Grant Agreement: 287829
Comprehensive Modelling for Advanced Systems of Systems

Refinement in CML and SysML

Document Number: Document Number
Version: 0.0
Date: Submission date
Public Document

http://www.compass-research.eu
Document History

<table>
<thead>
<tr>
<th>Ver</th>
<th>Date</th>
<th>Author</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>08-01-2014</td>
<td>Alvaro Miyazawa</td>
<td>Initial document version.</td>
</tr>
</tbody>
</table>
Abstract

This report collects the results related to refinement obtained in COMPASS project. It is structured in four parts:

1. CML Refinement Calculus
2. Refinement of CML models of SysML
3. SysML Refinement Calculus
4. Refinement tool

This report presents formal refinement in the context of the COMPASS project. First, we discuss the notion of refinement in CML, and present a catalogue of refinement laws for CML. Next, we use the CML refinement calculus to refine CML specifications obtained from SysML models based on our semantics. Next, we present a refinement calculus for SysML based on the CML semantics. Finally, we discuss current tool support for the CML refinement calculus.

Whilst the second and third parts are independent of each other and support different goals, they both depend heavily on the material presented in part one, which is heavily based on the Circus refinement calculus.
# Contents

## I CML Refinement Calculus

1. Notion of Refinement 9
2. Refinement Laws: example 11
3. Catalogue of CML refinement laws 22

## II Refinement of CML models of SysML

4. Formal models of SysML 25
5. Refinement Strategy: goals 33
6. Refinement Strategy: state machines 34
7. Refinement Strategy: other SysML elements 35
8. Conclusions 36

## III SysML Refinement Calculus

9. Refinement in SysML 39
10. SysML extensions 42
11. Refinement laws in SysML 45
12. Catalogue of SysML refinement laws 50
13. Conclusions 53
IV Refinement tool
Part I

CML Refinement Calculus
In this part, we discuss the notions of refinement and refinement law available in CML and illustrate their use by means of a simple example.
Chapter 1

Notion of Refinement

As previously mentioned, the CML semantics of SysML models allows us to reason about SysML using the notion of refinement embedded in CML. This powerful notion of refinement is akin to those embedded in the notations on which CML is based: VDM and CSP. In other words, it is based on the related notion of observation and captures a notion of correctness based on reduction of observation. More precisely, a CML process $P$ is refined by a CML process $Q$ if every observation of $Q$ is a possible observation of $P$. In this respect, if $Q$ is (the model of) a proposed implementation of a given specification $P$, for example, then refinement guarantees that a user that agreed on the specification $P$ has to be satisfied by $Q$ because every observation of the behaviour of $Q$ is in accordance with the behaviours prescribed by $P$.

Embedded in this view is reduction of nondeterminism. An abstract specification $P$ typically embeds some nondeterminism to express freedom of design and implementation. Refinement reduces this nondeterminism as it moves towards more specific architectural designs and patterns of implementation.

The notion of observation is, therefore, central to our understanding of refinement. In VDM, it corresponds to the observation of the behaviour of data operations in terms of acceptable inputs and properties of the produced outputs. In CSP, it can be about (1) traces of interactions; (2) traces and refusals, also known as failures; or (3) failures and divergences. If we can only observe traces of interactions, we can only reason about safety properties. With failures, we can also reason about liveness: absence of deadlock. Finally, with a failure-divergences observation model, we can in addition reason about the possibility of an operation aborting or entering in an infinite loop of internal actions.

The refinement notion of CML is based on its semantics given using Hoare and
He’s Unifying Theories of Programming (Hoare, et al., 1998). This is a notion based on failures and divergences, but also on the behaviour of data operations (which are internal to processes). In this respect, traditional refinement techniques based on data refinement, for instance, are valid, as are techniques for model checking traditionally adopted in CSP, after extension to deal with the rich data types of VDM (also adopted in CML). In the context of our SysML models, we note that observation comes from communications that model operation calls and signals in the diagrams. So, refinement ensures that possibility and availability of interaction between blocks is maintained at all levels of development. This is the basis for the refinement assertion in Figure 28; the sequence diagram describes possible interactions between blocks, and the assertion verifies that those interactions are allowed by the model as a whole.

In addition, nondeterminism can be embedded in the CML account of SysML models in the definition of the actions. The action language of SysML is not defined, and as already mentioned in our work we adopt an extended subset of the CML action language. It includes specification constructs that can be nondeterministic for abstraction. Refinement of the CML models can eliminate this nondeterminism.

Furthermore, it is often necessary to change the data representation and data operations used in an abstract SysML model (and captured in the corresponding CML model). This requirement may stem from the attempt to verify a program using refinement, in which case the specification is refined into the program, or from the need to optimise the time efficiency by exploring different data structures and algorithms. This can be achieved by data refinement, in which two processes with different states are compared.

Central to the practical application of refinement based notations is the refinement calculus, which consists of refinement laws that state that specific refinements hold. For example, one such law is the law int-choice-elim shown below; it specifies that the internal choice of two actions can be refined by one of them.

In most cases, the refinement laws contain provisos that are easier to verify than the refinement itself, thus providing a practical means for using the notion of refinement without having to prove complex statements.
Chapter 2

Refinement Laws: example

What we present in this chapter is a refinement strategy for our CML models of SysML diagrams. Such a strategy is a procedure for the application of a well-defined set of laws in a particular order. With the application of such a strategy, we can transform a CML model in a stepwise manner. In our case, we can transform the CML models of SysML diagrams that can be automatically generated.

Soundness of the transformations arises from the soundness of the laws of CML and the facts that refinement is transitive and the constructs of CML are compositional with respect to refinement. In this way, after a sequence of applications of refinement laws to the whole CML model or to any of its components, we are guaranteed that the new model that we obtain is a refinement to the original CML model.

As already mentioned, the objective of the strategy that we present here is the simplification of the model generated automatically to facilitate later reasoning. In this respect, in general, what we obtain is a model whose structure is much simpler. We consider the strategy in more detail in Chapter 6.

In this chapter, we illustrate one of the main refinement steps in our strategy: parallelism elimination by means of a small example. This is particularly relevant for two phases of our refinement strategy discussed in Chapter 6: “internal process refinement” and “collapsing of controller and internal processes”.

The objective of the refinement is to eliminate as much parallelism as possible from the CML action shown below.

```
Init(); mu X @ mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip); X [||{|a,b,e|}||]
mu X @ (}
```
This is a parallel action (\([\text{|cs|}]\), where cs is the synchronisation set of the parallelism) where the left hand side executes a CML operation Init, and starts a recursion (\(\mu X \ominus \ldots \)) whose body is itself a recursion (\(\mu Y \ominus \ldots \)). This inner recursion offers a choice (\([\ldots]\)) of synchronising on the channel a and executing the operation opA (a \(\rightarrow\) opA()), or synchronising on b and executing the operation opB, or synchronising on e and terminating the inner recursion (e \(\rightarrow\) Skip). Once the inner recursion terminates, a new step of the outer recursion starts (X).

The right hand side of the parallel action is also a recursive action (\(\mu X \ominus \ldots \)) whose step is an interleaving (\(\|\|\)) of two actions: the first receives a value x on the channel in1 (in1?x \(\rightarrow\) ...) and, if it is greater than zero (x > 0), synchronises on a and terminates, otherwise it synchronises on a twice and terminates; the second action similarly receives a value x on the channel in2 (in2?x \(\rightarrow\) ...), and if x is less than zero (x < 0), it synchronises on b once, otherwise it synchronises on b twice. Once the interleaving terminates, the action synchronises on the channel e and recurses (e \(\rightarrow\) X).

This pattern of two or more recursive actions in parallel appears throughout our models. For instance, in the model of a state the status action is a simpler version of the left hand side of the pattern, that is, it behaves as a server that executes a particular action in response to a specific communication. The off action fits into the right hand side of the pattern; it executes a recursive complex behaviour and delegates simple behaviours to the status action. Most of the steps required to eliminate the parallelism in this example are required for eliminating the parallelism of similar patterns of parallelism of recursions.

This form of parallelism is eliminated by first merging the parallel recursions, and then applying a series of step and distribution laws to obtain a sequential version of the parallel body of the recursion. In cases where it is not possible to merge the parallel recursions, we must calculate a fixed point of the target recursion by unfolding the parallel recursions and eliminating the parallelism, and then using fixed point laws to introduce the desired recursion. The internal process refinement phase discussed in Chapter 6 considers this case.

To refine our example action, we first apply the rule op-par-dist to extract Init() from the parallelism.

This law relies on the fact that a CML operation does not communicate on chan-
**Law 2 Law op-par-dist**

\[ \text{Op; (A1} \ [\text{ns1} \ | \ cs \ | \ ns2 \ ] \ A2) = \ (\text{Op; A1} [\text{ns1} \ | \ cs \ | \ ns2 \ ] \ A2) \]

provided

1. \(\text{wrtV(Op) subset ns1}\)
2. \(\text{wrtV(Op) inter usedV(A2) =} )\)

nels, that the particular operation only alters variables in \text{ns1} (wrtV(Op) subset \text{ns1}), and that the right hand side does not use any of the variables altered by the operation (wrtV(Op) inter usedV(A2) = ). The remaining laws used in this chapter can be found in Appendix 2. The result of applying law \text{op-par-dist} to our example is the following action.

\[
\text{Init(); } (\mu X @ (\mu Y @ (a -> \text{opA(); Y [ ] b -> \text{opB(); Y [ ] e -> Skip}); X) [\{|a,b,e|\}]) \\
\mu X @ (\text{in1?x -> (if x > 0 then a -> Skip else a -> a -> Skip) [ ]} \\
\text{in2?x -> (if x < 0 then b -> Skip else b -> b -> Skip) } \\
\text{); e -> X})
\]

In general, the partial elimination of parallelism relies on parallel actions where at least one side is a prefixing (comm -> A). The action above does not have prefixed actions in either side, so we must first put it in the right form. For that, first, we transform the parallelism of recursions into a recursion of parallelisms to be able to tackle the parallel action inside the recursion. This is achieved by the law sync-rec, the result of which is shown below.

\[
\text{Init(); } \mu Y @ (\mu Y @ (a -> \text{opA(); Y [ ] b -> \text{opB(); Y [ ] e -> Skip}); [\{|a,b,e|\}]) \\
( \text{in1?x -> (if x > 0 then a -> Skip else a -> a -> Skip) [ ]} \\
\text{in2?x -> (if x < 0 then b -> Skip else b -> b -> Skip) } \\
\text{); e -> Skip} \\
\text{); X})
\]

Law sync-rec relies on the fact that each step of the recursions is terminated by
a synchronisation on the channel e. When the recursion step on the right hand side reaches the synchronisation on e, it terminates (restarting the recursion). The same synchronisation terminates the inner recursion (mu Y @ ...) on the left hand side, thus restarting the outer recursion (mu X @ ...). This means that the outer recursion on the left hand side and the recursion on the right hand side occur in a lockstep fashion, that is, both steps of the recursions start and end at the same time.

The action resulting from merging the recursion still does not fit the pattern required to eliminate part of the parallelism, and we apply law rec-unfold to unfold the innermost recursion (mu Y @ ...) yielding the following action.

```plaintext
Init(); mu X @ ((
  (a -> opA()); mu Y @ (a -> opA()); Y [] b -> opB(); Y [] e -> Skip) []
  b -> opB(); mu Y @ (a -> opA()); Y [] b -> opB(); Y [] e -> Skip) []
  e -> Skip) []
  [] in1?x -> (if x > 0 then a -> Skip else a -> a -> Skip)
  []
  in2?x -> (if x < 0 then b -> Skip else b -> b -> Skip)
 ); e -> Skip)
); X)
```

Whilst the parallel action (within the recursion) is still not in the pattern we require, we can observe prefixed actions in the beginning of both actions. Now, we must apply distribution laws to obtain parallel actions with single prefixed actions on at least one side. First, we apply the law seq-ext-dist to distribute the action e -> Skip over the external choice, resulting in the following action.

```plaintext
Init(); mu X @ ((
  (a -> opA()); mu Y @ (a -> opA()); Y [] b -> opB(); Y [] e -> Skip) []
  b -> opB(); mu Y @ (a -> opA()); Y [] b -> opB(); Y [] e -> Skip) []
  e -> Skip) []
  [] in1?x -> (if x > 0 then a -> Skip else a -> a -> Skip)
  []
  in2?x -> (if x < 0 then b -> Skip else b -> b -> Skip)
 ); e -> Skip)
); X)
```
This is possible because independently of the choice (in1?x -> ... or in2?x -> ...), the action e -> Skip is still available upon termination of the choice.

Next, we apply the law par-ext-dist, which distributes the parallel composition over the external choice on the right hand side. The resulting action is shown below.

\[
\text{Init()}; \mu X @ (in1?x -> (if x > 0 then a -> Skip else a -> a -> Skip); e -> Skip) \]

\[
\text{in2?x -> (if x < 0 then b -> Skip else b -> b -> Skip); e -> Skip }
\]

This law relies on the fact that the left hand side can only progress if the right hand side agrees to synchronise (initials(LHS) in set cs), and on the fact that the left hand side is deterministic.

Now, we are ready to eliminate some of the parallelism. This task relies heavily on laws such as prefix-par-dist1, which extracts a communication on which both sides of the parallel action synchronise, and prefix-par-dist2, which extracts a communication from a single side of the parallel action. For our current action, the only applicable law is prefix-par-dist2, but a single application of this law is not enough, so we apply it exhaustively. The resulting action is as follows.

\[
\text{Init()}; \mu X @ (in1?x -> (a -> opA(); \mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip) 
\]

\[
[] b -> opB(); \mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip) 
\]

\[
[] e -> Skip 
\]

\[
[[a,b,e]] 
\]

\[
in1?x -> (if x > 0 then a -> Skip else a -> a -> Skip); e -> Skip 
\]

\[
[] b -> opB(); \mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip) 
\]

\[
[] e -> Skip 
\]

\[
[][] 
\]

\[
in2?x -> (if x < 0 then b -> Skip else b -> b -> Skip); e -> Skip 
\]

\[
); X 
\]
This law relies on the fact that one side of the parallelism (left hand side) cannot progress until the other side agrees to synchronise, while the other side may progress (in1 and in2 are not in the synchronisation set) independently.

Again, we have reached a point where neither law prefix-par-dist1 nor law prefix-par-dist2 can be applied. The strategy is as follows: distribution laws involving parallelism, sequence, external choice are first applied multiple times, and then prefix distribution laws are applied.

We follow this pattern and apply the law seq-cond-dist to move e -> Skip into the conditionals, law par-ext-dist to distribute the parallelisms over the external choices, and law par-cond-dist to distribute the parallelism over the conditionals.

The resulting action is as follows.

```
Init(); mu X @ (in1?x -> {
    (if x > 0
        then a -> opA(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
        [||{a,b,e}||]
        a -> Skip; e -> Skip
    else a -> opA(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
        [||{a,b,e}||]
        a -> a -> Skip; e -> Skip

    )

    (if x > 0
        then b -> opB(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
        [||{a,b,e}||]
        a -> Skip; e -> Skip
    else b -> opB(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
        [||{a,b,e}||]
        a -> a -> Skip; e -> Skip

    )

    (if x > 0
        then e -> Skip [||{a,b,e}||] a -> Skip; e -> Skip
    else e -> Skip [||{a,b,e}||] a -> a -> Skip; e -> Skip

    )

    )

} in2?x -> {
    (if x < 0
        then a -> opA(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
        [||{a,b,e}||]
        b -> Skip; e -> Skip
    else a -> opA(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
        [||{a,b,e}||]
        b -> Skip; e -> Skip

    )
```
The law seq-cond-dist relies on the fact that the action if \( b \) then \( A \) else \( B \) is equivalent to the action \( [b] \land A \land \lnot b \land B \) and on the law seq-ext-dist previously discussed. Similarly, law par-cond-dist relies on rewriting the conditional as an external choice and distributing the parallelism using law par-ext-dist.

Once again, we are in a position to apply prefix distribution laws. In this case, the only applicable law is prefix-par-dist1. The result of applying this law exhaustively is shown below.

```plaintext
Init(); mu X @ (\lnot x < 0 ?
    (\lnot x > 0 ?
        a -> (opA();
            mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
            [||{a,b,e}||]
            e -> Skip)
        else a -> (opA();
            mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
            [||{a,b,e}||]
            e -> Skip)

    else b -> opB();
        mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
        [||{a,b,e}||]
        a -> Skip; e -> Skip)

) (if x > 0
    then b -> opB();
        mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
        [||{a,b,e}||]
        a -> Skip; e -> Skip
    else b -> opB();
        mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
        [||{a,b,e}||]
        a -> a -> Skip; e -> Skip

) (if x > 0
    then e -> Skip
    else e -> Skip

17
This law relies on the fact that both sides of the parallelisms (to which it can be applied) can only progress through a single synchronisation that depends on the other side.

Only four parallel actions have been affected by the application of the prefix distribution law. This is because the remaining parallel actions offer synchronisations on different channels on which they need to agree. Their parallel composition therefore leads to a deadlock and this can be made explicit by applying law para-deadlock. The result is as follows.

\begin{verbatim}
Init(); mu X @ (in1?x -> {
  if x > 0 then a -> (opA(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
  [] in2?x -> {
    if x < 0 then a -> opA(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
    [] (if x < 0 then Stop else Stop)
  } (if x > 0 then Stop else Stop)
})
\end{verbatim}

This law is a simple example of how the prefix distribution law can be applied to parallel processes to ensure that they synchronise in a specific way.

The result is as follows.

\begin{verbatim}
Init(); mu X @ (in1?x -> {
  if x > 0 then a -> (opA(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
  [] in2?x -> {
    if x < 0 then a -> opA(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
    [] (if x < 0 then Stop else Stop)
  } (if x > 0 then Stop else Stop)
})
\end{verbatim}

The result is as follows.

\begin{verbatim}
Init(); mu X @ (in1?x -> {
  if x > 0 then a -> (opA(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
  [] in2?x -> {
    if x < 0 then a -> opA(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
    [] (if x < 0 then Stop else Stop)
  } (if x > 0 then Stop else Stop)
})
\end{verbatim}

This law relies on the fact that both sides of the parallelisms (to which it can be applied) can only progress through a single synchronisation that depends on the other side.

Only four parallel actions have been affected by the application of the prefix distribution law. This is because the remaining parallel actions offer synchronisations on different channels on which they need to agree. Their parallel composition therefore leads to a deadlock and this can be made explicit by applying law para-deadlock. The result is as follows.

\begin{verbatim}
Init(); mu X @ (in1?x -> {
  if x > 0 then a -> (opA(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
  [] in2?x -> {
    if x < 0 then a -> opA(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
    [] (if x < 0 then Stop else Stop)
  } (if x > 0 then Stop else Stop)
})
\end{verbatim}

This law is a simple example of how the prefix distribution law can be applied to parallel processes to ensure that they synchronise in a specific way.

The result is as follows.

\begin{verbatim}
Init(); mu X @ (in1?x -> {
  if x > 0 then a -> (opA(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
  [] in2?x -> {
    if x < 0 then a -> opA(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
    [] (if x < 0 then Stop else Stop)
  } (if x > 0 then Stop else Stop)
})
\end{verbatim}

This law is a simple example of how the prefix distribution law can be applied to parallel processes to ensure that they synchronise in a specific way.

The result is as follows.

\begin{verbatim}
Init(); mu X @ (in1?x -> {
  if x > 0 then a -> (opA(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
  [] in2?x -> {
    if x < 0 then a -> opA(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
    [] (if x < 0 then Stop else Stop)
  } (if x > 0 then Stop else Stop)
})
\end{verbatim}

This law is a simple example of how the prefix distribution law can be applied to parallel processes to ensure that they synchronise in a specific way.

The result is as follows.

\begin{verbatim}
Init(); mu X @ (in1?x -> {
  if x > 0 then a -> (opA(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
  [] in2?x -> {
    if x < 0 then a -> opA(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
    [] (if x < 0 then Stop else Stop)
  } (if x > 0 then Stop else Stop)
})
\end{verbatim}

This law is a simple example of how the prefix distribution law can be applied to parallel processes to ensure that they synchronise in a specific way.

The result is as follows.

\begin{verbatim}
Init(); mu X @ (in1?x -> {
  if x > 0 then a -> (opA(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
  [] in2?x -> {
    if x < 0 then a -> opA(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
    [] (if x < 0 then Stop else Stop)
  } (if x > 0 then Stop else Stop)
})
\end{verbatim}
[] (if \(x > 0\) then Stop else Stop)
)
[] \(\text{in2?x} \rightarrow \{
(\text{if } x < 0 \text{ then Stop else Stop})
\()
[] (\text{if } x < 0 \text{ then Stop else Stop})
\)
[] (if \(x < 0\) then Stop else Stop)
\)

To simplify this action further, we can apply the law cond-elim to eliminate a conditional where both branches are the same action. The result is shown below.

\[
\text{Init(); } \mu X \odot (\text{in1?x} \rightarrow \{
(\text{if } x > 0 \text{ then } a \rightarrow (\text{opA()} \mu Y @ (a \rightarrow \text{opA(); Y [] } b \rightarrow \text{opB(); Y [] e} \rightarrow \text{Skip})
[[\{a,b,e\}]]
\)
\text{e} \rightarrow \text{Skip})
\text{else } b \rightarrow (\text{opB(); } \mu Y @ (a \rightarrow \text{opA(); Y [] } b \rightarrow \text{opB(); Y [] e} \rightarrow \text{Skip})
[[\{a,b,e\}]]
\)
\text{b} \rightarrow \text{Skip; e} \rightarrow \text{Skip})
\)
[] (\text{if } x < 0 \text{ then Stop else Stop})
\)
\)

Next, since the deadlocked action Stop is the unit of the external choice, it can be eliminated by the application of the law ext-unit. The result is as follows.
Init(); mu X @ (in1?x -> {
    if x > 0 then a -> (opA(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
        [|||a,b,e|||])
        e -> Skip
    else a -> (opA(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
        [|||a,b,e|||])
        a -> Skip; e -> Skip
    ) []
    in2?x -> {
    if x < 0 then b -> (opB(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
        [|||a,b,e|||])
        e -> Skip
    else b -> (opB(); mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
        [|||a,b,e|||])
        b -> Skip; e -> Skip
    )
    ); X

\[\text{Next, the operation calls can be pulled out of the parallelism by applications of the law op-par-dist as in the first step of our refinement. The resulting action is shown below.}\]

Init(); mu X @ (in1?x -> {
    if x > 0 then a -> opA();
    mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
        [|||a,b,e|||])
        e -> Skip
    else a -> opA();
    mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
        [|||a,b,e|||])
        a -> Skip; e -> Skip
    ) []
    in2?x -> {
    if x < 0 then b -> opB();
    mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
        [|||a,b,e|||])
        e -> Skip
    else b -> opB();
    mu Y @ (a -> opA(); Y [] b -> opB(); Y [] e -> Skip)
        [|||a,b,e|||])
Once again, we have reached an action that is not suitable for parallelism elimination. In this case, the same pattern of unfolding a recursion, distributing the parallelism, and extracting the prefix can be applied to produce the final action shown below.

```
Init(); mu X @ {
  in1?x -> {
    if x > 0
      then a -> opA(); e -> Skip
      else a -> opA(); a -> opA(); e -> Skip
    }
  in2?x -> {
    if x < 0
      then b -> opB(); e -> Skip
      else b -> opB(); b -> opB(); e -> Skip
    }
} X
```

This action is much simpler than our original action and is fully sequential. Nevertheless, it is a correct refinement of the original action, that is, every behaviour of this action is allowed by the original one. The correctness of the refinement carried out is based on the fact that the refinement relation is monotonic with respect to CML action constructors, that is $A \models B \Rightarrow F(A) \models F(B)$, and that the refinement relation is transitive, that is $A \models B \land B \models C \Rightarrow A \models C$.

This example illustrates the use of refinement laws in an ad-hoc fashion; however, even at this level, we notice strategies that are used over and over during the refinement of a model. Such refinement strategies are related to particular patterns in our models.

Frequently, these patterns apply to whole families of models. In this case, we can formalise the patterns in the form of a refinement strategy and apply the strategy to any model that conforms to a particular structure. This is the subject of our next chapter.
Chapter 3

Catalogue of CML refinement laws
Part II

Refinement of CML models of SysML
SysML [OMG10] 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 [Rat], Atego [Art] and Sparx Systems [Spa] indicates that adoption is at least perceived as wide.

In COMPASS, the semantics of SysML is given in terms of the COMPASS modelling language (CML) (Woodcock, et al., 2012), which is a formal specification language that integrates VDM (Fitzgerald, et al., 2009), CSP (Hoare, 1985), Dijkstra’s language of guarded commands (Dijkstra, 1975) and the refinement calculus (Back, et al., 1998) to support the specification of state-rich distributed models.

A CML specification consists of a number of paragraphs declaring types, classes, functions, values, channels, channel sets and processes. Processes are the main elements of a specification and process definitions use all other elements to specify the behaviour of a process.

CML also supports a variety of analyses such as deadlock freedom, divergence freedom, and refinement. These analyses are intended to be supported by a model checker, a refinement tool and a theorem prover.

The formal basis given to SysML and the availability of a refinement notion in CML allows us to reason about SysML models by means of refinement. In this chapter, we introduce the formal notion of refinement available in CML and discuss how it can be applied to SysML models through their CML semantics.

Chapter ?? recapitulates our formal models of SysML as well as our guidelines of usage that characterise the subset of SysML models that are treated. Chapter 5 discusses the objectives of a refinement strategy when applied to SysML models, and Chapter 6 describes a refinement strategy that transforms our models of SysML state machines, and Chapter 7 discusses the application of similar strategies to models of blocks, activities and sequence diagrams. Chapter 8 summarises the main results presented in this chapter and discusses the necessity of tool support as well as related works.
Chapter 4

Formal models of SysML

In D22.4, we identify a subset of the SysML notation that is covered by the COMPASS formalisation technique. In doing so, we provide a number of guidelines of usage that aim at supporting the definition of interesting cohesive formal models of SysML models written in CML. The guidelines maximise the definedness of a SysML model at both the entity definition level and at the instance definition level.

The entity definition guidelines are as follows:

1. The body of each operation must be defined by a state machine diagram, by a CML statement, or by an activity diagram;
2. In the graph of the composition relation, there must be exactly one connected component with more than one node, and this component must be a tree;
3. The blocks in the model must form a connected graph where the edges are either generalisation or composition relations;
4. A composite block must not have attributes, operations, signals, activities or state machines;
5. Associations in the block definition diagram must be matched in the internal block diagram typed connectors;
6. Associations must be used in place of aggregation;
7. The connectors between ports are not typed.

The instance-level guidelines are as follows:
1. Each composition head must specify the connections between its parts through an internal block diagram;

2. All blocks in a composition whose minimum cardinality is larger than 0 must appear in the internal block diagram of the containing block in numbers compatible with their multiplicities;

3. The cardinalities that appear in an internal block diagram must be constants;

4. Ports may only be connected to other ports.

Since the action language of SysML is not defined, we adopt an extended subset of CML.

Finally, the simplification assumptions are as follows:

1. Each block must either have an associated state machine diagram or no other associated diagram specifying its behaviour;

2. Sequence diagrams specify scenarios of interactions between blocks that the model must or must not allow;

3. Operations are always synchronous. Asynchronous operations should be modelled by signals.

The rationale for these guidelines is presented in Deliverable D22.4, along with an example model. In a few words, they enable the construction of meaningful CML models as described in the sequel.

Figure 28 gives an overview of the structure of CML models as derived from SysML models. Each block, state machine, activity, port and sequence diagram is modelled as a CML process. These processes are combined in parallel to describe the overall model. The communication among the parallel processes is specified based on the connections of the elements in the SysML model.

As indicated in our guidelines, the top level elements of the SysML models we consider are blocks and sequence diagrams. Furthermore, blocks can contain sub-blocks, ports, activities and a state machine. As depicted in Figure 28, the semantics of each of these elements is given by a CML process. In the case of blocks that contain other elements, the process that models it is defined by the parallel composition of the processes that define each of its elements. Throughout the model, the composition of parts (e.g., block, state machine) is given by the parallel composition of the associated processes.

Unlike blocks, state machines and activities, the processes that model sequence diagrams are not used by other elements to define the semantics of the model. Sequence diagrams are interpreted as scenarios to which the model should (or
should not) conform, and as such, give rise to refinement assertions as shown in Figure 28. For a thorough description of the CML models, refer to (COMPASS D22.4, 2013).

Figure 28 Structure of CML models

It is worth mentioning that parallelism plays an important role in our models. It is used for modelling concurrent behaviour as well as, in the interest of compositionality, requirement conjunction as already explained. This is true at all levels of the model. For instance, the different parts of a state machine (e.g., states, transitions) are modelled as independent actions and the overall behaviour of the state machine is obtained by the parallel composition of its parts.

While the models of blocks use parallelism mostly to model the independence between blocks, the models of state machines, sequence diagrams and activities additionally use parallelism to model conjunctions of requirements. That is, the use of parallelism is not restricted to modelling parallel aspects (e.g., parallel regions in a state machine), but also to produce compositional models where the main elements of the notation are modelled by independent actions, which are then composed in parallel to yield the complete behaviour.

A consequence of this is that it is not desirable to eliminate the parallelism between blocks, thus the state-space reducing strategy focuses on state machines, activities and sequence diagrams, which are the SysML elements that make most usage of parallelism as conjunctions, that is, the kind of parallelism that we aim at eliminating to reduce the state-space of our models.

Block definition diagrams allow the declaration of blocks, which are the main modelling units in SysML used to define systems and their 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 4.1 and one concrete, shown in Figure 4.2, 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 4.1: Block definition diagram of abstract model

Figure 4.2: Block definition diagram of concrete model

Figure 4.1 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 4.3, it contains a single simple state. 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 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 4.2, 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 a port ip that provides 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.

The block Sec also has a single property, but two ports, port and ip. 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 ??.

The internal block diagram of the concrete model (Figure 4.4) shows the compos-
ite block CChronometer and its components (marked with \texttt{<<part>>}); it specifies that the port \texttt{port} of \texttt{Sec} is connected to the port of CChronometer, and that the ports \texttt{ip} and \texttt{ip Conj} are connected to each other. Finally, the blocks \texttt{Sec} and \texttt{Min} have each one state machine (Figure 4.5). The state machine of \texttt{Sec} is called \texttt{SecMain} and is similar to the state machine of the abstract model, except that it delegates the operations involving minutes to the block \texttt{Min}. These operations are treated by the state machine \texttt{MinMain} that contains a single state with two internal transitions that react to a call to the operations \texttt{minsReq} and \texttt{inc}. The first returns the value stored in \texttt{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 (\texttt{sec} and \texttt{min} in Figure 4.1), receptions (\texttt{tick}), which specify the signals that can be treated by the block, and operations (\texttt{time()}). Additionally, it may generalise other blocks, use and realise interfaces, and declare ports (\texttt{ip}, \texttt{ip Conj} and \texttt{port} in Figure 4.4), parts (\texttt{min} and \texttt{sec} in Figure 4.4) and references.

**State machine.** State machines contain states (\texttt{State} in Figure 4.3), which may be simple or composite, regions, transitions (the four arrows in Figure 4.3), junctions (the small black circle in Figure 4.3), joins, forks, history junctions, initial junctions (the larger black circle in Figure 4.3) 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 behaviour of a region has terminated.

Transitions allow the deactivation of one or more states, and the activation of
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 operation calls and signals; and
- **ext_op** and **ext_sig** channels for each port that also allow the receipt of operation call 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 4.5 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 com-
position of processes that model the block’s interface, the state machine that de-
scribes 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.

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 \( t_{\text{model}} \),
which takes a model as argument and characterises its corresponding CML speci-
fication. The formalisation of the semantics of SysML is in \([\text{ACC}+13]\).
Chapter 5

Refinement Strategy: goals
Chapter 6

Refinement Strategy: state machines
Chapter 7

Refinement Strategy: other SysML elements
Chapter 8

Conclusions
Part III

SysML Refinement Calculus
In [MLC13, LDC13, ACC+13] 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 [FL09], CSP [Hoa85, Ros98] and the refinement calculus [Mor94]. 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) [HJ98], which is a relational refinement framework. Circus has a refinement strategy [CSW03] with associated notions of refinement that can be directly adopted in the context of CML.

In this report 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 [CG06], take refinement as a syntactic notion based directly on UML components [PPG+03, Pon05, BRSV99], or do not focus on laws of refinement for model transformations as we do here [HHKT04, LLLJ04, VDS04, SBS09, DC03]. Our objective is to support sound model transformation at the diagrammatic level.

The structure of this part is as follows. Chapter 9 presents our notions of refinement. Chapter 10 discusses the limitations of SysML in supporting refinement and proposes extensions to overcome these limitations. Chapter 11 presents refinement laws by means of an example, and Chapter 12 provides a catalogue of refinement laws. Finally, Chapter 13 summarises our results and discusses related and future work.
Chapter 9

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 $M$ be a SysML model, and let $B_1$ and $B_2$
be blocks of \( \mathcal{M} \), then

\[
\mathcal{B}_1 \sqsubseteq_{\text{Block}} \mathcal{B}_2 \iff t_{-\text{model}}(\mathcal{M}).B_1 \sqsubseteq_p t_{-\text{model}}(\mathcal{M}).B_2
\]

That is, block \( \mathcal{B}_1 \) is refined by block \( \mathcal{B}_2 \) (written \( \mathcal{B}_1 \sqsubseteq_{\text{Block}} \mathcal{B}_2 \)) if, and only if, the CML process \( B_1 \) that models the block \( \mathcal{B}_1 \) is refined by the process \( B_2 \) that models \( \mathcal{B}_2 \). 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 \( (\approx_R \mathcal{M}) \) between blocks \( \mathcal{B}_1 \) and \( \mathcal{B}_2 \) of a SysML model \( \mathcal{M} \) is a relation \( R \) between \( \mathcal{B}_1.\text{PrivateProps} \) and \( \mathcal{B}_2.\text{PrivateProps} \) if, and only if, \( R \) is a forward simulation between the processes \( t_{-\text{model}}(\mathcal{M}).B_1 \) and \( t_{-\text{model}}(\mathcal{M}).B_2 \).

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.

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

\[
\mathcal{S}_1 \sqsubseteq_{\text{Stm}} \mathcal{S}_2 \iff t_{-\text{model}}(\mathcal{M}).S_1 \sqsubseteq_p t_{-\text{model}}(\mathcal{M}).S_2
\]

That is, a state machine \( \mathcal{S}_1 \) is refined by another state machine \( \mathcal{S}_2 \) (written \( \mathcal{S}_1 \sqsubseteq_{\text{Stm}} \mathcal{S}_2 \)) if, and only if, the CML process \( S_1 \) that models the state machine \( \mathcal{S}_1 \) is refined by the process \( S_2 \) that models \( \mathcal{S}_2 \).

Notions of refinement for states, 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 ??, 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.
Chapter 10

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.

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 4.2. 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, restric-
tions, 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 4.2. 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 4.2 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 4.4 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 4.4. 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 4.4.

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 4.3 and 4.5. The complete syntax of CML statements is in WCC+12.
Chapter 11
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.

Figure 11.1 presents the iterative refinement strategy of Circus. 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 4.2 is a refinement of the centralised abstract model in Figure 4.1.

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 [MC14].

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 4.5, 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 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 new blocks, Block1 and Block2, each with two ports (e.g., p1 and ip1), and each with its own state machine derived from one
**Law 3** Local operation introduction.

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

**Law 4** Region introduction.

of the regions $R_1$ and $R_2$. In this law, $I_1$ and $I_2$ 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 $I_1$ and $I_2$ (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 $I_1$ are not used in the triggers of the transitions of region $R_2$ and the required items of $I_1$ are not used in the actions of the states and transitions in $R_2$ (proviso 3), and the provided items of $I_2$ are not used in the triggers of the transitions of region $R_1$ and the required items of $I_2$ are not used in the actions of the states and transitions in $R_1$ (proviso 4).

Each block has an event pool where received events (operation calls and signals) are stored for processing. The proviso 2 of Law 5 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 [ACC+13, OMG11]), 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 pl and ip1. The ports pl 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 4.3), and (2) replace min in the record constructor mk_Time by the local variable. Finally, Law Operation call introduction [MC14] is applied twice, once to replace the assignment aux := min by a call to minReq via port ip_conj, and again to replace the assignment 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 [MC14].
**Law 5** Block decomposition.

1. \( I_1 \cap I_2 = \emptyset \land I_1 \cup I_2 \subseteq \text{Block} \)
2. \( \text{triggers}(R_1) \cap \text{triggers}(R_2) = \emptyset \land \text{usedV}(R_1) \cap \text{usedV}(R_2) = \emptyset \)
3. \( \text{provided}(I_1) \cap \text{trigger}(R_2) = \emptyset \land \text{required}(I_1) \cap \text{used}(R_2) = \emptyset \)
4. \( \text{provided}(I_2) \cap \text{trigger}(R_2) = \emptyset \land \text{required}(I_2) \cap \text{used}(R_2) = \emptyset \)

where

1. \( \text{Block1} \cap \text{Block2} = \emptyset \land \text{Block1} \cup \text{Block2} = \text{Block} \)
2. \( \text{provided}(II_1) = \text{Block1} \cap \text{used}(R_2) \land \text{required}(II_1) = \text{Block2} \cap \text{used}(R_1) \)
3. \( \text{provided}(II_2) = \text{Block2} \cap \text{used}(R_1) \land \text{required}(II_2) = \text{Block1} \cap \text{used}(R_2) \)
Chapter 12

Catalogue of SysML refinement laws

The complete refinement relies on other laws. For instance, the Law Block decomposition relies on the Law State machine partition that takes a state machine whose top level is formed by two regions whose triggers are disjoint, and replaces it with two separate state machines.

This refinement is possible because an event received by the original state machine is propagated to both regions for treatment, but since the triggers of the regions are disjoint, only one of them has the potential to treat it. The soundness of our refinement laws is further discussed in [MC14].

The behaviour of our abstract chronometer is defined by the state machine in Figure 4.3, it has a single state and responds to a signal tick and an operation call time. The signal tick is used as the trigger of a transition flow that increments the property sec and, if it is reset to zero, increment the property min. The operation call time triggers an internal transition that returns an instance of the Time datatype containing the values of the properties sec and min.

Whilst refining a state machine into two or more state machines is often a necessary step of a refinement strategy, in most cases it is an intermediate step by which we wish to decompose a block in two or more parts. The next law deals with this case.

The refinement between the abstract and concrete blocks can be verified by a similar strategy to that used for the verification of the state machines. In this case, instead of Law 6, Law 5 is used to obtain the model shown in Figures 4.2 and 4.4.
Law 6 State machine partition.

\[
\begin{array}{c}
\text{provided} \\
1. \text{triggers}(R1) \cap \text{triggers}(R2) = \emptyset
\end{array}
\]

Law 7 State entry action decomposition: sequential composition.

Whilst the use of the action language defined in Section ?? is convenient for specification, it is not as practical for concrete models. Existing CML refinement laws already support the refinement of abstract statements into concrete statements, but in the case of state machines it is often desirable to eliminate uncommon statements such as interleavings, guarded statements, internal and external choice. The laws in this section support the elimination of such statements by decomposing them through the hierarchy of states and transitions.

For instance, Law 7 applies to a simple state whose entry action is a CML statement of the form \(A_1; A_2\). This law allows the simple state to be converted into a composite state whose entry action is \(A_1\), and a new substate with entry action \(A_2\), an initial junction and a transition from the initial junction to the substate.

The soundness of this law relies on the fact that the only observable behaviours a state may contain are those specified in its actions. Since, no new action is introduced by the refinement law, the only requirement that remains is that the actions \(A_1\) and \(A_2\) are executed in the right order. This is guaranteed by the semantics of composite states in which the entry action of the parent state is executed (and completed) before the entry action of the child is executed.

Law 8 can similarly be applied to a simple state, but it requires that the entry action has the form \(A_1 \parallel|\parallel A_2\), that is, execute \(A_1\) and \(A_2\) in interleaving. This law replaces the simple state with a composite state containing two regions, each region containing an initial junction, a substate, and a transition from the initial junction to the substate. Furthermore, the substates have entry actions \(A_1\) and \(A_2\) respectively.
Law 8 State entry action decomposition: interleaving.

Law 9 Transition action decomposition: internal choice.

Similarly to Law 7, the soundness of this law relies on the observability of a state’s actions. Unlike the law for sequential composition, however, there is no requirement about the order in which the actions are executed, which is exactly the behaviour introduced by the parallel regions.

Law 9 on the other hand targets transitions; it applies to a transition whose action has the form $A_1 | A_2$, that is, choose and execute $A_1$ or $A_2$ nondeterministically. This law replaces the single transition by two transitions with the same source and target vertices (states, final states and junction), same trigger and guard, but different actions: one has $A_1$ as its action and the other $A_2$.

Comment: Need to modify the figure so that the transitions on the right hand side reach the same junction

The validity of this law relies on the fact that the only observable behaviours of a transition are those of the entry and exit actions of the source and target states, and those of the action itself. Since the source and target states are not changed, those behaviours are also not changed. The actions $A_1$ and $A_2$ are executed nondeterministically, which is still the case after the refinement is applied, due to the nondeterministic nature of transitions. That is, if two or more transitions from the same state are enabled at the same time, only one of them is executed, and the choice is non-deterministic.
Chapter 13

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, [CG06] 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 constrained with the notion of generalisation in [PPG+03], whilst in [Pon05], it is related (informally) to a formal notion of refinement as inspiration for transformation patterns. Bergner et al. [BRSV99] 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 [HHKT04, LLLJ04, VDS04, SBS09, DC03]; 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. [HHKT04] 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 [VDS04], where the notion of refinement is based on the observation of operation calls.
Liu et al. [LLLJ04] 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 [SBS09], refinement is explored in a formal variant of UML based on Event-B [Abr07]. This work differs from ours 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, [DC03] 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 [ACC+13 MLC13 LDC13] 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 use the CML theorem prover [FPC14] to formalise and mechanically verify our refinement laws.
Part IV

Refinement tool
Bibliography


