figure 9. They allow the state machine to read from and write to the attribute A of B. For example, in Figure 5 the Device state machine accesses the block's attributes, for instance, in the action petition:=petition-1. In the corresponding CML process, these attribute accesses are carried out through the channels `Device_get_petition` and `Device_set_petition`.

Besides accessing its block's attributes, the state machine can call operations and send signals to that block. This is achieved via the channels `B_op` and `B_sig` indicated in Figure 9 for communication between B and S1. These channels can be used by the environment to request services of the block, and by other blocks, state machines, and activities to use those services.

We observe that the channels detailed so far are in the interface of both the block and state machine processes. This ensures that when they are composed in parallel, the block process can send SysML events to be treated by the state machine process, and similarly, the state machine can request services of the block.

The interface of a CML process that models an activity contains channels that can be used to start the execution of the activity, interrupt it, and indicate its termination, and channels that can be used to call other activities and wait for their termination, access block operations, signals and attributes, and check the occurrence of an event in a block and request it. For instance, the state machine LeadershipElection in Figure 5 calls the activity ActBroadcast with Leader as a parameter. This is modelled by communications on the channels named `startActivity_ActBroadcast`, for initialisation, `interruptActivity_ActBroadcast`, for interruption, and `endActivity_ActBroadcast`, for termination.

Activities can also interact with a block to access its attributes, send signals, and call operations. In CML, these interactions take place using the same channels used by state machines as described above. Namely, as shown in Figure 9, the process for an activity A1 interacts with that for a block B via the channels `B_hasevent` and `B_getevent` to search and obtain events from the block, `B_op` and `B_sig` to access operations and signals of the block, `B_get_A` and `B_set_A` to access an attribute A, and `startActivity_A1` and `endActivity_A1` to allow the block to call the activity and wait for its termination. Additionally, an activity may call another activity A2 through the channels `startActivity_A2`, `endActivity_A2`.

The interface of a process for a port p1 has channels that allow sending and receiving operation calls and signals. The channels in this interface are divided in two sets, which we call internal interface and external interface. The channels in the internal interface are used to interact with the process for the block that contains the port as well as with those for its parts, state machines and activities. The names of the channels in the internal interface are of the form `p1_int_op` and `p1_int_sig` as shown in Figure 9. The external interface allows block processes to interact with the port process via the processes for their own ports. It contains channels named `p1_ext_op` and `p1_ext_sig`, which are omitted in Figure 9, and further illustrated in Figure 10.

We note that in the interface of a block process `B`, the names of the channels used to model events corresponding to operation calls and signals are of the form `B_op` and `B_sig`. In the case of a port process, they are `p1_int_op` and `p1_int_sig`. We use different names because a port can restrict the operations and signals that a block may accept or require. For instance, in Figure 3 the port pD communicates only calls to operations and signals described in the interfaces DeviceInterface and BusInterface. So, the CML process that models pD does not include events that do not correspond to these calls to operations and signals in its alphabet.

On the other hand, in the composition of processes for a port and its block, events corresponding to the same operation calls or the same signals need to be identified. For this purpose, we use CML renaming, which allows the renaming of the channels used in a process or action. For instance, the action `c?x -> Skip` waits for a value on the channel `c` and terminates, while the renamed action `(c?x -> Skip)[[c <- a, c <- c]]` is equivalent to `(a?x -> Skip [] c?x -> Skip)`, that is, it waits for a value on either the channel `c` or `a` before terminating.

In our models, this operator is used to rename the channels `p1_int_op` and `p1_int_sig` of the internal interface to `B_op` and `B_sig`, perhaps restricting the possible communications through these channels if p1 is realising specific SysML interfaces as described earlier. This mechanism allows the block and the port processes to interact. In our example, as just mentioned, Figure 1 shows a Device block with a pD port, hence, we rename the port channels `pD_int_op` and `pD_int_sig` to `Device_op` and `Device_sig`, but restrict the accepted synchronisations to those corresponding to the operations and signals described in DeviceInterface and BusInterface.

The interaction between two port processes is also modelled using renaming, but in this case the channels in the external interfaces of the ports are renamed to new channels that model the connector between the ports. This is discussed in the next section.

Finally, the interaction between processes for a port p1, state machines, and activities take place through the channels `p1_int_op` and `p1_int_sig` and `B_op`, where B is the block that contains the port. The first two channels are used by the processes for the state machines and activities to call operations and send signals via the port process, and the third channel to indicate the completion of an operation call that they have handled.

Some of the presented channels are just for internal communication, that is, they are not relevant for the external environment of a block. For example, a block should not access the state machine or the activities of another block directly. The hiding operator of CML is used to make channels internal; it is written `A \\ {|c|}` for an action `A` and channel `c`, whose communications are made not visible externally to `A`. By hiding the channels that a block process uses to communicate with its state machine process, the block process cannot communicate on such channels externally. Also, these internal channels are indexed by the identification of the block's instance, and that prevents the communications being sent to another instance's state machine as well.

Considering the channels shown in Figure 9, we have that the channels `B_inevent` and `B_consumed` related to the state machine processes, channels `B_hasevent`, `B_getevent`, `startActivity_A1`, `interruptActivity_A1` and `endActivity_A1` related to the process for the activity A1, the similar channels for the process for the activity A2, and the channels `p1_int_op` and `p1_int_sig`, related to the process for the port p1, are hidden to the environment of the process for B. Finally, any channels `B_get` and `B_set` related to private attributes of B are also hidden, as are the communications on `B_get` and `B_set` channels that are related to public attributes, but whose source and target are elements of the diagrams themselves. These communications correspond to internal uses (via accesses and assignments) of the attributes of a block, rather than external observations described, for example, in a sequence diagram.

Conversely, the services of a block are represented by the remaining channels, which correspond to signals, operations, attributes with public visibility, and the external interface of ports. In our example, they are `LE_SoS_get_Active`, `LE_SoS_set_Active`, `LE_SoS_get_Elected`, `LE_SoS_set_Elected`, `LE_SoS_op`, and `LE_SoS_sig`. In general, for a process block `B` like that in Figure 9, we have channels `B_op`, `B_sig`, `p1_ext_op`, `p1_ext_sig`, and `B_get_A` and `B_set_A`, for attributes `A` of B with public visibility.

These channels communicate at least three pieces of control data relevant to the interactions. Two of them are identifications of the sender and the receiver of the interaction. For instance, in the leadership-election example, we have three instances of the Device block interacting with one instance of a Bus block, as depicted in Figure 3, and each of the devices can send transmitPack signals to the bus. Each channel `Bus_sig` that communicates the transmitPack signal carries the information of which device sent the signal and the target of the message, which is the instance of the block Bus.

This control information is important for several reasons. Firstly, it is used to ensure correct communication between the CML processes for block instances, which run in parallel and synchronise on these channels. Additionally, with the control information in the communications, the traces of the CML processes represent accurately communications between the different block instances of a system. Finally, it is used to ensure that the reply of an operation call is returned to the specific block process that requested it. The same call can be sent by different elements of behaviour (state machine or activity) in a block. For example, the block B from Figure 9 can call an operation of another block through its state machine S1 or through its activity A1. The event corresponding to the return of this operation call must be sent exactly to the element of behaviour that requested it to allow it to continue.

In the CML model, the identifier of an element is a sequence of tokens that describes the position in the SysML model hierarchy of that element. Thus, when the state machine S1 process makes the call, the identifier of the sender is `<B,S1>`, where `B` is the token representing an instance of the block B and `S1` is that representing the state machine S1. Similarly, when the call comes from the activity process A1, the identifier is `<B,A1>`.

A third piece of control data is needed when the same interaction can occur concurrently inside the same element of behaviour: for instance, two calls to the same operation and with the same target, each originating from two different parallel regions (r1 and r2) of a state machine S1. In order to differentiate the calls, each channel communicates a unique index of the call represented by a natural number. Therefore, the communication corresponding to the call from r1 has a natural number index, while for the call from r2 we use a different index. These indices avoid that the reply to the call from r1 is delivered to the r2 action or vice-versa.

This third piece of control data is not relevant for the environment of the block process that represents the system. It is part of an internal protocol to ensure correct communication between processes and actions. The top-level block that represents the system, therefore, exposes a different version of the channels without this index. This is achieved through renaming. For instance, the LE SOS block process has its operations renamed in `LE_SOS[[LE_SOS_op.m <- LE_SOS_OP | m: nat]]` to define the system model. Communications `LE_SOS_op.m`, where `m` is a natural number representing the index that identifies the communication context, are renamed to `LE_SOS_OP`, which is a version of `LE_SOS_op` without the index. The same strategy is applied to signal channels.

4.2 Hierarchical models

We now address blocks that are structured in hierarchies. Figure 10 depicts the CML processes that model such blocks, and their ports, activities and state machines. In this figure, processes are represented by solid boxes, and the sets of channels that allow them to interact are included in dashed boxes associated with arrows connecting the relevant processes.

In Figure 10, while the block B is simple (and its CML model has the structure illustrated in Figure 9), A is composite and contains a block C as a part. The SysML model to which A belongs is, therefore, hierarchical. The block C may also have a state machine and activities, but C can only communicate with B through the interface of its owner block A. Although it is not shown in this example, our semantics caters for models with hierarchies of any depth; for example, C could have other blocks as its parts.

As indicated in the previous section, in general, as shown in Figure 10, processes for blocks can communicate through channels `A_op`, `A_sig`, `A_get_X`, `A_set_X`, `p_ext_op` and `p_ext_sig`, where `A` is the name of the block, `X` is the name of a public attribute of `A`, and `p` is the name of a port of `A`. As explained, for simple blocks, the channels `p_int_op` and `p_int_sig` corresponding to a port `p` are renamed to model communications between the processes for `p` and for its associated block.

In the case of a composite block, a port is used for communication with its part blocks rather than with state machines or activities. We recall that our guideline A2 requires that composite blocks do not have state

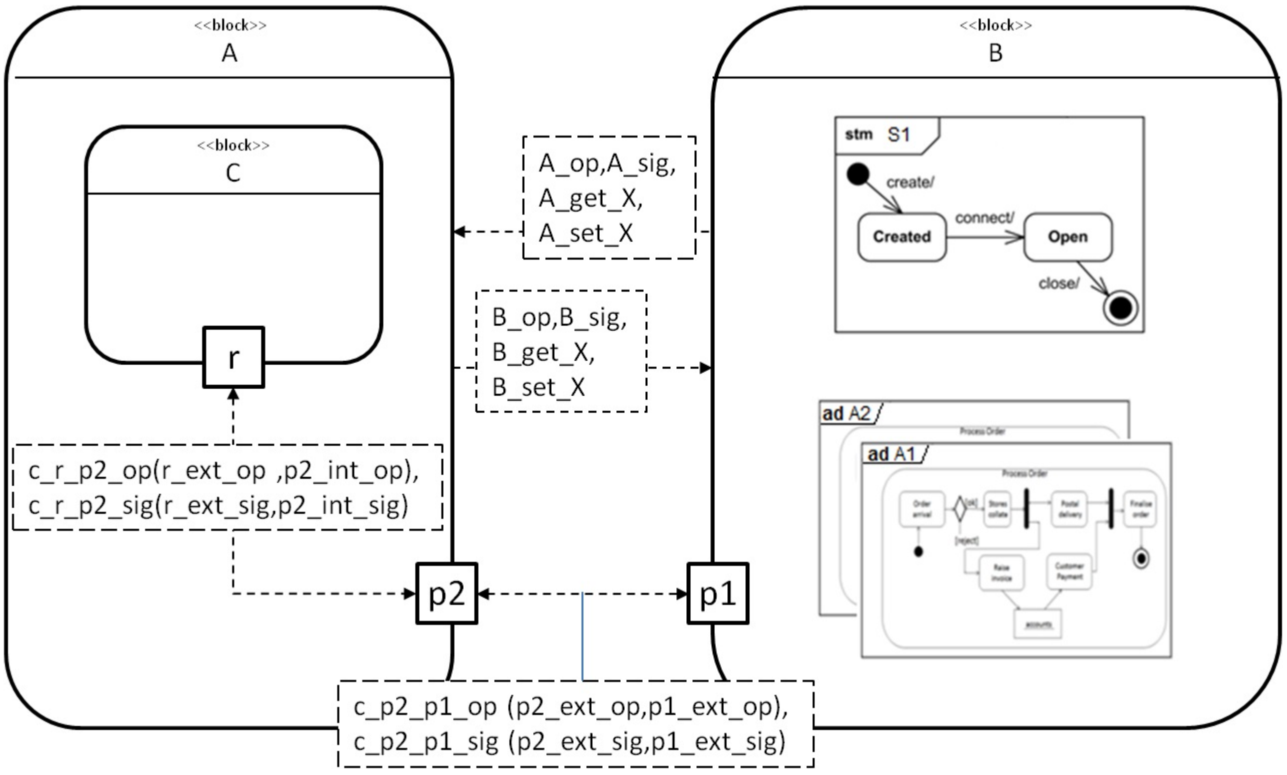


Fig. 10: Overview of the communication between the models of multiple blocks.

machines and activities. So, in this case, the channels `r_ext_op` and `p2_int_op` used by the processes for the ports r and p2 in Figure 10 are renamed to `c_r_p2_op`, and similarly the channels `r_ext_sig` and `p2_int_sig` are renamed to `c_r_p2_sig`. Using this strategy, the block process `A` can relay a message received by the port process `p2` by sending it to the port process `r`.

4.3 Scenarios

According to our guidelines, described in Section 3, sequence diagrams are used to validate the system model.

Since a SysML model is composed of several elements visualised through a variety of diagrams, establishing that they define a consistent integrated view of a system is not an easy task. In addition, finding undesired behaviours is hard due to the lack of tools. To address these issues, our guidelines cater for the possibility of analysis of the SysML model against scenarios described by sequence diagrams. Using the CML models of both the SysML system model and of an interaction, we can compare the traces specified by an interaction against those of the system model through refinement. Thus, the scenario described by a sequence diagram, for example, takes the role of a rule that should be valid for the SysML model. This approach provides a flexible way for validation because sequence diagrams can be constructed using a rich variety of constructors that increases the expressiveness of the validation models.

To perform this validation, we must relate the messages depicted in an interaction to the services of the SysML system model. As said before, these services are operations and signals of the blocks. A sequence diagram defines interactions in terms of the messages exchanged by the blocks, corresponding to these services.

The messages of a sequence diagram are modelled in CML by communications similar to those used in the CML specification of a SysML system model. An additional channel `inv` is used to identify traces of scenarios defined by a sequence diagram as forbidden. This channel is in the interface of the CML model of an interaction, but not in that of the CML system model.

The channels used in the model of an interaction to represent messages have slightly different types when compared to those in the CML system model. In the model of an interaction, the channels communicate identifiers of the source and the target of the messages. In addition, they communicate an index to identify a message in the interaction and an extra tag corresponding to the point where the message is sent (tag `s`) or where it is received (tag `r`). The definition of the CML process for an interaction, like that for a system model, uses renaming to eliminate the index and the tag. Since in-

SysML meta-model element	CML element
Block	Process
Activity	Process
State Machine	Process
Interaction	Process
Port	Process
Connector	Channel
InterfaceBlock	Set of tokens
Operation	Record Type
Signal	Record Type
Event	Communication

Table 1: SysML-CML correspondence

dices and tags are used for internal control, they are not relevant when we compare the CML models of an interaction and a system. The refinement-based analysis strategy for comparison is discussed in Section 6.

This section has provided an integrated view of how the CML model of the different SysML elements relate to each other. This uses the individual semantics provided for each element discussed in the next section.

5 CML generation

We now present our formalisation of the individual elements of SysML. We describe models of blocks in Section 5.1, of state machines in Section 5.2, of activities in Section 5.3, and of interactions in Section 5.4.

Table 1 summarises the correspondence between elements of a SysML model and elements of CML. As discussed in the previous section, SysML elements that exhibit some form of behaviour, namely, blocks, ports, activities and state machines, are modelled by CML processes. Interactions described in terms of sequence diagrams are also represented by a CML process. Connectors, which specify communication links, are modelled by channels. Static elements (that is, without intrinsic behaviour), namely operations and signals, are modelled by record types. Operations are considered static because they specify the message that is sent to blocks, not the behaviour of the operation itself, which is specified by a state machine or activity. InterfaceBlocks, which are collections of static elements, are modelled by sets of tokens.

Our formalisation of SysML models is a denotational semantics: we define functions from the constructs of the SysML metamodel to constructs of the CML abstract syntax. These functions are described by translation rules that take well-defined elements of SysML and output well formed CML components. The types of both the parameters and the return value of the functions are specified in the rules. Table 2 lists the main semantic functions and the elements to which they apply. Since a state machine or activity is defined in the context of a block, the semantic functions for these elements take a block as a second parameter.

Semantic function	**SysML element**
`t_model`	Model
`t_simple_block`	Simple block
`t_composite_block_process`	Composite block
`t_port`	Port
`t_type_operation`	Operation
`t_type_signal`	Signal
`t_statemachine`	State machine and Block
`t_activity_diagrams`	Activity and Block
`t_interaction`	Interaction

Table 2: Main semantic functions for SysML models.

```
t_model(m: SysML model): program =
   "types
          ID = seq of token"
          ...
   for each b in m.AllBlocks do
          if b.isSimple
          then t_simple_block(b)
          else t_composite_block_process(b)
          end if
   end for
   for each int in m.AllInteractions do
          t_interaction(int)
   end for
```

Fig. 11: Translation rule for SysML models.

The function that defines the semantics of a SysML model is `t_model`. It is defined by the application of other semantic functions in a top-down fashion. Figure 11 shows an excerpt of the corresponding translation rule; the complete definition of this and all other rules can be found in [25]. We use a meta-language to define the rules that resembles a simple programming language with `if-then-else` and `for each` commands, function calls, and access to SysML metamodel elements; text between double quotes is explicit CML code.

The function `t_model` takes a SysML model `m` as an input, and defines a CML `program`. As illustrated in Section 2.2, the CML model defines types. For instance, `ID` identifies the sources and targets of messages and is defined as a sequence of tokens; types for the operations, signals and interfaces, omitted in Figure 11, are also defined. Next, we have the definition of the processes for each block as identified in the component `AllBlocks` of `m` (in accordance with the SysML metamodel). If it is a simple block, then we use the function `t_simple_block()`, otherwise, we use `t_composite_block()`

to give the definition. At the bottom of the rule, we have the definition of the processes for interactions: the function `t_interaction()` is used for each interaction, as identified in the component `AllInteractions` of `m`.

In what follows, we give an informal description of the semantic functions.

5.1 Blocks

While Section 4 discusses the interfaces of the block models, here, we discuss the structure of those models.

The result of applying either `t_simple_block` or `t_composite_block_process` to a block B is a process whose state components are the attributes of B and whose behaviour consists of accepting operation calls and signals, which are modelled as communications through channels `B_op` and `B_sig` (see Figure 9).

For a composite block, like A in Figure 10, the CML model reflects its structure. The CML process `A` is defined by the parallel composition of the processes that model the parts of A: a process for C in parallel with processes for the ports r and p2, synchronising as explained in Section 4. The required synchronisation is identified by the connectors in the internal block diagram of A. For instance, the internal representation of SoS in Figure 1 is reflected in the internal block diagram in Figure 3, which illustrates the communication between instances of Device and Bus. Hence, the process for SoS is defined by the parallel composition of three processes for the instances of Device, three for each port of each instance of Device, one for the instance of Bus, and another three for each port of the instance of Bus. The synchronisation is determined by the connectors between the ports pD and pB as they require and provide services of DeviceInterface and BusInterface.

A process that models a simple block is defined by a parallel composition as well. In this case, the parallel processes include those that represent the ports, the state machine and the activities of the block, and two other processes. The first models the handling of operations, signals, and access to attributes. The second models the management of events (received operations and signals) via a pool and an associated queue of deferred events. The event pool records events for later treatment, and the queue of deferred events is where events that are deferred by the state machine are stored.

For instance, the block `B` in Figure 10 is simple: it has a state machine and two activities. Its process is the parallel composition of the processes `simple_B`, which models its services (operations, signals and access to attributes), `controller_B`, which models the management of events, `port_p1`, which models the port p1 and controls the routing of operations and signals in and out of B, `stm_S1`, which models the state machine, and `act_A1` and `act_A2`, which models the activities.

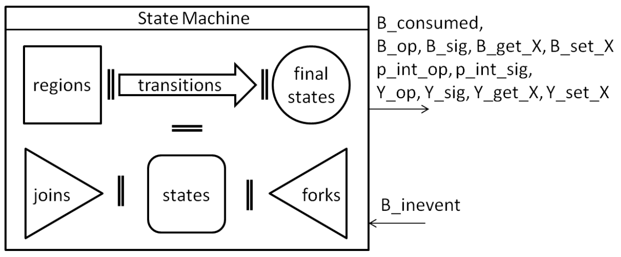


Fig. 12: Overview of the model of state machines.

5.2 State machines

The CML model of a state machine is defined by a single process whose actions model the elements of the state machine as shown in Figure 12. Each state, region, final state, transition (starting from a state), join and fork pseudostate is modelled by a CML action, and all these actions are composed in parallel to define the overall behaviour of the CML process.

As previously indicated, the process that models a state machine is defined by the application of the function `t_statemachine` to that state machine and its block. This process is parametrised by an identifier for the instance of the block to which the state machine belongs. Its behaviour is defined by a recursion that, at each iteration, receives an event through the channel `B_inevent`, communicates it to the actions for its active states and regions, and indicates through `B_consumed` the result of processing the event. This processing may lead to actions being executed, transitions being triggered, and states being activated and deactivated. The model of the execution of actions and the verification of conditions of transitions may involve communications over the channels `B_get_` (to read the state of the block's process), `B_set_` (to modify that state), and `*_op` (to answer and send operation calls) and `*_sig` (to send signals), where `*` stands for any block name.

The CML actions that model elements of the state machine (for instance, states and transitions) define protocols that specify how they interact with each other. This level of granularity allows us to focus on particular elements when analysing the CML model, and to trace back any issues to the original SysML model.

The actions that model the elements of the state machine are coordinated by a `machine` action. It defines the iterative behaviour described above; it initialises the state machine and controls the processing of events. In the initialisation, `machine` requests the actions that model the top regions of the state machine to carry out

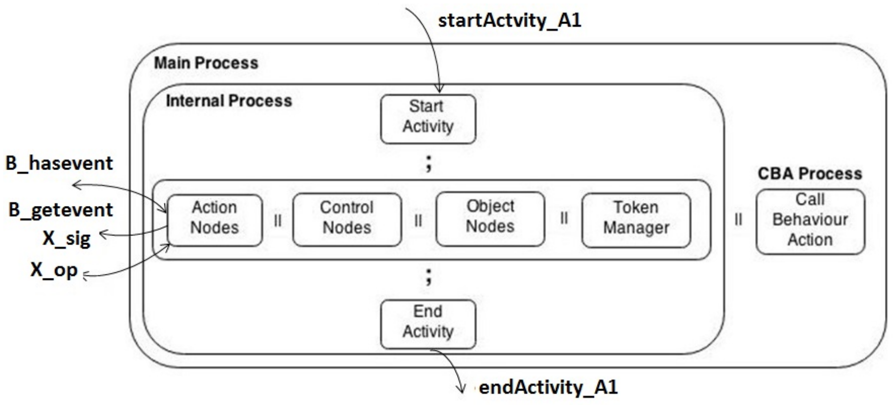


Fig. 13: Overview of the representation of activities in CML.

the behaviour corresponding to entering the regions. (This leads to a request for substates to be entered, and so on, until all regions of an active state are active, and exactly one substate of each active region is active.)

The processing of events is defined by a loop in which, at each iteration, the `machine` action accepts an event from the block (process), sends it to the top-region actions, which define the result of processing the event, sends the result back to the block process, executes the actions for the transitions whose triggers contain the event and whose guards evaluate to true, and waits for the transition actions to finish executing, before recursing. The pattern of interactions between the actions that model states and regions is similar.

5.3 Activities

The translation of the activities of a block is specified by the function `t_activity_diagrams`, which introduces a process defined by the interleaving of the processes for each activity. Each individual activity is translated by `t_activity_diagram`, which uses functions that define two processes: one that models the activity itself, and another that composes in parallel the first process with the processes for other activities that are invoked by call behaviour actions. This allows parallel executions of the same activity, if it can be called by a block and by another activity in parallel, for example.

Figure 13 depicts how an activity process is structured, taking as an example the activity A1 of block B from Figure 9. Each activity is described by means of a process (Main Process) whose behaviour is specified as the parallel composition of a process that models the (internal) behaviour (Internal Process) of the activity itself with other processes for activities that may be used inside this activity as call behaviour actions (CBA Process). If there is no call behaviour action, then the main process is simply the internal process.

In the definition of the internal process, we have a CML action for each node of the activity. Due to the token semantics of activities, there is also a CML action (Token Manager) for managing tokens; it models the control of the ending of an activity according to available tokens. All CML actions are composed in parallel in the main action of the activity internal process.

Control and object flows are established via synchronisations. Actions that model nodes have channels in their alphabet used for this purpose, so that the alphabetised parallelism of these actions enforce the order of execution of nodes depicted in the activity. We provide a CML representation for object nodes, all control nodes, and several actions including call operation, send signal, accept event, opaque, value specification, call behaviour, read self, and read structural feature.

The main action of the internal process is recursive, with each iteration defining the behaviour of one execution of the activity via a sequential composition of three actions (Figure 9). The first, Start Activity, synchronises on the `startActivity_A1` channel to fire the beginning of the activity flow and take any value when needed. For instance, the state machine Leadership Election of Figure 5 calls the activity ActBroadcast using `call ActBroadcast` passing the value corresponding to the claim of the Device. In the CML model, the state machine and the activity processes synchronise on the channel `startActivity_ActBroadcast`.

The Start Activity action is followed by the parallel composition of the CML actions for all nodes of the activity along with Token Manager. The third action, End Activity, communicates that the activity finished along with any output values via the `endActivity_A1`. For instance, when the activity ActBroadcast ends its flow,

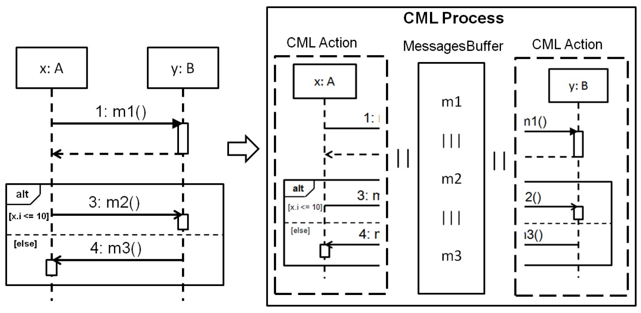


Fig. 14: Semantic representation of interactions in CML.

it synchronises with the process for the state machine LeadershipElection on `endActivity_ActBroadcast`.

5.4 Interactions

The function `t_interaction` defines the translation of an interaction using two other functions. If the interaction has no InteractionUse elements, `t_simple_int` is used. Otherwise, we use `t_complex_int`, which is itself defined in terms of `t_simple_int`. Any interaction referenced by an InteractionUse element is also translated. We note that there can be no mutual dependencies, via InteractionUse elements, in the construction of sequence diagrams. This is not explicitly ruled out in SysML, but it is in our guidelines (D4).

The CML process that models an interaction takes as parameters the identifications of the block instances that either are used in a lifeline or send messages through gates. This makes the model of the interaction as generic as the interaction itself, which is valid for any instances of the block types used in the interaction.

Figure 14 shows how the semantics of interactions is captured by CML elements. The interaction defined by the diagram on the left-hand side is translated to a CML process, whose structure is indicated in the box on the right-hand side. This CML process is defined by the parallelism of three CML actions: two related to the lifelines of the interaction and another to model is MessagesBuffer. Each lifeline is represented by a CML action defined by the sequential composition of CML actions that represent interaction fragments in the lifeline: either message occurrences or combined fragments.

Each message exchange is represented in CML by two communications, one corresponding to the point where the message is sent and another to the point where it is received. The channels used are `B_op` and `B_sig` presented in Section 4.

The CML action `MessagesBuffer` coordinates the message exchange between lifelines of the interaction. It interleaves CML actions for each of the messages.

A message action (for example, `msg_m1`, `msg_m2` and `msg_m3` in Figure 14) synchronises on the sending communication with the sender lifeline and then on the receiving communication with the receiver lifeline. In the action for a sender lifeline, in the case of a synchronous message, the sending communication is followed by the reply communication, so that the sender stays blocked until the reply is received. For asynchronous messages, the sender is ready to proceed with its behaviour after the sending communication. If the constraint of a state invariant yields false, a synchronisation on the extra channel `inv`, which is used specifically in our semantics for interactions, marks an invalid scenario.

The main action of the interaction CML process composes in parallel the lifeline actions together with the `MessagesBuffer` action.

Next we present how our semantics can be used for analysis and refinement of SysML models.

6 Applications of our semantics

This section describes applications of our semantics. The first, covered in Section 6.1, is a refinement calculus for SysML, and the second, in Section 6.2, is the use of the CML techniques to analyse SysML models.

6.1 Refinement in SysML

One of the immediate results that can be obtained from our semantics is a notion of refinement for SysML. For example, we say that a block $\mathfrak{B}_1$ is refined by a block $\mathfrak{B}_2$, written $\mathfrak{B}_1 \sqsubseteq^{\mathfrak{M}}_{Block} \mathfrak{B}_2$, if, and only if, the CML process B_1 that models the block $\mathfrak{B}_1$ is refined by the process B_2 that models $\mathfrak{B}_2$.

Definition 1 (Block refinement) Let $\mathfrak{M}$ be a SysML system model, and $\mathfrak{B}_1$ and $\mathfrak{B}_2$ blocks of $\mathfrak{M}$, then

$$\mathfrak{B}_1 \sqsubseteq^{\mathfrak{M}}_{Block} \mathfrak{B}_2 \Leftrightarrow \\ \quad \mathtt{t\_model}(\mathfrak{M}).B_1 \sqsubseteq_P \mathtt{t\_model}(\mathfrak{M}).B_2$$

The function `t_model` produces the CML semantics of the SysML model and is presented in Section 5. Since a system is specified by a block in a SysML model, block refinement is the main relation that must be verified to establish refinement between SysML models.

As already said, blocks define systems by defining the operations and signals they offer and require, and their public attributes. A refinement $\mathfrak{B}_1 \sqsubseteq^{\mathfrak{M}}_{Block} \mathfrak{B}_2$, therefore, requires the following properties to hold.

1. $\mathfrak{B}_1$ and $\mathfrak{B}_2$ must accept exactly the same public signals and public operations;
2. $\mathfrak{B}_1$ and $\mathfrak{B}_2$ must have exactly the same public attributes (in this way, both blocks offer the same possibilities for setting their values);
3. for each public operation of $\mathfrak{B}_1$, if its return value is non-deterministically chosen from a set S, the same operation on block $\mathfrak{B}_2$ must return a value that is non-deterministically chosen from a subset of S. Moreover, the inputs of each public operation of $\mathfrak{B}_1$ must be accepted by that operation on block $\mathfrak{B}_2$;
4. for each attribute of $\mathfrak{B}_1$, if its value is non-deterministically chosen from a set S, the same property on the block $\mathfrak{B}_2$ must have a value non-deterministically chosen from a subset of S.

In our example, it is possible to show that, in a scenario of stability (that is, a period where no devices turn on or off and the election can terminate) the abstract system model is refined by the concrete model we have presented. This refinement holds because the abstract and concrete interfaces are the same and, during stability, the abstract LE SoS selects non-deterministically a leader, while the concrete SoS chooses the leader deterministically using internal communications to share information among the distributed devices.

To establish refinement of blocks compositionally, it is necessary to define refinement relations for the different elements of SysML (state machines, states, transitions, activities, and so on). In what follows, we discuss the formalisation of these relations in terms of the CML semantics of a SysML model and CML refinement.

The notions of refinement for state machines and several other elements of a SysML model are defined in a way similar to that used in Definition 1. Such direct lift of the CML notion of process refinement, is not possible for all elements, though.

For instance, refinement of states is defined in terms of CML action refinement, and requires consideration of behaviours involving substates, subregions and internal transitions. The hierarchical structure of a state machine is not represented in the structure of the parallel composition that defines the main action of the state machine process, though (see Figure 12). All actions for all components of the state machine are auxiliary actions of its process, irrespective of their position in the state machine. It is the alphabets of the actions in the parallel composition that specify on which channels the actions synchronise to execute the protocol that defines the interaction between states and substates, states and transitions, and so on. In other words, instead of reproducing the hierarchical structure of the state machine in CML, we use the alphabets to restrict the interactions to the appropriate group of actions (for instance, those for a state and its substates), and hide all interactions that are internal to the state machine after all actions are composed. This is necessary due to certain features of state machines, such as interlevel transitions, which break the simple hierarchy of states and substates by allowing one element to affect the execution of elements at unrelated levels of the hierarchy.

When comparing two states, however, it is necessary to consider just the aspects of the protocol that affect other parts of the state machine, for instance, transitions reaching and leaving the state as well as any actions such as operation calls and assignments. Other aspects of the protocol that affect only the subregions and substates of the states being compared, such as communications that allow a transition to exit a substate and enter another, should be considered internal.

Accordingly, informally, a state $\mathfrak{s}_1$ is refined by another state $\mathfrak{s}_2$, written $\mathfrak{s}_1 \sqsubseteq^{\mathfrak{M}}_{State} \mathfrak{s}_2$, if, and only if, the overall (external) behaviour of the action for $\mathfrak{s}_1$ is refined by the overall behaviour of the action for $\mathfrak{s}_2$. The action for a state $\mathfrak{s}$ includes those for the subregions, substates, and transitions, but with channels used to establish the internal behaviours hidden.

Formally, the definition of state refinement uses the following syntactic functions that can be easily defined for any SysML model: *reg*, a function from a state to the set of subregions of that state, *st*, a function from a states to the sets of substates of that state, *trans*, a function from a state to a set of transitions whose source or target, but not both, is the state or one of its substates, *int*, a function from a state to a set of transitions whose source and target are the state or one of its substates, and *name*, a function from a state, region or transition to the name used in CML for the action that models the given element. Definition 2 characterises the behaviour of a state $\mathfrak{s}$ via a function *behaviour* that takes a model $\mathfrak{M}$ and a state $\mathfrak{s}$, and specifies the action that combines the behaviours of the substates, subregions and internal transitions of $\mathfrak{s}$ in parallel, and hides the channels used to define the internal protocol of $\mathfrak{s}$.

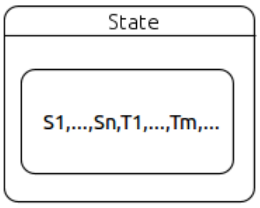
Definition 2 (State refinement) Let $\mathfrak{M}$ be a SysML model, and $\mathfrak{s}_1$ and $\mathfrak{s}_2$ be states of $\mathfrak{M}$, then

$$\mathfrak{s}_1 \sqsubseteq^{\mathfrak{M}}_{State} \mathfrak{s}_2 \Leftrightarrow behaviour(\mathfrak{M}, \mathfrak{s}_1) \sqsubseteq_A behaviour(\mathfrak{M}, \mathfrak{s}_2)$$

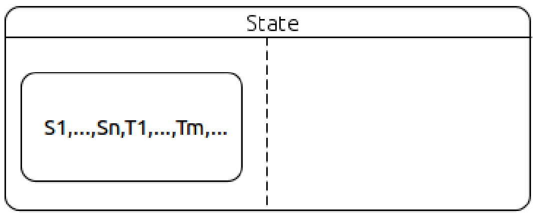
where
$behaviour(\mathfrak{M}, \mathfrak{s}) =$

```
 (|| e in set $(\{\mathfrak{s}\} \cup reg(\mathfrak{s}) \cup st(\mathfrak{s}) \cup trans(\mathfrak{s}))$ @
    [chanset_$name(e)$] t_model($\mathfrak{M}$).$name(e)$
  )\\ dunion {e in set $(reg(\mathfrak{s}) \cup st(\mathfrak{s}) \cup int(\mathfrak{s}))$  @
              ichanset_$name(e)$
           }
```

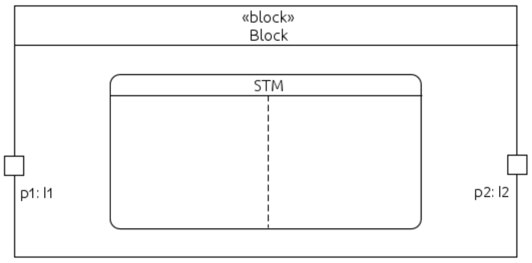
Law 1 Region introduction.



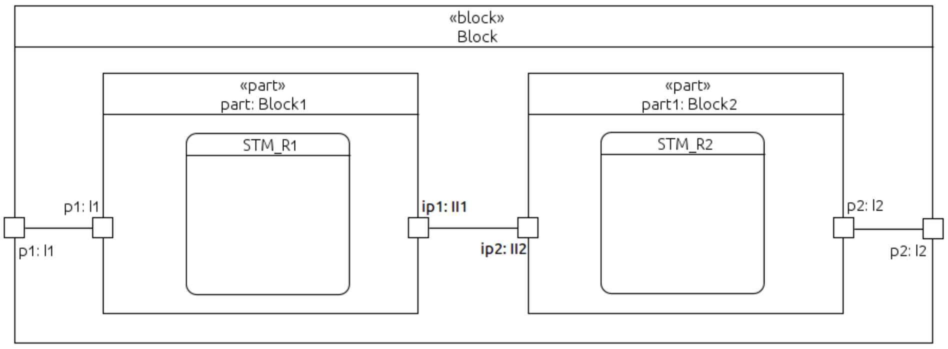
$\sqsubseteq^{\mathfrak{M}}_{State}$



Law 2 Block decomposition.



$\sqsubseteq^{\mathfrak{M}}_{Block}$



provided

1. $I1 \cap I2 = \emptyset \wedge I1 \cup I2 \subseteq Block$;
2. $triggers(R1) \cap triggers(R2) = usedV(R1) \cap usedV R2 = \emptyset$;
3. $provided(I1) \cap trigger(R2) = required(I1) \cap used(R2) = \emptyset$;
4. $provided(I2) \cap trigger(R2) = required(I2) \cap used(R2) = \emptyset$.

where

1. $Block1 \cap Block2 = \emptyset \wedge Block1 \cup Block2 = Block$;
2. $provided(II1) = Block1 \cap used(R2) \wedge required(II1) = Block2 \cap used(R1)$;
3. $provided(II2) = Block2 \cap used(R1) \wedge required(II2) = Block1 \cap used(R2)$.

Intuitively, state refinement allows the internal structure of a state to be modified as long as the observable behaviours (for instance, operation calls) are preserved.

The channel set $\texttt{chanset\_}name(e)$ identifies all the communications on which the action for a SysML element e can engage. By considering all elements in $\{\mathfrak{s}\} \cup reg(\mathfrak{s}) \cup st(\mathfrak{s}) \cup trans(\mathfrak{s})$, we characterise all communications involved in the actions that model the execution of $\mathfrak{s}$ as a whole. The set $\texttt{ichanset\_}name(e)$, on the other hand, identifies the subset of $\texttt{chanset\_}name(e)$ containing only those communications that are used to control the interaction of e with other state machine elements. This excludes, for example, communications to access attributes and call operations. Above, by considering all elements in $reg(\mathfrak{s}) \cup st(\mathfrak{s}) \cup int(\mathfrak{s})$, we hide the communications related to the internal protocol of $\mathfrak{s}$. It is worth mentioning that $int(\mathfrak{s}) \subseteq trans(\mathfrak{s})$ and, only the channels associated with the transitions in $int(s)$, which are internal to $\mathfrak{s}$, are hidden.

It is possible to refine our abstract model of the leadership-election protocol to obtain a concrete model. To express all laws that are needed, we have to extend SysML to include some concepts central to refinement, like hiding. We describe below, however, a viable refinement strategy with three phases. The first phase introduces the Bus block in Figure 1. The second phase introduces parallel regions into the state machine in Figure 4, and parallelises the election algorithm using these regions and the Bus block to allow communication between the regions. Finally, the third phase breaks the block LE SoS into a set of Device blocks, introducing the concrete architecture depicted in Figures 1 and 3.

We illustrate the definition and use of SysML refinement laws through Laws 1 and 2, which are applied in the second and third phases. The first introduces parallel regions that are later completed with distributed behaviours, and the second decomposes a block with parallel behaviour into two separate blocks.

Law 1 applies to a composite state with a single region (and any number of substates and transitions). It can be used to refine such a state into a composite state with two regions: the first is the original region, and the second is empty. This refinement holds because the two regions are executed in parallel, and the empty region does not introduce new behaviours observable outside the composite state.

In our example, as shown in Figure 15, Law 1 can be used to introduce an empty region in parallel with another containing Atomic State. Further refinements can enrich the empty region with the behaviours of an individual device. At the end of the second phase, a new parallel region has been introduced for each device, and the (centralised) behaviour has been distributed through the new parallel regions.

Law 2 is a block-refinement law that applies to a simple block with two ports, `p1` and `p2`, and a state machine that at the top level has two regions, `R1` and `R2`. The interfaces `I1` and `I2` represent the provided and required interfaces of the ports `p1` and `p2`. Law 2 can be used to refine such a simple block into a composite block with the same two ports and two parts (typed by `Block1` and `Block2`) each with two ports (`p1` and `ip1`, and `p2` and `ip2`), and each with its own state machine derived from one of the regions `R1` and `R2`.

In the third phase of our refinement strategy, Law 2 is applied three times to turn the block SoS into a composite block containing an instance of the block Bus and three instances of the block Device. The behaviour of the newly introduced block Device is the behaviour specified by a single region of the state machine obtained at the end of the second phase.

The provisos of Law 2 guarantee that no new operations or signals are introduced and that their treatments (in the state machine) are independent and, therefore, can be separated. In other words, Law 2 can be applied as long as a subset of the operations and signals (the external ones) of the original block are partitioned in the interfaces `I1` and `I2` (proviso 1), the transitions of the two top regions of the state machine have no triggers in common, and the two regions do not share block properties (proviso 2); the provided items (operations and signals) of `I1` are not used in the triggers of the transitions of region `R2` and the required items of `I1` are not used in the actions of the states and transitions in `R2` (proviso 3), and similarly the provided items of `I2` are not used in the triggers of the transitions of `R1` and the required items of `I2` are not used in the actions of the states and transitions in `R1` (proviso 4). Further details of refinement laws can be found in [27].

In our example, since the parallel regions of the block SoS do not share operations (they use the Bus block to communicate), the internal ports (`ip1` and `ip2`) do not require or provide any operations and can later be removed using another refinement law. Finally, it is possible to apply Law 2 multiple times obtaining a deep block decomposition, and then apply specific laws to flatten the hierarchy. The resulting model obtained after the application of the third phase is the model presented in Figures 1 and 3.

The verification of a refinement is particularly useful in the context where it is easier to verify properties of the abstract model. The verification of properties of SysML models is discussed in the next section.

6.2 Analysis of SysML models

Here we describe how we can use the refinement notions of CML to validate a SysML model. In particular, we focus on the use of animation and model checking of the corresponding CML models.

For animation or model checking, the infinite types used in our semantics or defined in the SysML model need to be eliminated. We have developed a tool to define automatically finite subsets of values of these types using existing information about the CML semantics or provided input regarding the SysML model types. For instance, for the leadership-election protocol example, the subset of identifiers actually used in the CML model is automatically defined, but the possible sets of Device, a type defined in SysML, must be provided.

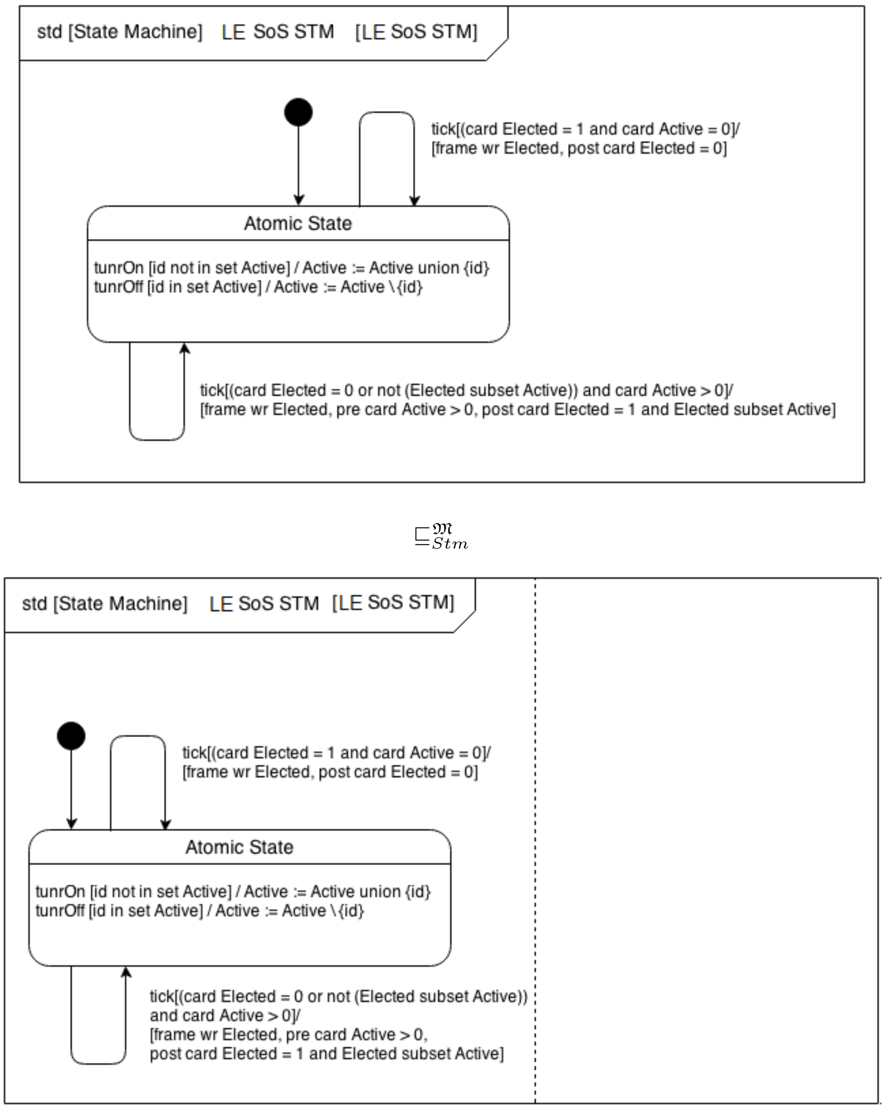

std [State Machine] LE SoS STM [LE SoS STM]
tick[(card Elected = 1 and card Active = 0]/
[frame wr Elected, post card Elected = 0]
Atomic State
tunrOn [id not in set Active] / Active := Active union {id}
tunrOff [id in set Active] / Active := Active \ {id}
tick[(card Elected = 0 or not (Elected subset Active)) and card Active > 0]/
[frame wr Elected, pre card Active > 0, post card Elected = 1 and Elected subset Active]
$\sqsubseteq^{\mathfrak{M}}_{Stm}$
std [State Machine] LE SoS STM [LE SoS STM]
tick[(card Elected = 1 and card Active = 0]/
[frame wr Elected, post card Elected = 0]
Atomic State
tunrOn [id not in set Active] / Active := Active union {id}
tunrOff [id in set Active] / Active := Active \ {id}
tick[(card Elected = 0 or not (Elected subset Active))
and card Active > 0]/
[frame wr Elected, pre card Active > 0,
post card Elected = 1 and Elected subset Active]

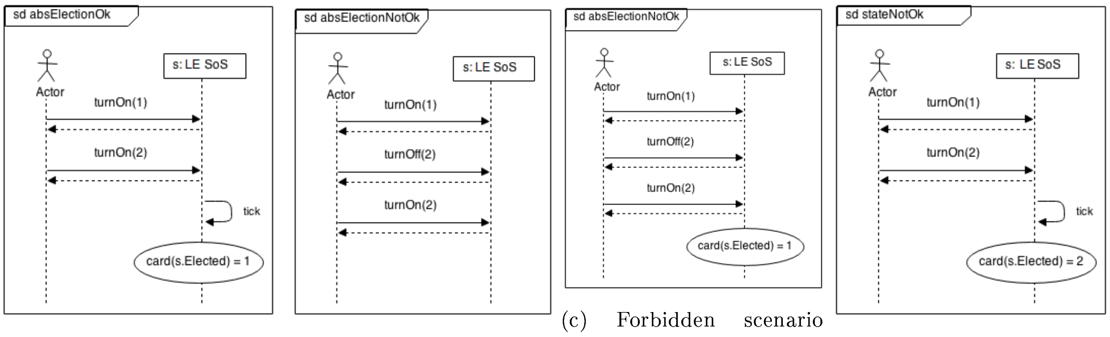
Fig. 15: Application of law `Region introduction` to abstract state machine.

Animation provides a means to exercise the SysML model by executing the communications of the CML model. This allows analysers to identify possible scenarios arising from usages of the services modelled. For instance, in the leadership-election example, the communication between the devices and the bus can be exercised according to the behaviours described by their state machines and activities.

Another approach uses refinement model checking to verify that the system can execute according to scenarios described by a sequence diagram. Using traces refinement, we can verify that the traces defined by the CML model of an interaction are in the set of traces of the CML model of a SysML model defined via a collection of diagrams. In this case, the scenarios defined by the interaction are valid for the system.

More formally, if `M` is the CML system model and `I` is the CML model of the interaction, then $\mathtt{M} \sqsubseteq_{\mathrm{T}} \mathtt{I}$ asserts that the traces of `I` are traces of `M`. The CML trace semantics is adequate for analysis based on sequence diagrams because, in this case, we are interested only on the services of the system. We recall that these correspond to operation calls, signals, and access to attributes of blocks. Availability of services (deadlock) and divergences (livelocks) are not relevant for properties defined by sequence diagrams.

As an example, we consider the first two sequence diagrams in Figure 16, describing scenarios of the abstract leadership-election model. Diagram 16a shows a



(a) Valid scenario (b) Forbidden scenario (c) Forbidden scenario with state (d) Forbidden state

Fig. 16: Sequence diagram examples.

valid scenario, where the devices 1 and 2 are turned on and the control event tick happens; we ignore the state invariant card(s.Elected = 1) for now. Diagram 16b describes a forbidden scenario: the second message is a turn off operation call for the device 2, which has not been turned on. For Diagram 16a, the model checker does not return any counterexample, so the traces of the interaction defined by the first sequence diagram are valid. In the case of Diagram 16b, a counterexample shows a trace of the interaction that is not a trace of the system model: `<[A].[B].turnOn.1,[A].[B].turnOff.2>`, where `[A]` is the identifier of the lifeline for Actor, and `[B]` of the lifeline for LE_SoS. This confirms that the scenario of Diagram 16b is not valid for the system.

We note, however, that when we consider the use of state invariants like in Diagrams 16a, 16c, and 16d, the refinement $\mathtt{M} \sqsubseteq_T \mathtt{I}$ may not hold even when the sequence diagram does specify a valid scenario of the system. Unless there are no values of the state components that violate the invariant, some of the traces of `I` include a synchronisation on the channel `inv` not used in `M`. This is perhaps surprising, but since `I` does not have a record of the state of `M`, and takes the value of the state components as input (through _get_ channels) to evaluate the invariant, its traces covers all possible values that a state component can have. On the other hand, the definition of `M` typically restricts the possible values of a state component in a given scenario via the definition of the data operations and their behaviour.

For instance, in the Diagram 16a the state invariant defines a property of the attribute Elected. Hence, the CML process `I` for the interaction that the diagram defines takes an input on the channel `LE_SOS_get_Elected` to evaluate the state invariant. Since this input is not constrained, there are traces of `I` for all values that Elected can take in this communication, including those for which the constraint does hold: the empty set and sets with more than one element. Continuations of such traces include a synchronisation on `inv`, which is never present in a trace of the CML process `M` for the leadership-election protocol model. In addition, in `M`, the value that can be output on `LE_SOS_get_Elected` after the communications corresponding to the operations `turnOn` indicated in the sequence diagram, are restricted. So, traces for other communications on `LE_SOS_get_Elected` are not included in `M`. In summary, there are several traces of `I` that are not traces of `M`, even though the sequence diagram presents a valid scenario of model

Consequently, to perform an accurate verification involving sequence diagrams with state invariants, we use another strategy performed in two steps. We note that the process $\mathtt{M} \underset{\Sigma\setminus\{\mathtt{inv}\}}{\parallel} \mathtt{I}$, which composes `M` in parallel with `I`, synchronising on all their common channels, captures the behaviour of the interaction when it uses state information provided by the system model captured by `M`. The set Σ contains all the communications used in our CML models. The traces of $\mathtt{M} \underset{\Sigma\setminus\{\mathtt{inv}\}}{\parallel} \mathtt{I}$ are the traces of `I` whose communications are also allowed by `M`. Consequently, the problematic traces mentioned above, recording spurious values for the state components not allowed by `M`, are no longer included. On the other hand, traces may be eliminated due to deadlocks that arise when a trace of `I` is not allowed by `M`, not because of the _get_ and `inv` channels, but because the communications corresponding to the sequence of messages defined in the interaction are not allowed by `M`. In this case, $\mathtt{M} \underset{\Sigma\setminus\{\mathtt{inv}\}}{\parallel} \mathtt{I}$ is a traces refinement of `M`, even though the interaction is not valid for the system.

As an example, we consider Diagram 16c, which is a variation of Diagram 16b with an added state invariant. The sequence of messages defined by 16c are not valid for the reasons already discussed earlier in relation to Diagram 16b. In spite of that, the process $\mathtt{M} \underset{\Sigma\setminus\{\mathtt{inv}\}}{\parallel} \mathtt{I}$ is a traces refinement of `M`. The parallelism $\mathtt{M} \underset{\Sigma\setminus\{\mathtt{inv}\}}{\parallel} \mathtt{I}$ deadlocks after the communication corresponding to the first message turnOn(1). At this point, the interaction process I is ready only for a communication corresponding to the second message turnOff(2), but such a communication is not available in the process M for the system model at this point. The traces of the parallelism are only the empty trace and the singleton trace with the communication for turnOn(1). Both of these are traces of M, and so $\mathtt{M} \underset{\Sigma \setminus \{\mathtt{inv}\}}{\parallel} \mathtt{I}$ is a trace refinement of M, although the interaction is not valid.

Therefore, our analysis strategy based on sequence diagrams with state invariants has two steps. First, we verify the validity of the scenario in the sequence diagram, ignoring all state invariants. Afterwards, we consider the properties defined in state invariants.

Formally, in the first step, we check the refinement $\mathtt{M} \sqsubseteq_T \mathtt{I} \setminus\setminus \{|\mathtt{inv}, *\_\mathtt{get}\_*|\}$, where we consider only traces of I without communications related to state invariants. We use *_get_* to refer to all _get_ channels, which are used in I only to access attributes for evaluation of state invariants. If the refinement holds, then we check $\mathtt{M} \sqsubseteq_T (\mathtt{M} \underset{\Sigma \setminus \{\mathtt{inv}\}}{\parallel} \mathtt{I})$ as suggested above.

When using this strategy, we find no counterexamples when considering Diagram 16a. As already explained, Diagram 16c describes a forbidden scenario with a state invariant. In this case, only the first step is necessary because, as the scenario is not valid, the refinement fails. The same counterexample presented above as part of the analysis based on Diagram 16b is relevant here. Diagram 16d, on the other hand, describes a forbidden scenario, where after two devices are turned on and the tick event takes place, the number of elected leaders is two: only one leader must exist when there are active devices after a tick event. In terms of our refinement checking strategy, the refinement checked in the first step holds, but a counterexample for the refinement checked in the second step is the trace <[A].[B].turnOn.1, [A].[B].turnOff.2, inv>, where inv indicates that $\mathsf{card}(\mathsf{s.Elected} = 2)$ is violated.

7 Implementation and validation

To validate our semantics and enable automatic generation of CML models from SysML diagrams, we have developed a model-to-text transformation plug-in for Atego's Artisan Studio. This tool can produce code written in various languages, using the ACS (Automatic Code Synchronizer) generator. ACS is a real-time synchroniser that keeps a model and its generated code synchronised, and can also generate code on demand.

We use ACS to implement the SysML to CML transformation rules described in Section 5 and produce a CML model generator. When it is added as a plug-in, the CML model-generation algorithm becomes available alongside those for other languages.

ACS generates code using a generation scheme implemented using the Transformation Development Kit (TDK). Our implementation of the CML model-generation consists of an encoding of the SysML to CML transformation rules, which the TDK then translates into a CML generator plug-in.

All the rules of the block, state machine, and activity semantics are implemented. This has made it feasible to use CML tools to validate the specified CML models. Besides syntactic errors, important modelling issues have been identified and addressed. For instance, the encoding of interfaces as subsets of operations and signals had to be modified to cope with finiteness constraints imposed on sets (as opposed to types) by CML.

Other category of problems that has been addressed is related to gaps between the semantics of the different elements, which became apparent during the implementation of the rules and the validation of the models generated by them. For example, the event management processes (controller_B processes), which were part of the models of state machines, have become part of the models of blocks. The semantics of state machines was initially developed in isolation, and, while the integration of the models for state machines and blocks did not reveal any issues with the management of events, the incorporation of activities uncovered issues in the interaction of blocks, state machines, and activities via events. It has been necessary for the controller process to become part of the parallelism that defines a block process to allow an activity process to search for particular events in a block's pool of events.

Figure 17 shows the analysis process supported by the CML tool set. In step 1, we can use Artisan Studio for building SysML models according to our guidelines, and, in step 2, the CML plug-in can be used to generate an associated CML model. The translation via compositional rules ensures full traceability from the CML model back to SysML elements. In order to perform analysis via animation and model checking, however, we have to tailor the CML model to the particular tools we have available. This is the aim of steps 3 (for animation) and 5 (for model checking).

The transformed models can be animated using the Symphony[1] [28] tool in step 4. Symphony is an integrated development environment for CML. It has several tools for editing and executing CML specifications, including an animator. Finally, the refinement analysis is performed in step 6 using the FDR3 [29] refinement checker for CSP. For that, the CML specifica-

[1] Available in http://symphonytool.org/.

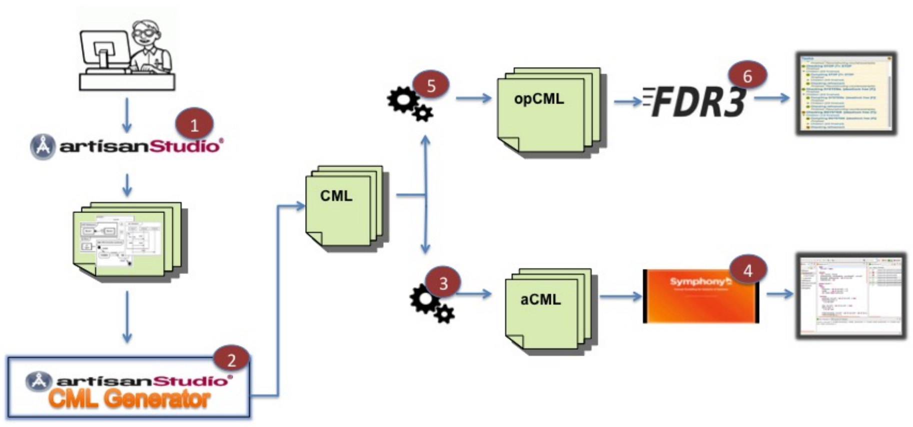


Fig. 17: Overview of the modelling approach workflow.

tion is given a CSP representation optimised for model checking as described in [30].

The modelling approach we have presented here is the result obtained after validation with the tools described in Figure 17. A number of example models generated by the tool are available in [26], including the leadership-election example we have been discussing.

8 Related work

Here we discuss works on the formalisation of SysML and UML models. Besides describing their main characteristics, we also make a comparison regarding the coverage of different model elements.

Breu *et al.* [31] have proposed a formal language called system model to specify object-oriented systems in the style of UML. A system model specification has a pre-defined mathematical structure comprising object identifiers, message passing, behaviour, communication histories, states, and so on. A UML diagram is modelled as a projection of a system model, which is regarded as a complete and unified model of the entire system. Class diagrams, state machine diagrams and sequence diagrams can be translated to a system model. On the other hand, although the semantics of these diagrams is defined in a single formalism, the verification of the consistency among the diagrams and the development of tools has not been reported.

The project Precise UML (pUML) started the development of a precise semantic model for UML diagrams. Lano and Evans [32] have proposed a systematic development process using UML and a mix of syntactic checks and formal verification of consistency, enhancements and refinements among class diagrams, statecharts, sequence, and collaboration diagrams. Modelling and verification are carried out by hand, using first-order set theory. No general translation strategy like that presented here has been developed.

Kuske *et al.* [33] have proposed an integrated semantics for UML class, object, and state machine diagrams using graph transformation. A state machine is modelled as a transformation of an object diagram. The integrated semantics allows us to visualise the evolution of a particular object diagram with respect to the state machine. This is, however, the only consistency check supported. A consistency check similar to ours, based on sequence diagrams, is proposed as future work.

Rasch and Wehrheim [34] have presented extensions to the UML class-diagram notation in which bodies of methods, their pre and postconditions, and the initialisations of attribute are specified in Object-Z [43]. An integrated semantics in CSP is proposed for these extended class diagrams and for state machine diagrams. Five notions of consistency are used: at least one trace does not deadlock (satisfiability); the model is deadlock free (basic consistency); every method is called at least once (executability); and every method becomes enabled infinitely often (availability). We do not consider all these checks, but they can be done in CML. Additionally, our approach does not extend the SysML diagrams, thus, formalisms are hidden from modellers. In particular, the behaviours of operations are specified by activities and state machines.

Davies and Crichton [12] have also proposed a formal semantics in CSP for UML class, object, statechart, sequence and collaboration diagrams. Both inter and intra-object concurrency are addressed. Inter-object concurrency allows objects to run in parallel, while intra-object concurrency allows concurrent calls of operations of a single object. The semantics is used to verify both refinements and the consistency between sequence and the remaining diagrams. The translation

	Diagram									
	bdd	obj	coll	ibd	stm	act	sd	par	Formalism	Purpose
Breu *et al.* [31]	✓				✓		✓	N/A	System Model	Spec
Lano *et al.* [32]	✓		✓		✓			N/A	Set Theory	Cons, Ref
Kuske *et al.* [33]	✓	✓			✓			N/A	Graph Theory	Cons
Rasch *et al.* [34]	✓				✓			N/A	CSP	Cons
Davies *et al.* [12]	✓	✓	✓		✓		✓	N/A	CSP	Cons, Ref
Hamilton *et al.*[35]	✓	N/A	N/A	✓		✓		✓	001AXES	Spec
Graves [36–38]	✓	N/A	N/A	✓					Description Logic	Well-Form
Café *et al.* [39]	✓	N/A	N/A	✓	✓				SystemC-AMS	Sim
Broy *et al.* [40–42]	✓					✓			Set Theory	Well-Form
fUML + PSCS [13,14]	✓			✓		✓			PSL	Well-Form
Our work	✓	N/A	N/A	✓	✓	✓	✓		CML	Cons, Ref

Table 3: Summary of related works.

from UML to CSP has been illustrated via examples. No general transformation rules are introduced.

A semantics for SysML is proposed by Hamilton *et al.* [35] in terms of axioms of the Universal Systems Language 001AXES. This language is based on a set of axioms and rules for applying function mappings and type mappings. Three primitive structures specify a mapping via sequential and parallel composition, and choice. Hamilton *et al.* have provided examples of how to model block, internal block, parametric and activity diagrams. No formal verification is proposed in this work; the focus is on the benefits of using a formal semantics to prevent the introduction of bugs.

Graves [36–38] has proposed a semantics for SysML block and internal block diagrams using a knowledge-based model for UML class diagrams. Both diagrams are formally specified using the logic ALCQI [44]; this encoding is proved to capture the part-decomposition relation correctly as a tree-like structure. No other diagrams of SysML have been formalised in this work.

Café *et al.* [39] have proposed a semantics for SysML block, internal block, and state machine diagrams as a SystemC-AMS [45] program, an extension of SystemC [46] for heterogeneous systems. Their main contribution is an interpretation of SysML diagrams for systems that mix continuous and discrete signals. Once the SysML diagrams are translated to SystemC-AMS, they can be simulated on standard tools. No formal verification is employed in that work.

Broy *et al.* [40–42] propose one of the first foundational semantics for a subset of UML2, which is called the *system model*. It is defined in terms of state machines that describe the behaviour of objects and their data structures. The system model is formalised using mathematical theories instead of existing formalisms. It is claimed that the semantics of any UML model can be represented in terms of the system model. Classes, actions and activities are mapped to the system model representation. Although the approach is significant in providing a unambiguous UML semantics, it lacks automatic support for analysing the models.

The fUML standard [13] provides a precise semantic for UML classes, activities and actions, and an extension to fUML has been developed to cover composite structures [14]. It has an executable semantics described in pseudo Java-code, formally defined using PSL (Process Specification Language) [47], an axiomatic foundational language. Despite providing a reliable semantics for a subset of UML, fUML lacks tools for formal reasoning. Some works have proposed transformations to other formal languages to enable analysis [15,16], and fUML provides a basis for validation of these transformations. Besides the constructs covered by fUML, our work considers state machines and interactions.

Table 3 summarises the works described above. The columns bdd, obj, coll, ibd, stm, act, sd and par refer to block-definition or class, object, collaboration, internal block or composite structure, state machine, activity, sequence and parametric diagrams. A tick ✓ indicates that the corresponding diagram is formalised by the work of the authors named. Parametric diagrams are not available in UML, while object and collaboration diagrams are not used in SysML, so their coverage is not applicable (N/A) in some lines of work. The purposes of the formalisations are classified as Spec (specification), Cons (consistency), Ref (refinement), Well-Form (well-formedness), and Sim (simulation).

Our work is distinctive in its definition of an integrated semantics for a substantial subset of SysML. We have a comprehensive result in terms of both the amount of diagrams covered (five diagrams—tied with Davies and Crichton [12]) and the amount of constructors covered per diagram. Our semantics cover 10 block-definition diagram constructs, 12 state machine constructs, 21 activity diagram constructs, and 12 sequence-diagram constructs. Only [48–50] cover as many constructs as we do per diagram and none, to our knowledge, covers more constructors than we do. Moreover, our semantics is mechanised, that is, it is implemented in a tool in order to generate CML specifications from the SysML model automatically, and can be used as part of a tool set for reasoning about SysML models.

9 Conclusions

We have presented a formal semantics for a comprehensive subset of SysML through a mapping into CML. Building on, and adapting, our previous work on a CML semantics for individual elements, we have defined a semantics for an integrated view of a SysML model provided by the relationships between elements in a typical SysML design. Our work is extensive with respect to the coverage of both elements and constructions of diagrams, when considered in isolation.

To allow a coherent interpretation of a SysML model, we have proposed guidelines that assign some design roles to be played by each of the considered elements in an integrated model. The structural model and the behaviour of its internal components are captured by block, internal block, state machines and activity diagrams. Desired interactions that the model must (or must not) allow are specified by sequence diagrams.

Apart from generality and integration, the following concerns have guided the design of our models.

– *Abstraction.* The semantics is given at a level of abstraction that makes them suitable for a variety of analysis techniques, including theorem proving. They are not executable, but we have shown how instantiation (to limit, for instance, the number of blocks and operations that are allowed) can produce an executable version of the model.
– *Compositionality.* Parts of the CML models can be analysed independently, and problems found can be traced back to elements of the SysML model.
– *Independence of particular tools.* The models define a theory of refinement and programming for SysML. We have explored this feature by considering refinement notions and laws for SysML models.

The main purpose of our formal semantics is to serve as a sound basis for a comprehensive reasoning strategy for SysML models based on refinement.

We take advantage of the wide range of CML constructions and operators, concerning both state and control behaviour, as well as its refinement theory. The semantics is presented as a set of compositional translation rules whose automation contributes both for validation and applicability of our work.

With the mapping of SysML into CML, it is possible to check whether traces defined by interactions are valid or not in the obtained model. This can be automatically achieved, via refinement checking, using the CML model checker being developed, or by translating the CML model into an input notation for other tools. So far, we have translated CML into CSP and used the refinement checker FDR. Another form of validation that we have discussed is animation using the CML simulator, which exercises the SysML model by executing the communications of the CML model.

In addition to scenario verification, another support for reasoning is a refinement calculus for SysML. Refinement notions for SysML have been defined by lifting CML relations for process and action refinement. This has allowed us to define a family of refinement relations for block, state machine, and state, for instance. Based on these relations, some refinement laws have been defined, as exemplified by state decomposition laws.

We do not relate our notion of refinement with the "refine" stereotype of SysML because there is no formal definition for it. On the other hand, we do formalise a notion of refinement, and it would not be difficult to use that notion to underpin the use of this stereotype.

The entire approach has been illustrated using an industrial leadership-election case study. The semantic mapping has been exemplified for some constructions of the several elements, and both model refinement and scenario validation have been carried out.

There are some interesting opportunities for further research that will contribute to the proposed approach. One of the challenges of our approach is to achieve complete traceability between CML elements, at the semantic level, and SysML constructions of the source diagrammatic model. This has been addressed by defining the translation rules in a compositional manner and documenting the generated CML to facilitate traceability. This limitation persists, however, when we optimise the generated CML in order to perform model checking because the traceability information is lost.

Another concern is the maintainability of our translation rules. Again, compositionality is helpful. Our rules are organised into groups according to the SysML elements they address, so changes are contained to the groups concerned with the elements affected. A more sophisticated mechanism of maintenance is best considered in the context of Artisan Studio.

Regarding our guidelines of usage of SysML, to facilitate their communication to practitioners and their implementation in other tools, it can be beneficial to formalise them, using OCL, for example. In addition, practitioners can also benefit from a method to construct SysML models that satisfy our guidelines; this can complement automated support to check that an existing model follows the guidelines.

A more theoretical line for future work is the use of the precise semantic defined by fUML to establish consistency between fUML and our CML semantics, for the constructs covered by fUML. Besides its denotational semantics, CML has an operational semantics, used in its animator and model checker, and an algebraic semantics, used in its theorem prover and refinement editor. Exploring the relationship between PSL and CML, in the context of the Unifying Theories of Programming [51], which are used to give the a denotational semantics for CML, is a promising way forward.

The generated CML models are not suitable for human readers. Whereas this is not relevant for our goal of reasoning purely at the SysML level, readability can be useful in other contexts. Further modularisation of the CML model to separate the semantics of the SysML protocols from the CML models of features of particular elements can improve readability. This most likely, however, requires the use of higher-order actions, a feature not currently available in CML.

One of our major objectives is to develop a comprehensive framework to allow reasoning purely at the SysML diagrammatic level, with the CML models and analyses totally hidden from the developer. We plan to develop other case studies to explore the semantic mapping and the reasoning strategy described here.

As discussed, we have implemented our translation rules and the transformations required for simulation using the CML animator. Other opportunities for automation include support for refinement in general, especially at the SysML diagrammatic level, and for optimising model checking in FDR and other tools.

Moreover, we aim to expand the coverage of our approach. We plan to provide semantics for other SysML features, including new concepts, like the parametric diagram, and exclusive characteristics of SysML.

Finally, extension of SysML to include CML concepts for refinement, and of CML to include SysML concepts, like asynchronous communication and shared variables, are interesting avenues for further work. What we have now, however, is a comprehensive and formal account for refinement of SysML, as it is currently available and supported in commercial tools.

Acknowledgements This work has been funded by the EU FP7 project COMPASS (Grant Agreement 287829), the EU Horizon 2020 INTO-CPS and the Brazilian National Council for Scientific and Technological Development - CNPq (grant agreement 483329/2012-6).

References

1. OMG, "OMG Systems Modeling Language (OMG SysML), Version 1.3," 2012.
2. J. Holt and S. Perry, *SysML for Systems Engineering.* IET, 2008.
3. S. Friedenthal, A. Moore, and R. Steiner, *A Practical Guide to SysML: The Systems Modeling Language.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. 2nd edition.
4. "Rational Rhapsody Architect for Systems Engineers." `www-142.ibm.com/software/products/us/en/ratirhaparchforsystengi`, 2013.
5. "Artisan Studio." `atego.com/products/artisan-studio/`, 2013.
6. "Sparx Systems' Enterprise Architect supports the Systems Modeling Language." `sparxsystems.com/products/mdg/tech/sysml/`, 2013.
7. J. Woodcock, A. Cavalcanti, J. Fitzgerald, P. Larsen, A. Miyazawa, and S. Perry, "Features of CML: A formal modelling language for Systems of Systems," in *7th International Conference on System of Systems Engineering*, pp. 1–6, 2012.
8. H. Graves and Y. Bijan, "Using formal methods with SysML in aerospace design and engineering," *Ann. Math. Artif. Intel.*, pp. 1–50, 2011.
9. R. Ramos, A. Sampaio, and A. Mota, "A semantics for UML-RT active classes via mapping into *circus*," in *FMOODS* (M. Steffen and G. Zavattaro, eds.), vol. 3535 of *Lecture Notes in Computer Science*, pp. 99–114, Springer, 2005.
10. H. Storrle, "Trace semantics of interactions in uml 2.0 abstract," 2004.
11. I. Abdelhalim *et al.*, "Formal verification of Tokeneer behaviours modelled in fUML using CSP|," in *Proceedings of the 12th international conference on Formal engineering methods and software engineering*, ICFEM'10, (Berlin, Heidelberg), pp. 371–387, Springer-Verlag, 2010.
12. J. Davies and C. Crichton, "Concurrency and Refinement in the Unified Modeling Language," *Formal Aspects of Computing*, vol. 15, no. 2-3, pp. 118–145, 2003.
13. Object Management Group, "Semantics of a Foundational Subset for Executable UML Models (FUML)," tech. rep., Object Management Group, 2013. OMG Document Number: formal/2013-08-06.
14. Object Management Group, "Precise Semantics Of UML Composite Structures (PSCS)," tech. rep., Object Management Group, 2014. OMG Document Number: 1.0 - Beta 1.
15. I. Abdelhalim, S. Schneider, and H. Treharne, "An optimization approach for effective formalized fuml model checking.," in *SEFM* (G. Eleftherakis, M. Hinchey, and M. Holcombe, eds.), vol. 7504 of *Lecture Notes in Computer Science*, pp. 248–262, Springer, 2012.
16. Y. Laurent, R. Bendraou, S. Baarir, and M.-P. Gervais, "Formalization of fuml: An application to process verification," in *Advanced Information Systems Engineering* (M. Jarke, J. Mylopoulos, C. Quix, C. Rolland, Y. Manolopoulos, H. Mouratidis, and J. Horkoff, eds.), vol. 8484 of *Lecture Notes in Computer Science*, pp. 347–363, Springer International Publishing, 2014.
17. A. Miyazawa, L. Lima, and A. Cavalcanti, "Formal models of sysml blocks," in *Formal Methods and Software Engineering* (L. Groves and J. Sun, eds.), vol. 8144 of *Lecture Notes in Computer Science*, pp. 249–264, Springer Berlin Heidelberg, 2013.
18. L. Lima, A. Didier, and M. CornÃľlio, "A formal semantics for sysml activity diagrams," in *Formal Methods: Foundations and Applications* (J. Iyoda and L. Moura, eds.), vol. 8195 of *Lecture Notes in Computer Science*, pp. 179–194, Springer Berlin Heidelberg, 2013.
19. L. Lima, J. Iyoda, and A. Sampaio, "A formal semantics for sequence diagrams and a strategy for system analysis,"

in *Proceedings of the International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, 2014.
20. Object Management Group, "OMG Unified Modeling Language (OMG UML), superstructure, version 2.3," tech. rep., OMG, 2010.
21. OMG, "OMG Unified Modeling Language (OMG UML), superstructure, version 2.4.1," tech. rep., 2011.
22. J. Fitzgerald and P. G. Larsen, *Modelling Systems – Practical Tools and Techniques in Software Development.* Cambridge University Press, Second ed., 2009.
23. C. A. R. Hoare, *Communicating sequential processes.* Prentice-Hall, Inc., 1985.
24. J. Woodcock, A. Cavalcanti, J. Coleman, A. Didier, P. G. Larsen, A. Miyazawa, and M. Oliveira, "CML Definition 0," Tech. Rep. D23.1, COMPASS, 2012.
25. A. Miyazawa, L. Albertins, J. Iyoda, M. Cornélio, R. Payne, and A. Cavalcanti, "Final report on combining SysML and CML," tech. rep., COMPASS, 2013.
26. L. Lima, A. Miyazawa, and A. Cavalcanti, "Case Studies of SysML to CML transformations," tech. rep., University of York, available in `http://www.compass-research.eu/whitepapers.html`, 2014.
27. A. Miyazawa and A. Cavalcanti, "Formal refinement in SysML," in *Proceedings of the 11th International Conference on Integrated Formal Methods*, 2014. Accepted for publication.
28. J. Coleman, A. Malmos, P. Larsen, J. Peleska, R. Hains, Z. Andrews, R. Payne, S. Foster, A. Miyazawa, C. Bertolini, and A. Didier, "COMPASS Tool Vision for a System of Systems Collaborative Development Environment," in *7th International Conference on System of Systems Engineering*, pp. 451–456, 2012.
29. T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. Roscoe, "Fdr3 âĂŤ a modern refinement checker for csp," in *Tools and Algorithms for the Construction and Analysis of Systems* (E. ÃĄbrahÃąm and K. Havelund, eds.), vol. 8413 of *Lecture Notes in Computer Science*, pp. 187–201, Springer Berlin Heidelberg, 2014.
30. L. Lima, "Report on Guidelines for Analysis of SysML Diagrams," tech. rep., University of York, available in `http://www.compass-research.eu/Project/Publications/SysML2CML/reportYork2014.pdf`, 2014.
31. R. Breu, R. Grosu, F. Huber, B. Rumpe, and W. Schwerin, "Systems, views and models of UML," in *UML Workshop*, pp. 93–108, 1997.
32. K. Lano and A. Evans, "Rigorous development in UML," in *Proceedings of the Fundamental Approaches to Software Engineering (FASE)*, pp. 129–144, 1999.
33. S. Kuske, M. Gogolla, R. Kollmann, and H.-J. Kreowski, "An integrated semantics for uml class, object and state diagrams based on graph transformation," in *Integrated Formal Methods* (M. Butler, L. Petre, and K. Sere, eds.), vol. 2335 of *Lecture Notes in Computer Science*, pp. 11–28, Springer Berlin Heidelberg, 2002.
34. H. Rasch and H. Wehrheim, "Checking consistency in UML diagramms: Classes and state machines," in *Formal Methods for Open Object-Based Distributed Systems (6th FMOODS'03)* (E. Najm, U. Nestmann, and P. Stevens, eds.), vol. 2884 of *Lecture Notes in Computer Science (LNCS)*, pp. 229–243, Paris, France: Springer-Verlag (Berlin/New York), Nov. 2003.
35. M. H. Hamilton, W. R. Hackler, C. Margaret, H. Hamilton, W. R. H. Published, and U. I. Permission, "A formal universal systems semantics for sysml," 2007.
36. H. Graves and Y. Bijan, "Using formal methods with SysML in aerospace design and engineering," *Annals of Mathematics and Artificial Intelligence*, vol. 63, no. 1, pp. 53–102, 2011.
37. H. Graves, "Integrating reasoning with SysML," in *INCOSE Symposium*, (Rome, Italy), 2012.
38. H. Graves, "Modeling structure in description logic," in *Proceedings of the International Workshop on Description Logics (DL2011)*, (Barcelona, Spain), 2011.
39. D. C. Café, F. Boulanger, C. Jacquet, C. Hardebolle, and F. V. D. Santos, "Multi-paradigm semantics for simulating SysML models using SystemC-AMS," *Proceedings of the Forum on Specification & Design Languages*, Sept. 2013.
40. M. Broy, M. V. Cengarle, and B. Rumpe, "Semantics of UML – Towards a System Model for UML: The Structural Data Model," Tech. Rep. TUM-I0612, Institut für Informatik, Technische Universität München, February 2006.
41. M. Broy, M. V. Cengarle, and B. Rumpe, "Semantics of UML – Towards a System Model for UML: The Control Model," Tech. Rep. TUM-I0710, Institut für Informatik, Technische Universität München, February 2007.
42. M. Broy, M. V. Cengarle, and B. Rumpe, "Semantics of UML – Towards a System Model for UML: The State Machine Model," Tech. Rep. TUM-I0711, Institut für Informatik, Technische Universität München, February 2007.
43. G. Smith, *The Object-Z Specification Language.* Kluwer Academic Publishers, 2000.
44. D. Berardi, D. Calvanese, and G. D. Giacomo, "Reasoning on UML class diagrams," *Artificial Intelligence*, vol. 168, no. 1–2, pp. 70–118, 2005.
45. A. Vachoux, C. Grimm, and K. Einwich, "Analog and mixed signal modelling with SystemC-AMS," in *ISCAS (3)*, pp. 914–917, 2003.
46. P. R. Panda, "SystemC," in *ISSS*, pp. 75–80, 2001.
47. M. Grüninger and C. Menzel, "The process specification language (psl) theory and applications," *AI Magazine*, vol. 24, no. 3, pp. 63–74, 2003.
48. J. Lilius and I. P. Paltor, "The semantics of UML State Machines," tech. rep., Turku Centre for Computer Science, 1999.
49. S. Meng, Z. Naixiao, and L. S. Barbosa, "On the semantics and refinement of uml statecharts: a coalgebraic view," in *Proceedings of the 2nd International Conference on Software Engineering and Formal Methods*, IEEE Computer Society, 2004.
50. C. Eichner, H. Fleischhack, R. Meyer, U. Schrimpf, and C. Stehno, "Compositional semantics for UML 2.0 sequence diagrams using Petri Nets.," in *SDL Forum*, vol. 3530 of *LNCS*, pp. 133–148, Springer, 2005.
51. C. A. R. Hoare and H. Jifeng, *Unifying Theories of Programming.* Prentice-Hall, 1998.