Formal refinement in SysML

Alvaro Miyazawa[1] and Ana Cavalcanti[2]

[1]alvaro.miyazawa@york.ac.uk
[2]ana.cavalcanti@york.ac.uk
Department of Computer Science, University of York, York, YO10 5GH, UK

July 27, 2015

Abstract

SysML is a UML-based graphical notation for systems engineering that is becoming a *de facto* standard. Whilst it reuses a number of UML diagrams, it introduces new diagrams, and maintains the loose UML semantics. Refinement is a formal technique that supports the validation and verification of models by capturing a notion of correctness based on observable behaviour. In this paper, we analyse the issue of formal refinement in the context of SysML. First, we identify the requirements for supporting refinement in SysML, next we propose extensions to SysML that satisfy these requirements, and finally we present a few refinement laws and discuss their validity.

1 Introduction

SysML [1] is a profile of UML 2.0 for systems engineering. SysML retains a number of UML 2.0 diagrams, modifies others (like the block definition, internal block, and state-machine diagrams) and adds a new type of diagram. It supports modelling of a variety of aspects of a system, including software and hardware components, and socio-technical aspects. SysML includes a notion of refinement, but it is informal and there is no universally accepted understanding of its meaning. Whilst it is difficult to gauge adoption of SysML in industry, its current support by tool vendors such as IBM [2], Atego [3] and Sparx Systems [4] indicates that adoption is at least perceived as wide.

In [5, 6, 7] a denotational semantics for a subset of SysML has been proposed; it is based on the state-rich refinement process algebra CML, which is a combination of VDM [8], CSP [9, 10] and the refinement calculus [11]. CML is related to the *Circus* family of refinement languages, and its semantics is specified in Hoare and He's Unifying Theories of Programming (UTP) [12], which is a relational refinement framework. *Circus* has a refinement strategy [13] with associated notions of refinement that can be directly adopted in the context of CML.

In this paper we lift the notion of refinement of CML to SysML, propose extensions to SysML that enable reasoning based on refinement at the level of the diagrammatic notation rather than CML, and present refinement laws both for diagrams written using only standard SysML and for diagrams that use our extensions. Our objective is to support stepwise refinement, and we also explain how the *Circus* refinement strategy can be lifted to SysML; the laws that we present are useful in the context of that strategy.

There have been several studies of refinement in UML. They either do not consider formal refinement [14], take refinement as a syntactic notion based directly on UML components [15, 16, 17], or do not focus on laws of refinement for model transformations as we do here [18, 19, 20, 21, 22]. Our objective is to support sound model transformation at the diagrammatic level.

The structure of this paper is as follows. Section 2 introduces SysML and its formal model [7]. Section 3 presents our notions of refinement. Section 4 discusses the limitations of SysML in supporting refinement and proposes extensions to overcome these limitations. Section 5 presents refinement laws. Finally, Section 6 summarises our results and discusses related and future work.

2 SysML and its formal model

Although our definition of refinement applies to models involving an arbitrary set of diagrams, in this paper, we focus on block definition, internal block and state-machine diagrams. Block definition diagrams allow the declaration of blocks, which are the main modelling units in SysML used to define systems and their

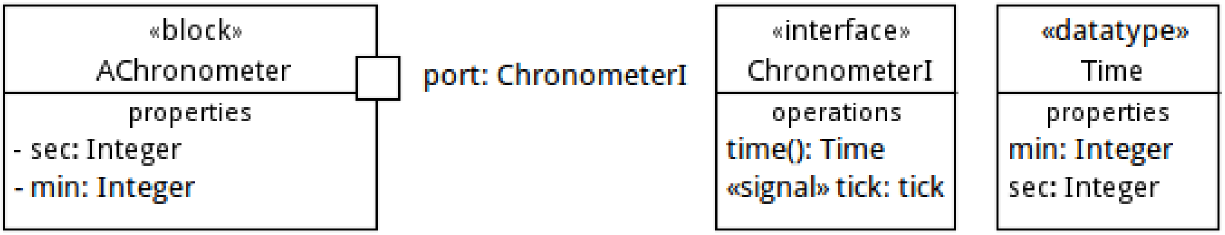


Figure 1: Block definition diagram of abstract model

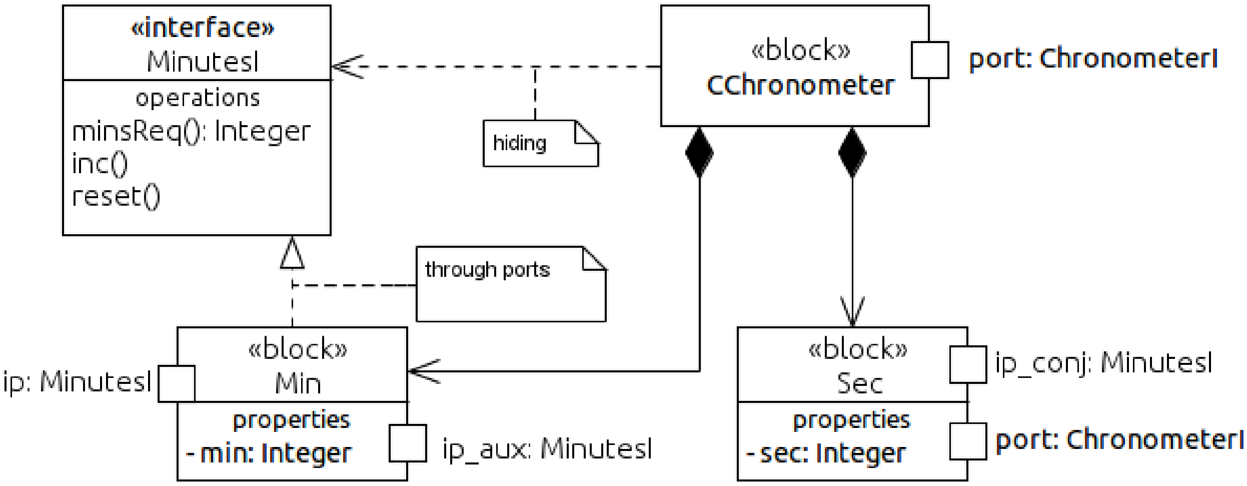


Figure 2: Block definition diagram of concrete model

components, and their relationships (composition, aggregation, generalisation and association), internal block diagrams support the specification of the internal connections of a composite block, and state machines provide the means of specifying the behaviour of a block. Activities play a similar role to state machines, and are omitted here to simplify the exposition.

To illustrate refinement in SysML, we introduce a simple example of a chronometer that records seconds and minutes, and accepts a `tick` signal that increments the chronometer and a `time` operation that queries the recorded time. The example consists of two distinct models, one abstract, depicted in Figure 1 and one concrete, shown in Figure 2, related by refinement. Whilst the abstract model is centralised, the concrete one has two components, one recording the seconds and the other recording the minutes. The components of the concrete model cooperate to realise the behaviour specified in the abstract model.

Figure 1 shows the block definition diagram of the abstract model; it declares a single block `AChronometer` with two private properties `sec` and `min`, both of type `Integer`, and a port `port` that provides the operations and signals in the interface `ChronometerI`. This interface is also defined by a block and contains an operation `time` that returns a value of type `Time`, and a signal `tick` that models the passing of time. The type `Time` is a datatype with two components, `min` and `sec`, that encode a time instant in minutes and seconds.

Since the block `AChronometer` is simple, there is no internal block diagram specifying its internal structure. The remaining diagram in the abstract model is the state-machine diagram shown in Figure 3; it contains two simple states. When the state machine is started, the properties `min` and `sec` are initialised to 0, and the state `State` is entered. When the state is active, either the internal transition triggered by `time` is executed, or the transition triggered by `tick` is executed. The first models the treatment of a call to the operation `time` and returns a value of type `Time` built from `min` and `sec`, whilst the second models the passing of time and increments the block's properties. The second transition leads to a state that is exited as soon as it is entered due to the fact that its outgoing transitions do not have triggers.

The concrete model is formed by four diagrams: one block definition diagram, one internal block diagram and two state-machine diagrams. The first is shown in Figure 2; it declares three blocks `CChronometer`, `Min` and `Sec`. The first is composed of the other two as indicated by the composition relation (arrow with a black diamond). The block `CChronometer` is similar to the block of the abstract model except that it has no properties. These are distributed in the components `Min` and `Sec`. The block `Min` has a single private property and two ports, `ip` and `ip_aux`, that both provide the operations in the interface `MinutesI`. The provided and required interfaces of a port are the sets of operation calls and signals that the block, respectively, receives and sends through the port.

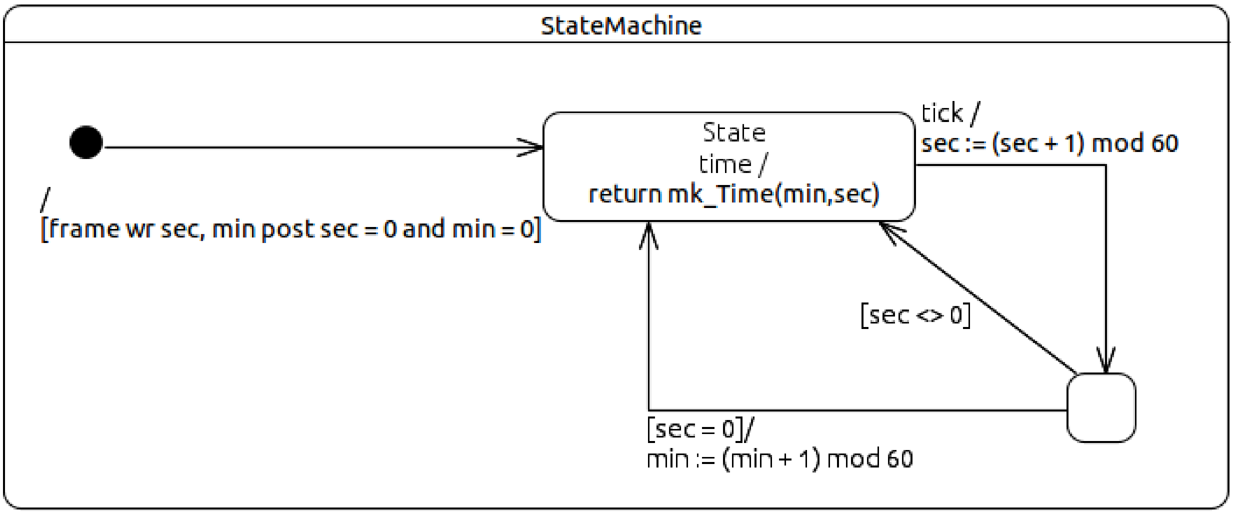


Figure 3: State machine diagram of the abstract model.

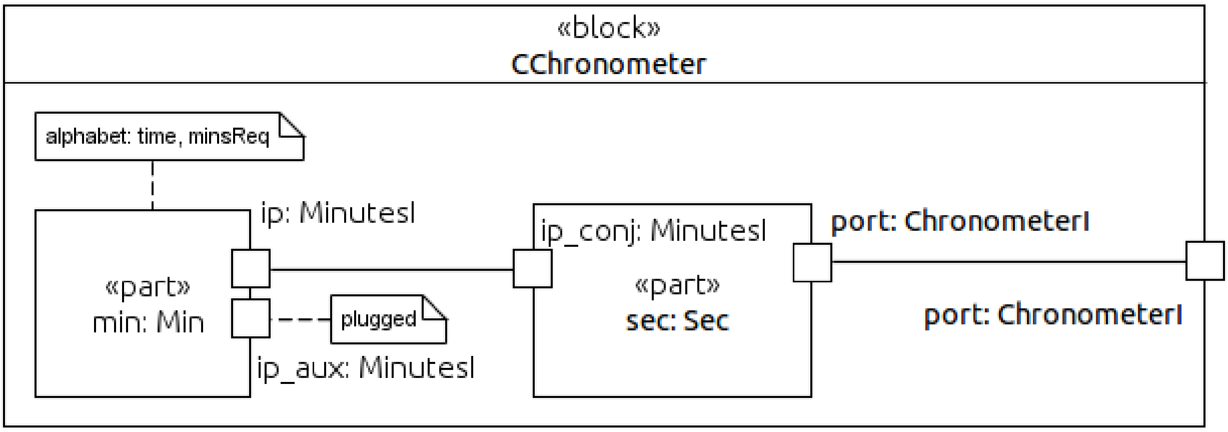


Figure 4: Internal block diagram of concrete model

The block `Sec` also has a single property, but two ports, `port` and `ip_conj`. The first is identical to the port of the block `CChronometer`, whilst the second is complementary to the port `ip` of `Min` and requires the operations in the interface `MinutesI`. The block definition diagram has some extra annotations (hiding and through port), which support refinement and are explained in Section 4.

The internal block diagram of the concrete model (Figure 4) shows the composite block `CChronometer` and its components (marked with ≪part≫); it specifies that the port `port` of `Sec` is connected to the port of `CChronometer`, and that the ports `ip` and `ip_conj` are connected to each other. Finally, the blocks `Sec` and `Min` have each one state machine (Figure 5). The state machine of `Sec` is called `SecMain` and is similar to the state machine of the abstract model, except that it delegates the operations involving minutes to the block `Min`. These operations are treated by the state machine `MinMain` that contains a single state with two internal transitions that react to a call to the operations `minsReq` and `inc`. The first returns the value stored in `min`, and the second increments it. Next, we describe the main elements of SysML that are covered in this paper.

Block. A block may declare properties, which are typed named elements (`sec` and `min` in Figure 1), receptions (`tick`), which specify the signals that can be treated by the block, and operations (`time()`). Additionally, it may generalise other blocks, use and realise interfaces, and declare ports (`ip`, `ip_conj` and `port` in Figure 4), parts (`min` and `sec` in Figure 4) and references.

State machine. State machines contain states (`State` in Figure 3), which may be simple or composite, regions, transitions (the four arrows in Figure 3), junctions (the small black circle in Figure 3), joins, forks, history junctions, initial junctions (the larger black circle in Figure 3) and final states. States may declare entry actions (executed when a state is entered), do activities (executed after the state is entered), and exit actions (executed when the state is exited). Composite states have one or more regions, which are entered, executed and exited in parallel. Regions may contain states, both simple and composite, initial junctions and final states. Initial junctions specify which state of a region is entered first, and final states indicate when the

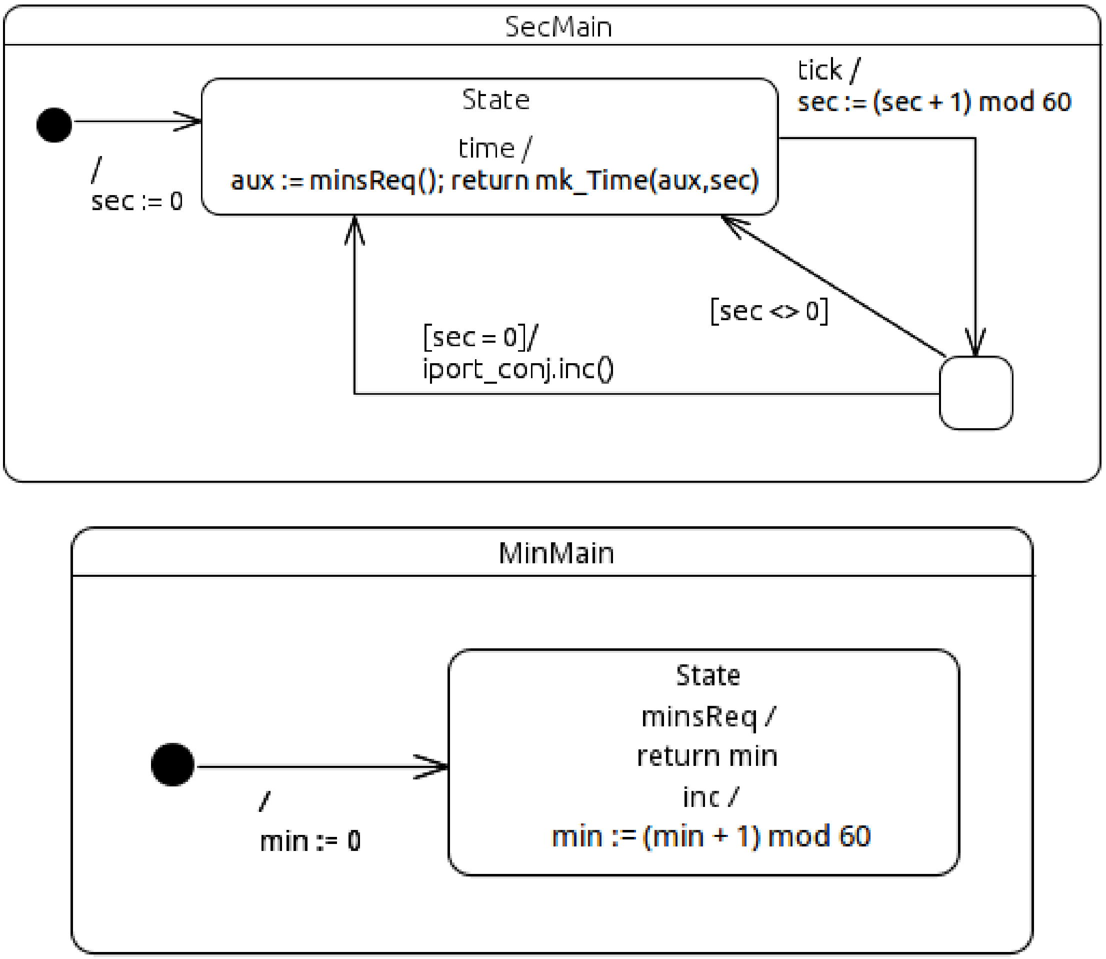


Figure 5: State machine diagrams of the concrete model.

behaviour of a region has terminated.

Transitions allow the deactivation of one or more states, and the activation of others. Whilst two states may be connected directly by a transition, the connection may be extended by the use of junctions that allow the specification of more complex flows; for instance, the three transitions connected to the junction in Figure 3 specify that after `sec` is incremented, if its value is zero, nothing is done and `State` is re-entered, otherwise, `min` is incremented and `State` is re-entered Transitions may be triggered by events (`tick` in Figure 3) and guarded by conditions (`[sec <> 0]` in Figure 3). Additionally, they may specify an action that is executed when the transition is taken (`[frame wr sec, min post sec = 0 and min = 0]` in Figure 3). Joins and forks allow transitions to link one state to multiple parallel states (contained in regions).

CML. A CML specification is a sequence of paragraphs that declare types, constants, functions, channels, channel sets and processes. Figure 6 illustrates the specification of a message buffer in CML.

First, the type of identifiers (`ID`) is defined as the type `token`, which is a universal type in CML, and the type of messages is defined as a record type with three components that record the origin of the message, its destination and the text of the message. An invariant requires that the origin and destination must be different. Next, the maximum size of the buffer is declared as a constant `MAX`, and two channels as specified: `read` and `write`. They each communicate a message, and are used to add and remove values from the buffer.

Finally, the process Buffer is declared. It encapsulates a single state component `b` that holds sequences of messages of size at most `MAX`. A data operation `Init` is declared to initialise the state component with an empty sequence, and two actions, `Read` and `Write`, specify how the channels `read` and `write` are used to interact with the buffer. In the first case, it is only possible to obtain a value through the channel `read`, if there is at least one value in the buffer. In the second case, the action specifies that it is only possible to send a value through the channel `write` if there is space on the buffer (`len b < MAX`). Finally, the data operation and actions are combined to specify the behaviour of the process (after `@`): it initialises the state and starts a recursive action (`mu X @ ...`) that at each step offer a choice between the actions `Read` and `Write`.

Central to CML is the notion of refinement that supports the comparison of processes with respect to their external communications. Calculational stepwise refinement is supported in CML by a rich catalogue of

```
types
   ID = token
   MSG :: origin: ID dest: ID msg: seq of char
      inv mk_MSG(o,d,-) == o <> d
values MAX = 5
channels read, write: MSG
process Buffer =  begin
   state b: seq of MSG inv len b <= MAX
   operations
      Init: () ==> ()
      Init() == b := []
   actions
      Read = [len b > 0] & read!(hd b) -> b := tl b
      Write = [len b < MAX] & write?x -> b := b^[x]
   @ Init(); mu X @ ((Read [] Write); X)
end
```

Figure 6: A CML specification of a message buffer

refinement laws that cover both data and process refinement.

Formalising SysML. Our formal model of SysML is a CML specification that declares a number of types, values, channels, channel sets and processes. The semantics of a block is given by a CML process; it offers interactions through a number of channels:

- `set` and `get` channels for each property of a block to allow properties to be read and written to;
- `op` and `sig` channels that allow the receipt of operations and signals; and
- `ext_op` and `ext_sig` channels for each port that also allow the receipt of operations and signals.

The model of the system defined by the SysML diagrams is captured in CML by the process that defines the block that characterises that system. The process defines the system interface in terms of the above channels, and interacts with other processes that capture other diagrams of the SysML model and restrict the interface of the system as indicated in those diagrams.

The structure of the processes that model blocks differs according to the nature of the block: simple or composite. The process that models a composite block is formed by the parallel composition of the processes that model the blocks that type its parts; the parallel composition is determined by the internal blocks diagram that describes the composite block. For instance, the block `CChronometer` in Figure 5 is modelled by a process that is defined by the parallel composition of the processes that model the blocks `Min` and `Sec` with their channels appropriately renamed to allow the communication between ports `ip` and `ip_conj`, and ports `port` of `Sec` and `port` of `CChronometer`.

Simple blocks, on the other hand, are modelled by processes that describe which operations and signals can be received by the block and may interact with a state machine process to treat them. These processes are formed by the parallel composition of processes that model the block's interface, the state machine that describes the behaviour of the block, and the block's ports. State machines are modelled by CML processes that are prepared to receive SysML events and react according to the behaviour specified in the state machines. Whilst communication in CML is synchronous, it is asynchronous in SysML, and, therefore, cannot be specified directly in CML. In our semantics, SysML communications are modelled in terms of buffers and CML communications.

The semantics of SysML is specified by inductive functions over the metamodel of SysML. The semantics of a SysML model is given by the function `t_model`, which takes a model as argument and characterises its corresponding CML specification. The formalisation of the semantics of SysML is in [7].

3 Refinement in SysML

Informally, our notion of refinement for SysML models compares the two blocks that define the systems with respect to their operations and signals. Essentially, if a block `A` is refined by a block `B`, the following properties must hold:

1. `A` and `B` must accept exactly the same public signals;
2. `A` and `B` must accept exactly the same public operations;
3. `A` and `B` must have exactly the same public properties;

4. for each public operation of A, if its return value is nondeterministically chosen from a set S, the same operation on block B must return a value that is nondeterministically chosen from a subset of S;

5. for each property of A, if its value is nondeterministically chosen from a set S, the same property on the block B must have a value nondeterministically chosen from a subset of S.

This refinement relation is induced by the CML semantics of SysML and corresponds to the refinement relation of CML process.

First of all, since process refinement is compositional in CML, and the main SysML elements (blocks, state machines and activities) map to processes used to define the CML process that defines the system model, we can refine the models of the individual diagrams to refine the SysML model as whole.

Next, we formalise the notion of refinement for blocks and state machines.

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

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

That is, block $\mathfrak{B}_1$ is refined by 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 $\mathfrak{B}_2$. With the view that a system is specified by a block in a SysML model, block refinement as formalised in Definition 1 is the main relation that must be verified to establish refinement between systems.

Data-refinement in SysML is defined similarly to behavioural refinement by lifting CML data-refinement, which is based on forward simulation.

Definition 2 (Forward Simulation) *A forward simulation ($\preccurlyeq^{\mathfrak{M}}_R$) between blocks $\mathfrak{B}_1$ and $\mathfrak{B}_2$ of a SysML model $\mathfrak{M}$ is a relation R between $\mathfrak{B}_1.PrivateProps$ and $\mathfrak{B}_2.PrivateProps$ if, and only if, R is a forward simulation between the processes $\texttt{t\_model}(\mathfrak{M}).B_1$ and $\texttt{t\_model}(\mathfrak{M}).B_2$.*

Unlike state components of CML processes, properties of blocks are not necessarily encapsulated (private). For this reason, forward simulation in SysML is defined with respect to private properties of the blocks. This is an extension of a calculational approach to data refinement presented in [11] to allow (private) variables in local blocks to have their types data refined.

Definition 3 (State machine refinement) *Let $\mathfrak{M}$ be a SysML model, and let $\mathfrak{S}_1$ and $\mathfrak{S}_2$ be state machines of $\mathfrak{M}$, then*

$$\mathfrak{S}_1 \sqsubseteq^{\mathfrak{M}}_{Stm} \mathfrak{S}_2 \Leftrightarrow \texttt{t\_model}(\mathfrak{M}).S_1 \sqsubseteq_P \texttt{t\_model}(\mathfrak{M}).S_2$$

That is, a state machine $\mathfrak{S}_1$ is refined by another state machine $\mathfrak{S}_2$ (written $\mathfrak{S}_1 \sqsubseteq^{\mathfrak{M}}_{Stm} \mathfrak{S}_2$) if, and only if, the CML process S_1 that models the state machine $\mathfrak{S}_1$ is refined by the process S_2 that models $\mathfrak{S}_2$.

Notions of refinement for states (written $\sqsubseteq^{\mathfrak{M}}_{State}$), regions, transitions and actions are similarly defined. For states, the observations that are preserved by refinement are the activation and deactivation of the top state (that is, the substates are not observable), and the signals and operation calls performed inside the state. The observations of regions are the activation and deactivation of the region and the signals and operation calls performed inside the region. Transition refinement preserves the observation of activation and deactivation of states as well as the signals and operation calls performed by the transition.

As indicated in the previous section and further explained in Section 4, a subset of CML is used as action language for SysML. This subset excluding signals, operation calls and return statements retains the original CML semantics (except that they are enclosed in a variable block that models a local copy of the shared state). For this reason, CML refinement laws for such statements can be reused in the refinement of SysML models.

4 SysML extensions

In general, we wish to prove that an abstract model, where possibly no particular design has been chosen, is refined by a more concrete model in which some design decisions have been taken. In SysML, the more concrete model often adds new operations to the abstract model in order to implement particular designs. This, however, makes the refinement invalid as new operations are now observable in the concrete model. The extra operations should in fact be internal, and used solely to implement behaviour specified in the abstract model. An alternative notion of refinement could allow addition of operations and use hiding to lift the CML notion of refinement. Here, we adopt the standard notion of refinement in process algebra directly.

In our example, we wish to show that the block AChronometer is refined by CChronometer. However, based solely on the pure SysML model, it is not possible to verify this refinement since block CChronometer clearly offers more operations than AChronometer: inc, minsReq and reset from block Min in Figure 2. These operations are used to implement the operation time, and are not meant to be visible outside the block CChronometer, that is, they are meant to be internal.

Moreover, since some of the ports of internal parts can be left unconnected, the operations and signals they offer are not called by another part, and simply making them internal, could lead them to occur spontaneously. For this reason, there needs to be a way of making them unavailable when hidden.

Finally, SysML does not provide adequate support for specifying abstract behaviours: both state machines and activities define very concrete models, and the fact that their action language is undefined is also a hindrance. The use of a programming language to define the action does not address this issue; as it does not provide support for abstract specifications.

We address the problems above through five extensions to SysML: hiding, restrictions, alphabets, plugs and the definition of an action language. The first extension supports the specification of internal signals and operations, the next three can be used to make certain signals and operations unavailable, and the fifth adds support for abstract specifications.

In order to specify that certain operations and signals are internal to a block, we propose the use of the hiding extension. A set of operations and signals represented by a SysML interface is hidden in a block by creating a dependency between the block and the interface, and adding a hiding comment to the dependency as shown in Figure 2. The semantics of this extension is given by the hiding operator of CML, which makes a set of channels internal to a process, and therefore, independent from external influences.

As already mentioned, any internal operation or signal that is offered but not used can occur spontaneously, which in turn leads to an infinite loop of internal behaviours. Therefore, only operations and signals that are used as specified by the internal block diagram can be made internal.

This restriction, however, is too strong as unused operations are often assumed to be unavailable. In fact, in our example, none of the extra operations of CChronometer can be hidden due to this restriction: some of them (reset) are not used at all, and the others are used only through a particular port of Min. The next three extensions provide mechanisms to indicate that an operation or signal is unavailable under certain situations (and can, therefore, be hidden).

Operations and signals of a block can be called by referring directly to an instance of the block, or through the ports that provide them. In order to support the specification of operations and signals that are only used through ports or (directly) through the block, we propose the use of restrictions.

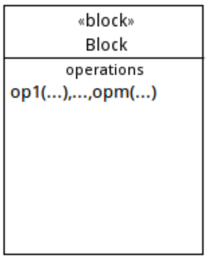
Restrictions are represented by a through ports or through block comment linked to an operation, signal or interface to indicate that it is offered only through ports or only through the block. If there is no comment, it remains available through both. Figure 2 illustrates the use of restrictions in the realisation between the block Min and the interface MinutesI. The semantics of a restriction that declares an operation O only available through ports is given by a reduction of the alphabet of the process that models the block. This reduction removes all communications that allow calls to the operation directly to the block.

Alphabets specify which operations and signals of a block are available when it is used as a part of another block. This extension is represented by a SysML comment that lists the used operations and signals and is associated with particular instances of blocks (parts). Alphabets must be connected to part (and not blocks) because they specify restrictions over the use of a block as a part. A block may be used in different contexts with different alphabets. In our example, to prevent the part min shown in Figure 4 from offering the operation reset, it is annotated with an alphabet containing minsReq and inc.

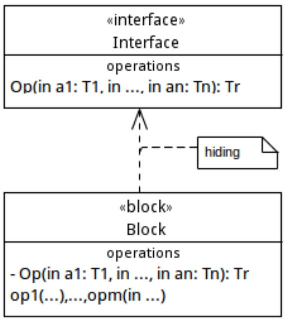
At this point, the operation reset can be hidden because it is not offered by the block CChronometer or its parts, but the remaining operations of Min cannot. They are offered on ports ip and ip_aux, but only used on ip as indicated by the connector between ip and ip_conj in Figure 4. Before these operations are made internal, the unused port ip_aux must be disabled. This is achieved by means of the plug extension, which allows us to mark a port as unused (plugged). A plug is specified by a comment linked to the port that is unavailable. Similarly to alphabets, this annotation must be placed on a port of a part since it does not affect a block in general, but only a particular use of a block. In our example, the port ip_aux is plugged as shown in Figure 4.

Finally, the problem of supporting abstract specifications is addressed by the use of a subset of CML as action language in state machines (and activities). This subset includes the CML statements (like the specification statement for instance), as well as sequential composition, external and internal choice, interleaving and guarded statements. These are the basic CML action constructors, except for those that involve communication (prefixing and parallel composition) since the communication paradigm of SysML (asynchronous) is different from that of CML (synchronous) and is already supported by signals and operation calls. In our example, the only statements that are used are assignments. specification statement and and sequential compositions as shown in Figures 3 and 5. The complete syntax of CML statements is in [23]

Law 1 Local operation introduction.



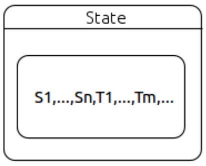
$\sqsubseteq^{\mathfrak{M}}_{Block}$



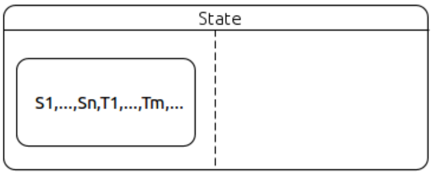
provided

1. $Op \notin used(Block.behaviour) \cup triggers(Block.behaviour)$

Law 2 Region introduction.



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



5 Refinement laws in SysML

In this section, we describe how the *Circus* refinement strategy can be applied to SysML models and present a few laws that support the strategy. These laws fall into two main groups: refinement laws that rely solely on existing SysML constructs, and laws that use alphabets, restrictions, plugs or hiding.

The *Circus* refinement strategy is an iterative process. In each iteration, initially, a centralised abstract process is data refined to introduce concrete data models, next the actions of the process are refined to introduce parallelism, and finally the process is partitioned into one or more processes that interact with each other to implement the abstract process. Each of the new processes may become the object of a subsequent iteration of the strategy.

We illustrate the use of this refinement strategy for SysML and the new laws through a simple example that verifies that the distributed concrete model shown in Figure 2 is a refinement of the centralised abstract model in Figure 1.

The first phase of the refinement strategy is supported in SysML by simulation laws that distribute a forward simulation (see Definition 2) through a SysML model. Since the data model of our concrete specification is the same as that of the abstract specification, this phase is not required in this simple example. Simulation laws can be found in [24].

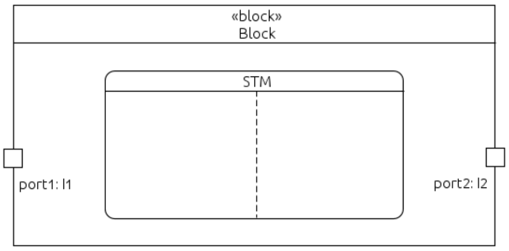
In the second phase, we start by introducing local auxiliary behaviours in the abstract model that are initially not used via state machine refinement laws. Namely, the local hidden operations `minsReq` and `inc` are introduced by the Law *Local operation introduction* presented below.

This law takes a block and an operation, introduces the operation in the block as a private operation, and hides it. The resulting block is identical to the original because the new operation is only available internally and is not used. Whilst this seems useless, further laws can take advantage of the availability of the local operation to replace behaviours by calls to it.

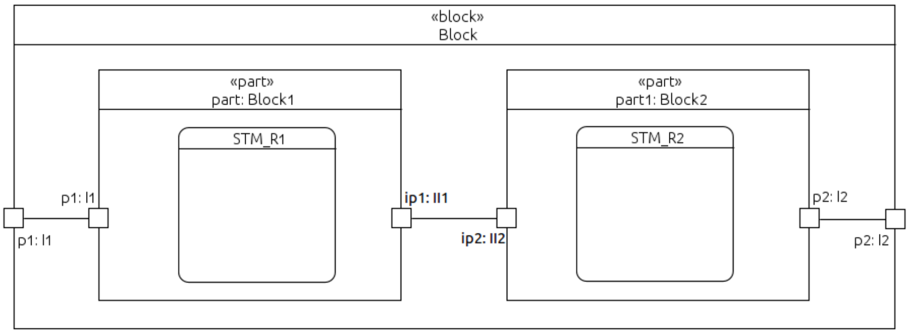
Still as part of the second phase of the strategy, we introduce some of the structure of the design. In our example, the single state in the abstract state machine is refined into a composite state with two regions using the Law *Region introduction* shown below – the first region corresponds to `SecMain` in Figure 5, and the second contains a state that offers the behaviours associated with the local operations. This does not modify the behaviour of the state machine because the newly introduced behaviours are triggered by unused local operations.

This law takes a composite state with a single region containing any number of substates and transitions, and refines it into a composite state with two regions: the first is the original region, and the second is an

Law 3 Block decomposition.



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



provided

1. $I1 \cap I2 = \varnothing \wedge I1 \cup I2 \subseteq \mathit{Block}$
2. $\mathit{triggers}(R1) \cap \mathit{triggers}(R2) = \varnothing \wedge \mathit{usedV}(R1) \cap \mathit{usedV}\,R2 = \varnothing$
3. $\mathit{provided}(I1) \cap \mathit{trigger}(R2) = \varnothing \wedge \mathit{required}(I1) \cap \mathit{used}(R2) = \varnothing$
4. $\mathit{provided}(I2) \cap \mathit{trigger}(R2) = \varnothing \wedge \mathit{required}(I2) \cap \mathit{used}(R2) = \varnothing$

where

1. $\mathit{Block}1 \cap \mathit{Block}2 = \varnothing \wedge \mathit{Block}1 \cup \mathit{Block}2 = \mathit{Block}$
2. $\mathit{provided}(II1) = \mathit{Block}1 \cap \mathit{used}(R2) \wedge \mathit{required}(II1) = \mathit{Block}2 \cap \mathit{used}(R1)$
3. $\mathit{provided}(II2) = \mathit{Block}2 \cap \mathit{used}(R1) \wedge \mathit{required}(II2) = \mathit{Block}1 \cap \mathit{used}(R2)$

empty region. This is possible because the two regions are executed in parallel, and the empty region does not introduce new behaviours observable outside the composite state.

In the third phase, the block `AChronometer` is partitioned in two using Law *Block decomposition*. This law is a block refinement law that takes a simple block with two ports, `p1` and `p2`, and a state machine that at the top level has two regions, `R1` and `R2`. It refines the simple block into a composite block with the same two ports, but whose parts are two new blocks, `Block1` and `Block2`, each with two ports (e.g., `p1` and `ip1`), and each with its own state machine derived from one of the regions `R1` and `R2`. In this law, `I1` and `I2` represent both the provided and required interfaces of the port.

The provisos of this law guarantee that no new operations or signals are introduced and that their treatments (in the state machine) are independent and, therefore, can be separated. That is, this law can be applied as long as the 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 the provided items of `I2` are not used in the triggers of the transitions of region `R1` and the required items of `I2` are not used in the actions of the states and transitions in `R1` (proviso 4).

Each block has an event pool where received events (operation calls and signals) are stored for processing. The proviso 2 of Law 3 guarantees that it is possible to partition the event pool of the block into two parts: one containing only events that may be consumed by the first region, and the other containing the events that may be consumed by the second region. Since the order in which events are sent to the state machine is non-deterministic (see [7, 25]), it is not possible to distinguish the two pairs of event pools and state machines from the original pair, thus allowing the block to be decomposed in two.

The two new blocks produced by the Law *Block decomposition* partition the operations and signals of the original block, and each has two ports; for instance, in `Block1` they are `p1` and `ip1`. The ports `p1` and `p2` are identical to those of the original block and are linked by a connector to the corresponding ports of the composite block. The connected ports `ip1` and `ip2` are introduced to allow one part to call operations of the other, which accounts for the use of block operations in the original state machine. The interfaces `II1` and `II2` of these internal ports are such that they contain as provided items those operations and signals of the associated block (`Block1` or `Block2`) that are used by the region associated with the other block, and contain as required items those that are used by its associated region. Both of these interfaces are hidden.

In a second iteration of the refinement strategy, standard CML refinement laws are used to (1) introduce a local variable `aux` initialised with `min` in the behaviour of the transition triggered by `time` (see Figure 3), and (2) replace `min` in the record constructor `mk_Time` by the local variable. Finally, Law *Operation call introduction* [24] is applied twice, once to replace `aux := min` by a call to `minsReq` via port `ip_conj`, and again to replace `min := (min + 1) mod 60` by a call to `inc` through the same port.

The soundness of these laws can be verified using the CML models induced by our semantics, and our notions of refinement. The soundness of the refinement laws presented in this paper is further discussed in [24].

6 Conclusions

In this paper we have presented our initial results regarding the use of refinement in SysML models. We have identified limitations of the diagrammatic notation that restrict, if not disallow, the use of refinement for all but the most trivial examples where concrete models add no extra operations, signals and components. To address these limitations, we have proposed extensions to SysML that address those limitations, and described a number of laws that support the development and verification of SysML models by stepwise refinement.

Current work on refinement in UML tends to follow three main directions First, [14] provides some extensions for structuring refinements, but does not present a formal notion of refinement.

Second, the notion of refinement in UML is analysed and contrasted with the notion of generalisation in [15], whilst in [16], it is related (informally) to a formal notion of refinement as inspiration for transformation patterns. Bergner et al. [17] use an extension of the informal notion of refinement available in UML to record evolution of models across different levels of abstraction. Our notion of refinement is induced by our semantics of SysML, whilst the above results use a notion of refinement based directly on components of UML models.

A third line of work is pursued [18, 19, 20, 21, 22]; our work differs from those most noticeably in our support for stepwise refinement at the level of SysML rather than of the model adopted. Hnatkowska et al. [18] formalise in a description logic refinement between models at different levels of abstraction and semantic levels, but it is not clear what properties are preserved by such notions of refinement. A similar approach is taken in [20], where the notion of refinement is based on the observation of operation calls.

Liu et al. [19] formalise a subset of UML in an object-oriented specification language that supports refinement. Similarly to our work, refinement patterns are proposed and their soundness is argued based on the formalisation. However, complicating aspects such as concurrency are not explored. Furthermore, it is not clear if compositional refinement patterns for state machines are supported as the formalisation of state machines is based on a preprocessing phase that flattens the state machine eliminating the hierarchical structure.

In [21], refinement is explored in a formal variant of UML based on Event-B [26]. This work differs from our mainly in that the Event-B approach is based on the guess-and-verify paradigm, where a new model is created and the refinement is verified rather than on refinement laws.

Finally, [22] explores refinement in UML by formalising a subset of UML in CSP. The notions of refinement are those of CSP, and the preserved properties are similar to ours, interaction between blocks via operations and signals. In that work, however, refinement supports verification via model checking.

The soundness of the refinement laws is based on the formal semantics of SysML published in [7, 5, 6] and the CML refinement calculus. As future work, we will extend the catalogue of refinement laws for SysML models and apply it to more examples. Furthermore, we plan to extend the SysML profile in [27] to include our extensions and to use the CML theorem prover [28] to formalise and mechanically verify our refinement laws, as well as to automate as much as possible the verification of the provisos generated by a refinement.

References

[1] OMG: OMG Systems Modeling Language (OMG SysML$^{\text{TM}}$). Technical report (2010) OMG Document Number: formal/2010-06-02.

[2] Rational Rhapsody Architect for Systems Engineers. `http://tinyurl.com/rrafse` Accessed: 11-04-2013.

[3] Artisan Studio. `atego.com/products/artisan-studio/` Accessed: 11-04-2013.

[4] Sparx Systems' Enterprise Architect supports the Systems Modeling Language. `www.sparxsystems.com/products/mdg/tech/sysml` Accessed: 11-04-2013.

[5] Miyazawa, A., Lima, L., Cavalcanti, A.: Formal models of sysml blocks. In Groves, L., Sun, J., eds.: Formal Methods and Software Engineering. Volume 8144 of LNCS. Springer Berlin Heidelberg (2013) 249–264

[6] Lima, L., Didier, A., Cornélio, M.: A Formal Semantics for SysML Activity Diagrams. In Iyoda, J., Moura, L., eds.: Formal Methods: Foundations and Applications. Volume 8195 of LNCS. Springer Berlin Heidelberg (2013) 179–194

[7] Albertins, L., Cavalcanti, A., Cornélio, M., Iyoda, J., Miyazawa, A., Payne, R.: Final Report on Combining SysML and CML. Technical Report D22.4, COMPASS Deliverable (March 2013)

[8] Fitzgerald, J., Larsen, P.G.: Modelling Systems – Practical Tools and Techniques in Software Development. Second edn. Cambridge University Press (2009)

[9] Hoare, C.A.R.: Communicating sequential processes. Prentice-Hall, Inc. (1985)

[10] Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall Series in Computer Science. Prentice-Hall, New York (1998) Oxford.

[11] Morgan, C.C.: Programming from Specifications. 2nd edn. Prentice Hall International Series in Computer Science (1994)

[12] Hoare, T., Jifeng, H.: Unifying Theories of Programming. Prentice Hall (1998)

[13] Cavalcanti, A.L.C., Sampaio, A.C.A., Woodcock, J.C.P.: A Refinement Strategy for *Circus*. Formal Aspects of Computing **15**(2 - 3) (2003) 146 — 181

[14] Correa, N., Giandini, R.: A UML extension to specify model refinements. In: XXXII Latin American Conference on Informatics. (2006)

[15] Pons, C., Perez, G., Giandini, R., Kutsche, R.D., TU-Berlin, F.: Understanding Refinement and Specialization in the UML. In: 2nd International Workshop on MAnaging SPEcialization/Generalization Hierarchies (MASPEGHI). (2003)

[16] Pons, C.: On the definition of UML refinement patterns. In: 2nd MoDeVa workshop, model design and validation, ACM/IEEE. (2005)

[17] Bergner, K., Rausch, A., Sihling, M., Vilbig, A.: Structuring and refinement of class diagrams. In: Proc. of the 32nd Annual Hawaii International Conference on Systems Sciences, IEEE (1999)

[18] Hnatkowska, B., Huzar, Z., Kuźniarz, L., Tuzinkiewicz, L.: On understanding of refinement relationship. Consistency Problems in UML-based Software Development: Understanding and Usage of Dependency (2004)

[19] Liu, Z., Li, X., Liu, J., Jifeng, H.: Consistency and refinement of UML models. Consistency Problems in UML-based Software Development: Understanding and Usage of Dependency (2004)

[20] Van Der Straeten, R.: Formalizing Behaviour Preserving Dependencies in UML. Consistency Problems in UML-based Software Development: Understanding and Usage of Dependency (2004)

[21] Said, M.Y., Butler, M., Snook, C.: Class and state machine refinement in UML-B. In: Proc. of Workshop on Integration of Model-based Formal Methods and Tools. (2009)

[22] Davies, J., Crichton, C.: Concurrency and refinement in the unified modeling language. Formal Aspects of Computing **15**(2-3) (2003) 118–145

[23] Woodcock, J., Cavalcanti, A., Coleman, J., Didier, A., Larsen, P.G., Miyazawa, A., Oliveira, M.: CML Definition 0. Technical Report D23.1, COMPASS Deliverable (June 2012)

[24] Miyazawa, A., Cavalcanti, A., Foster, S.: Refinement in SysML and CML. Technical report, Department of Computer Science, The University of York (2014) Available in `cs.york.ac.uk/~alvarohm/report2014a.pdf`.

[25] OMG: OMG Unified Modeling Language (OMG UML), superstructure, version 2.4.1. Technical report (2011)

[26] Abrial, J.R.: The Event-B Modelling Notation (October 2007)

[27] Bryans, J., Fitzgerald, J., Payne, R., Miyazawa, A., Kristensen, K.: SysML Contracts for Systems of Systems. In: Proc. of the 9th International Conference on Systems of Systems Enginering. (2014)

[28] Foster, S., Payne, R., Couto, L.D.: Towards Verification of Constituent Systems through Automated Proof. In: Proc. of the Workshop on Engineering Dependable Systems of Systems (EDSoS). (2014)