Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

COMPASS

Formal Refinement Support

Technical Note Number: D33.4

Version: 1.00

Date: September 2014

Public Document

http://www.compass-research.eu

Contributors:

Simon Foster, UY
Alvaro Miyazawa, UY

Editors:

Simon Foster, UY
Alvaro Miyazawa, UY

Reviewers:

Jan Peleska, UB
Zoe Andrews, Newcastle
Luís D. Couto, AU

Document History

Ver	Date	Author	Description
0.01	11-07-2014	Simon Foster	Initial document version
0.02	23-07-2014	Alvaro Miyazawa	Outline of the report
0.03	15-08-2014	Simon Foster	Added initial technical texts
0.04	28-08-2014	SF and AHM	Completed MiniMondex and introduction
1.00	29-09-2014	SF and AHM	Responded to comments

Contents

1 Introduction

This deliverable describes the implementation of the CML formal refinement tool. Refinement is a verification and formal development technique pioneered by Ralph-Johan Back[BW98] and Carroll Morgan[Mor90]. It is based on a behaviour preserving relation that allows the transformation of an abstract specification into more and more concrete models, potentially leading to an implementation. One of the key aspects of refinement is the reduction of non-determinism, which is a common abstraction mechanism used in specification languages.

CML provides various abstraction mechanisms including implicit operations, specification statements, non-deterministic choice etc. Its refinement calculus, which is derived from the *Circus* [WC01] refinement calculus [Oli06], supports both action refinement (as in CSP) and data refinement (as in Z and VDM), and provides a framework for the development and verification of distributed state-rich specifications.

The COMPASS refinement tool enables a user to apply refinement laws to a CML model, thus transforming it to a new model retaining existing behaviour whilst reducing non-determinism. This enables a developer to 1) refine a concrete constituent system from an abstract specification and 2) demonstrate that a given system satisfies (or does not satisfy) a suitable Systems of Systems (SoS) contract. The latter is explored in more detail in Section 3.2.

The tool ships with an initial set of refinement laws, including some of those described in [Oli06] and other sources. This initial set is extensible either by implementing them programmatically or by specification in *Maude* rewrite logic engine [CDE+99, CDE+07]. It is worth mentioning that while the refinement tool does not attempt to prove the refinement laws, the link between the refinement tool and the theorem prover [FP13] can record the dependency between the soundness of the refinement and the soundness of the laws, though the latter is out of the scope of this deliverable.

This deliverable is structured as follows. Section 2 describes how to install and use the tool. Section 3 describes two verification case studies developed in the context of task T3.3.4: a simple chronometer and a more complicated example based on a simplified version of the Mondex electronic purse [WSC+08]. The latter case study also provides an insight into the verification of emergent properties in an SoS. Section 4 describes the implementation of the tool as well as the steps necessary to extend the catalogue of refinement laws. Finally, Section 5 describes related tools, and Section 6 summarises our results, and discusses limitations and future work.

2 User Manual

This section describes the use of the refinement tool plug-in, which is distributed with the Symphony IDE (versions ¿= 0.3.6). This plug-in takes CML specifications and supports the application of refinement laws. This plug-in currently contains a number of simple laws, but can be extended with new refinement laws in two different ways: implementing the refinement law in Java or encoding the refinement law in Maude. In general, the latter option is simpler, more powerful, less error prone, and does not require recompilation of the plugin. However, for certain types of laws, for instance the copy rule that replaces an action call by its definition, it is easier to implement directly in Java due to the need to search the specification for the definition of the action being called. Details of how the refinement tool can be extended with new laws is given in Section 4.

It is important to notice that the use of Maude refinement laws is optional. In order to use Maude in the refinement tool, it is necessary to first install Maude. Maude is available for the three main platforms, Linux, OS X, and Windows; at time of writing the latest version is 2.6.

For Linux and Mac, please visit the Maude website at `http://maude.cs.uiuc.edu/download/`, download the binary archive, and then extract the archive to a suitable location. Alternatively, Maude is available in the *apt* repositories for both Debian and Ubuntu, and can be installed easily with `apt-get install maude`.

For Windows, please visit the *MOMENT* project website at `http://moment.dsic.upv.es/`, select the *Downloads* menu, and click on *Maude for Windows*, from which you can download the archive for Maude 2.6 on Windows. Run the downloaded executable to install Maude to a suitable location.

Once installed, Maude must be setup within Symphony. Configure the refinement tool by selecting menu *Windows* → *Preferences* → *Maude Setup* as shown in Figure 1. You must provide the path to the Maude binary or `.exe` file on Windows, and the path to the Maude encoding `cml-refine.maude` provided with the tool. Symphony will try and detect the location of this file and populate it in advance.

Next, the refinement perspective must be selected as shown in Figure 2, by selecting menu entry *Window* → *Open Perspective* → *Other...*.

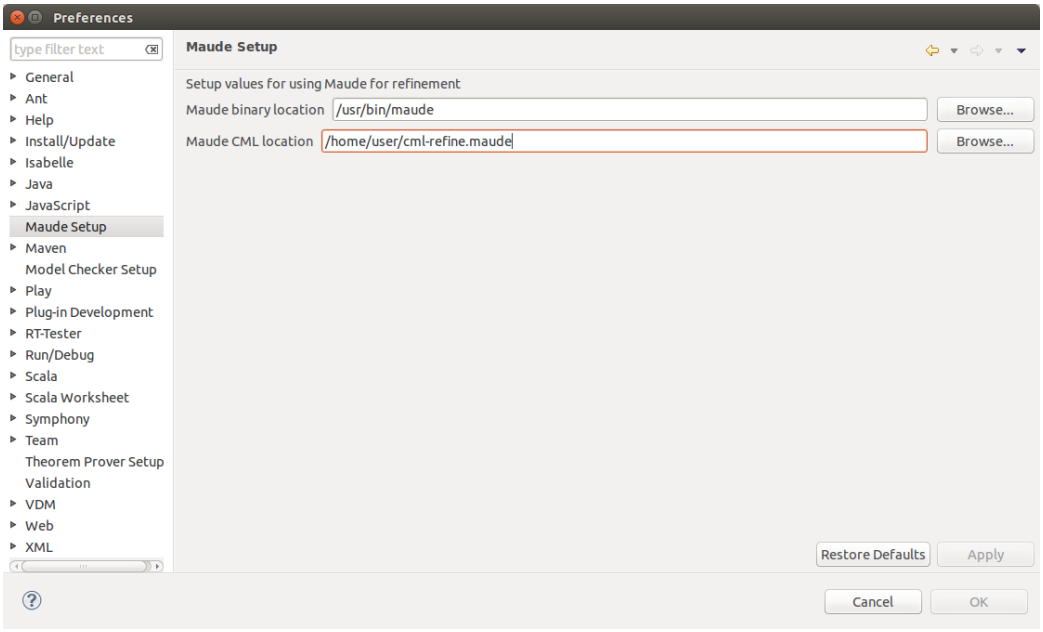

Figure 1: Maude configuration

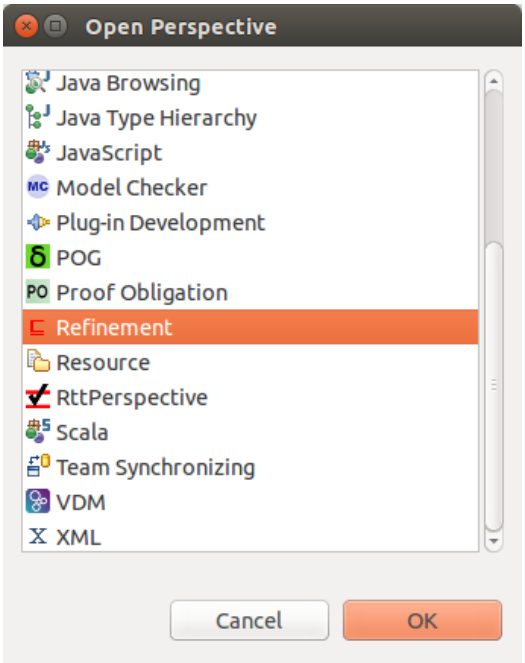

Figure 2: Selection of refinement perspective

The refinement perspective consists of six areas as shown in Figure 3: CML Explorer, CML Editor, Refinement Law Details, Refinement Laws, CML RPO List, and CML RPO Details. The first two are the same as in the CML perspective. The panel Refinement Laws presents the list of applicable refinement laws, the Refinement Law Details panel shows the details of the selected refinement law, the CML RPO List panel lists all the proof obligations (refinement provisos) generated by the application of refinement laws, and the RPO Details panel presents the details of a selected proof obligation.

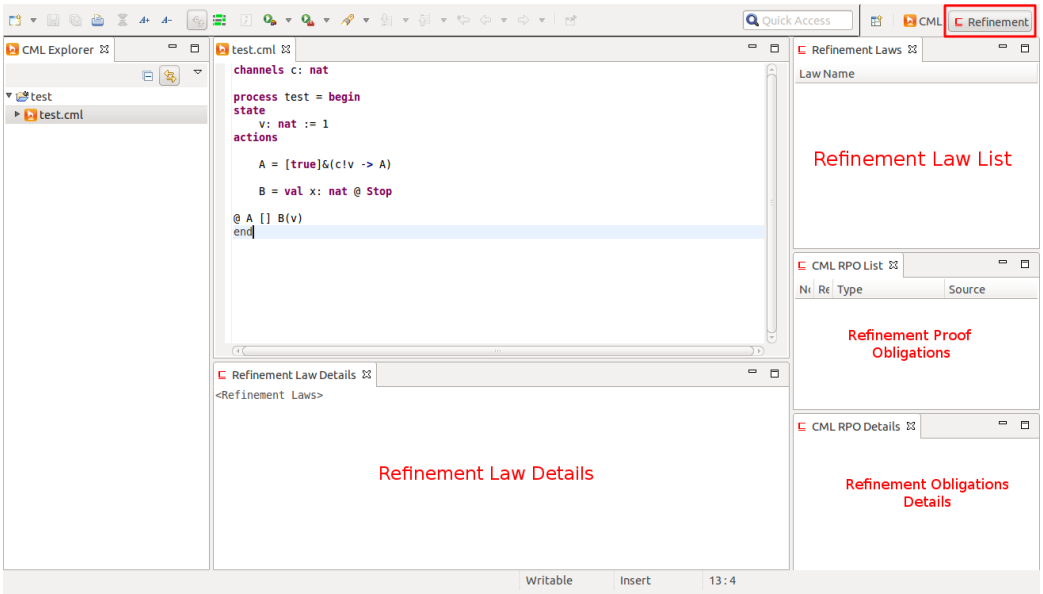

Figure 3: Refinement perspective

In order to apply a refinement law, the excerpt of the specification to which the law is to be applied must be selected as shown in Figure 4. The selection algorithm chooses the least common node of the beginning and end of the selection.

Next, the applicable laws can be searched by right clicking on the excerpt and selecting *Refinement* → *Refine* (or pressing *Ctrl+6*) as shown in Figure 5.

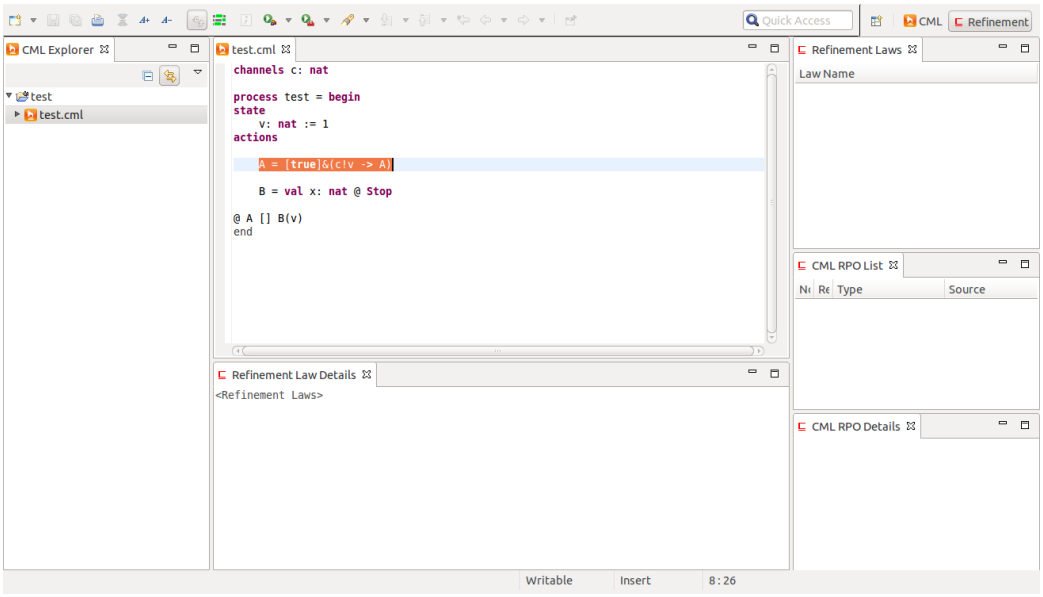

Figure 4: Specification excerpt selection

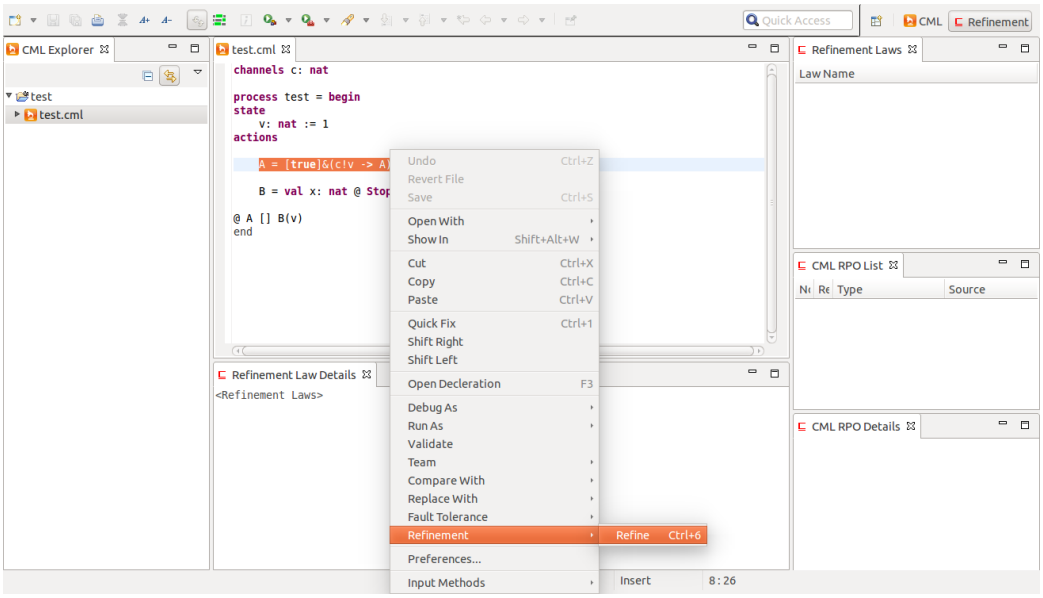

Figure 5: Searching for applicable laws

The applicable laws are then loaded onto the Refinement Laws panel, and a law may be selected as shown in Figure 6. In this case, the details of the law are presented in the bottom panel, and the law can be applied by double clicking it. In some cases application of a law may lead to a refinement proviso being raised which will be added to the refinement proviso list.

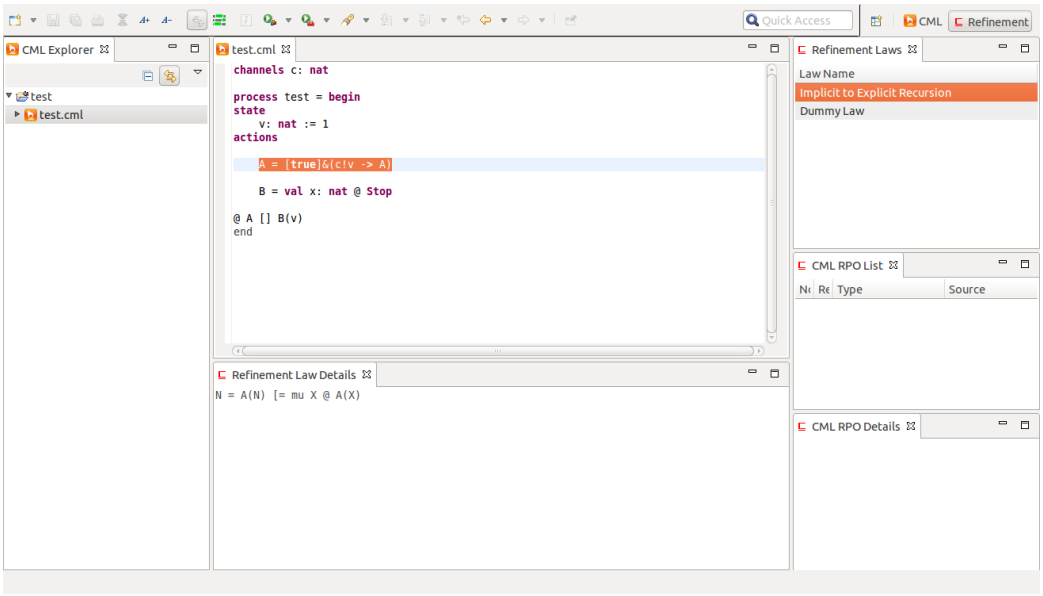

Figure 6: Selection of refinement law

The refined CML specification is shown in the CML Editor, and further refinement laws can be applied (Figure 7). In this example, a refinement law is applied to make the implicit recursion in `A` explicit using the `mu` operator.

3 Case Studies

In this section, we discuss two case studies for refinement in CML:

(1) Chronometer refinement. This example is a simple refinement that introduces a parallel implementation of a centralised specification. Its main goal is to demonstrate the usability of the tool.

(2) Distributed mini-Mondex. The mini-Mondex example is slightly more complex and aims at demonstrating the suitability of the refinement calculus for the verification of emergent properties of distributed systems. Due to its complexity, this example cannot be developed using Symphony, however, suitable extensions

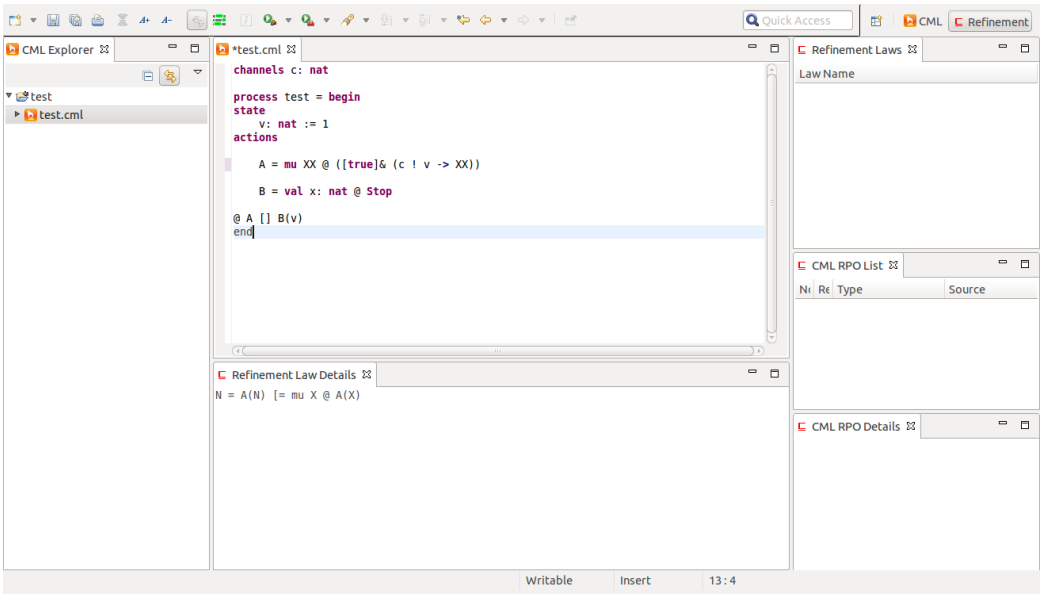

Figure 7: Application of refinement law

(data refinement and new refinement laws) of the refinement plugin suffice to support this example.

All the refinement laws used in these case studies, unless otherwise states, have been previously published in [Oli06]. Novel laws are presented in the body of the text.

3.1 Chronometer refinement

In this section, we present the example of a chronometer that is implemented through parallel processes. This example was first introduced in [Oli06], it specifies a centralised abstract chronometer `AChronometer` and a parallel concrete implementation `Chronometer`, and proves that `Chonometer` is a refinement of the abstract specification using the refinement calculus. An overview of this refinement is shown in Figure 8.

The abstract specification declares a type `RANGE` of natural numbers between 0 and 59 that is used to encode minutes and seconds.

```
types
        RANGE = nat inv i == i < 60
```

It also declares three channels: `tick`, `time` and `out`. The first marks the passage

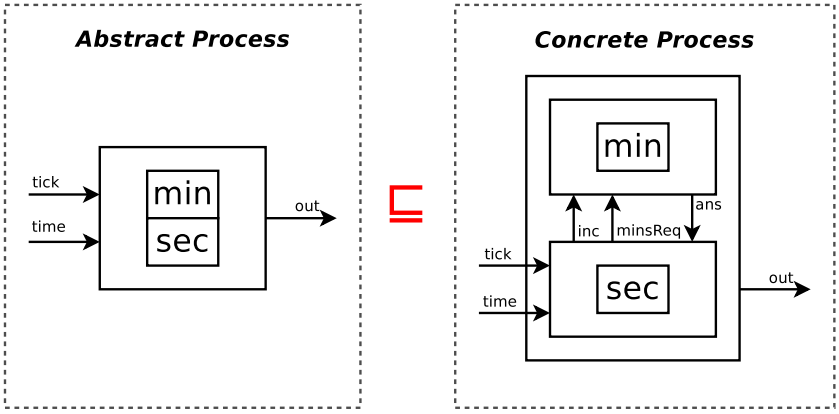

Figure 8: Refinement of the Chronometer

of 1 second, the second requests the current time, and the third answers to the request with the number of minutes and seconds that have elapsed.

```
channels
        tick, time
        out: RANGE*RANGE
```

The process that models the abstract chronometer is called `AChronometer` and declares a state formed by two components: `sec` and `min`. The processes declares four actions: `AInit`, `incSec`, `incMin` and `Run`. `AInit` initialises the state components, `incSec` increments `sec` and `incMin` increments `min`. The action `Run` offers a choice of synchronising on `tick` or `time`. In the first case, the seconds are incremented (using `incSec`) and if the result is zero the minutes are incremented, otherwise the action terminates. In the second case (synchronisation on `time`), the values stored in the state components are communicated through the channel `out`.

The behaviour of the process `AChronometer` is given the main action (after `@`), which initialises the state and starts a loop (**`mu`** `X @ F(X)`) that at each step calls the action `Run`.

```
process AChronometer = begin
state sec: RANGE
          min: RANGE

actions

AInit =
  [frame wr sec, min pre true post sec = 0 and min = 0]
```

```
incSec =
  [frame wr sec pre true post sec = (sec~ + 1) mod 60]

incMin =
  [frame wr min pre true post min = (min~ + 1) mod 60]

Run = tick -> incSec; (
                          [sec = 0] & incMin
                          []
                          [sec <> 0] & Skip
          )
      []
      time -> out!min!sec -> Skip

@ AInit; mu X @ (Run; X)
end
```

We wish to verify that this process is refined by a concrete process that controls the storage of seconds and minutes in parallel components. This concrete specification uses three extra channels: `inc`, `minsReq` and `ans`. The first is used to request that the minutes are incremented, the second is used to request the current value of the state component minutes, and the third is used to obtain the answer to the request of `minsReq`.

```
channels inc, minsReq
channels ans: RANGE
```

The parallel process declares the same state as the abstract process, and six actions: `SecInit` initialises the component `sec`, `MinInit` initialises `min`, `incSec` increments `sec`, `incMin` increments min, `RunSec` accepts requests to increment the time or output the elapsed time, and `RunMin` accepts request to increment the minutes or output the elapsed minutes.

The main action of this process is the parallel composition of the initialisation of `sec` followed by the iterative execution of `RunSec`, and the initialisation of `min` followed by the iterative execution of `RunMin` with the channels `inc`, `minsReq` and `ans` hidden (`\\`). The parallel composition (`[|_|_|_|]`) specifies that the left hand side can only modify the state components `sec`, the right hand side can only modify the state component `min` and both sides must synchronise on the channels `inc`, `minsReq` and `ans`.

This process reacts to `tick` or `time`, which are both offered by the left hand side of the parallelism. In the first case, this action increments the components `sec`, and, if the result of the increment is zero, requests through the channel `inc` that the

other parallel action increment `min`. In the second case, it requests, through the channels `minsReq` and `ans`, the current value of `min` and communicates it along with the value of `sec` through the channel `out`. The channels `inc`, `minsReq` and `ans` are only used for internal communications between the two parallel actions are internal and are, therefore, hidden.

```
process CChronometer = begin
state sec: RANGE
          min: RANGE

actions

SecInit = [frame wr sec pre true post sec = 0]

MinInit = [frame wr min pre true post min = 0]

incSec = [frame wr sec post sec = (sec~ + 1) mod 60]

incMin = [frame wr min post min = (min~ + 1) mod 60]

RunSec = tick -> incSec; (
                          [sec = 0] & inc -> Skip
                          []
                          [sec <> 0] & Skip
                    )
          []
          time -> minsReq ->
            ans?mins -> out!mins!sec -> Skip

RunMin = inc -> incMin [] minsReq -> ans!min -> Skip

@ (
         (SecInit(); mu X @ (RunSec; X))
                  [|{sec}|{|inc, minsReq, ans|}|{min}|]
         (MinInit(); mu X @ (RunMin; X))
)\\{|inc, minsReq, ans|}
end
```

In order to verify this refinement, we start from the abstract specification and, as a first step, add all the auxiliary actions of the concrete specification. This first step does not change the meaning of the abstract specification because these new actions are not used directly or indirectly in the main action of the abstract specification.

In what follows we highlight changes introduced by refinement laws in yellow, and actions that must be selected for the application of laws in gray. The yellow highlights refer to the laws described in the paragraph before the CML code, and the gray highlights refer to selections required for the laws described in the paragraph that follows the CML code.

```
process AChronometer = begin
state sec: RANGE
          min: RANGE

actions

AInit =
  [frame wr sec, min pre true post sec = 0 and min = 0]

SecInit = [frame wr sec pre true post sec = 0]

MinInit = [frame wr min pre true post min = 0]

incSec = [frame wr sec post sec = (sec~ + 1) mod 60]

incMin = [frame wr min post min = (min~ + 1) mod 60]

RunSec = tick -> incSec; (
                           [sec = 0] & inc -> Skip
                           []
                           [sec <> 0] & Skip
                    )
          []
          time -> minsReq ->
                      ans?mins -> out!mins!sec -> Skip
RunMin = inc -> incMin [] minsReq -> ans!min -> Skip

Run = tick -> incSec; (
                           [sec = 0] & incMin
                           []
                           [sec <> 0] & Skip
           )
      []
      time -> out!min!sec -> Skip
```

```
@ AInit; mu X @ (Run; X)
end
```

Next, we apply the law `Copy Rule from Left to Right`[1] to `AInit`, `Run`, `incSec` and `incMin`, in order to obtain the main action without any action calls.

```
([frame wr sec, min pre true post sec = 0 and min = 0]);
mu X @ (
  (tick -> (
    [frame wr sec post sec = (sec  + 1) mod 60]) ;
      ([sec = 0] &
        ([frame wr min post min = (min  + 1) mod 60])
          []
          [sec <> 0] & Skip)
  []
  time -> out!(min)!(sec) -> Skip);
X)

([frame wr sec, min pre true post sec = 0 and min = 0]);
mu X @ (
  (tick -> (
    [frame wr sec post sec = (sec~ + 1) mod 60]) ;
      ([sec = 0] &
        ([frame wr min post min = (min~ + 1) mod 60])
          []
          [sec <> 0] & Skip)
  []
  time -> out!(min)!(sec) -> Skip);
X)
```

Next, we apply the law `Unit of logical And 2` to transform the precondition of the first specification statement into a conjunction of `true`.

```
([frame wr sec, min pre true and true
  post sec = 0 and min = 0]);
mu X @ (
  (tick -> (
    [frame wr sec post sec = (sec~ + 1) mod 60]) ;
      ([sec = 0] &
        ([frame wr min post min = (min~ + 1) mod 60])
```

[1] It is worth mentioning that the early application of copy rules is sometimes unnecessary from the point of view of the refinement, but necessary in our case due to a limitation of the tool in the calculation of functions such as `usedC` that require the complete definition of the action passed as parameter.

```
            []
            [sec <> 0] & Skip)
  []
  time -> out!(min)!(sec) -> Skip);
X)
```

This is necessary to put the action in the correct form for the application of the next law, which is `Specification Sequential Introduction`.

```
([frame wr sec, min pre true and true
  post sec = 0 and min = 0]);
mu X @ (
  (tick -> (
    [frame wr sec post sec = (sec~ + 1) mod 60]) ;
      ([sec = 0] &
        ([frame wr min post min = (min~ + 1) mod 60])
          []
          [sec <> 0] & Skip)
  []
  time -> out!(min)!(sec) -> Skip);
X)
```

The law `Specification Sequential Introduction` is applied to the first specification statement and breaks it into two statements in sequential composition.

```
([frame wr min pre true post min = 0] ;
[frame wr sec pre true post sec = 0]);
mu X @ ((
  tick -> (
    [frame wr sec post sec = (sec  + 1) mod 60]) ;
      ([sec = 0] &
        ([frame wr min post min = (min  + 1) mod 60])
      []
      [sec <> 0] & Skip)
  []
  time -> out!(min)!(sec) -> Skip);
X)
```

Next, we refine the recursion (`mu X @ ...`) into a parallelism of recursions using the law `Parallel Recursion Distribution with Hiding` with parameters `ns2 = {min}, cs = {|inc,minsReq,ans|}, ns1 = {sec},`

`A2 = RunMin`, `A1 = RunSec` and `Sync = {|inc,minsReq,ans|}`. This law is a specialisation of the fixed point law in [Oli06] and, as such, generates a proviso of the form `A [= B`, which can be verified through model-checking, theorem proving or the refinement tool via a separate refinement.

```
([frame wr min  pre true post min = 0] ;
[frame wr sec  pre true post sec = 0]);
((
   (mu X @ (RunSec ; X))
   [| sec | | ans, inc, minsReq | | min |]
   (mu X @ (RunMin ; X))
) \\ {| ans, inc, minsReq |})
```

```
([frame wr min  pre true post min = 0] ;
[frame wr sec  pre true post sec = 0]);
((
  (mu X @ (RunSec ; X))
  [| {sec} | {| ans, inc, minsReq |} | {min} |]
  (mu X @ (RunMin ; X))
) \\ {| ans, inc, minsReq |})
```

Again, we apply the law `Copy Rule from Left to Right` to remove the calls to `RunSec`, `RunMin`.

```
([frame wr min  pre true post min = 0] ;
 [frame wr sec  pre true post sec = 0]) ;
   ((((mu X @ ((
     time -> minsReq -> ans?mins -> out!mins!sec -> Skip
     []
     tick -> incSec ; (
        [sec = 0]& inc -> Skip
        []
        [sec <> 0]& Skip
     )
   ) ; X))
   [| {sec} | {| ans, inc, minsReq |} | {min} |]
   (mu X @ ((
     inc -> incMin
     []
     minsReq -> ans!min -> Skip
   ) ; X))
   )
```

```
) \\ {| ans, inc, minsReq |})

([frame wr min  pre true post min = 0] ;
 [frame wr sec  pre true post sec = 0]) ;
   ((((mu X @ ((
     time -> minsReq -> ans?mins -> out!mins!sec -> Skip
     []
     tick -> incSec ; (
       [sec = 0]\& inc -> Skip
       []
       [sec <> 0]\& Skip
     )
   ) ; X))
   [| {sec} | {| ans, inc, minsReq |} | {min} |]
   (mu X @ ((
     inc -> incMin
     []
     minsReq -> ans!min -> Skip
   ) ; X))
   )
) \\ {| ans, inc, minsReq |})
```

One more time, we apply the law `Copy Rule from Left to Right` to remove the calls to `incSec` and `incMin`.

```
([frame wr min  pre true post min = 0] ;
 [frame wr sec  pre true post sec = 0]) ;
   ((((mu X @ ((
     time -> minsReq -> ans?mins -> out!mins!sec -> Skip
     []
     tick ->
       ( [frame wr sec post sec = (sec  + 1) mod 60] );(
         [sec = 0]& inc -> Skip
         []
         [sec <> 0]& Skip
       )
   ) ; X))
   [| {sec} | {| ans, inc, minsReq |} | {min} |]
   (mu X @ ((
     inc ->
       ( [frame wr min post min = (min  + 1) mod 60] )
     []
     minsReq -> ans!min -> Skip
```

```
    ) ; X))
    )
) \\ {| ans, inc, minsReq |})
```

```
([frame wr min pre true post min = 0] ;
    [frame wr pre true sec post sec = 0]) ;
    ((((mu X @ ((
       time -> minsReq -> ans?mins -> out!mins!sec -> Skip
       []
       tick ->
          ([frame wr sec post sec = (sec  + 1) mod 60]);(
             [sec = 0]& inc -> Skip
             []
             [sec <> 0]& Skip
          )
    ) ; X))
    [| {sec} | {| ans, inc, minsReq |} | {min} |]
    (mu X @ ((
       inc ->
          ([frame wr min post min = (min  + 1) mod 60])
       []
       minsReq -> ans!min -> Skip
    ) ; X))
    )
) \\ {| ans, inc, minsReq |})
```

Next, we apply the law `Distribute hiding over sequential composition` to the whole main action.

```
((([frame wr min pre true post min = 0] ;
    [frame wr sec pre true post sec = 0]) ;
    ((mu X @ ((
       time -> minsReq -> ans?mins -> out!mins!sec -> Skip
       []
       tick ->
          ([frame wr sec post sec = (sec  + 1) mod 60]);(
             [sec = 0]& inc -> Skip
             []
             [sec <> 0]& Skip
          )
```

```
    ) ; X))
    [| {sec} | {| ans, inc, minsReq |} | {min} |]
    (mu X @ ((
      inc ->
        ([frame wr min post min = (min  + 1) mod 60])
      []
      minsReq -> ans!min -> Skip
    ) ; X))
    )
) \\ {| ans, inc, minsReq |})

((([frame wr min pre true post min = 0] ;
    [frame wr sec pre true post sec = 0]) ;
    ((mu X @ ((
      time -> minsReq -> ans?mins -> out!mins!sec -> Skip
      []
      tick ->
        ([frame wr sec post sec = (sec  + 1) mod 60]);(
          [sec = 0]& inc -> Skip
          []
          [sec <> 0]& Skip
        )
    ) ; X))
    [| {sec} | {| ans, inc, minsReq |} | {min} |]
    (mu X @ ((
      inc ->
        ([frame wr min post min = (min  + 1) mod 60])
      []
      minsReq -> ans!min -> Skip
    ) ; X))
    )
)\\ {| ans, inc, minsReq |})
```

Now, we apply the law `Sequential Composition Associativity 2` to the whole action except the hiding (`\\{|...|}`) in order to group the second specification statement and the parallel action together.

```
((([frame wr min pre true post min = 0] ;
    ([frame wr sec pre true post sec = 0] ;
```

```
    ((mu X @ ((
       time -> minsReq -> ans?mins -> out!mins!sec -> Skip
       []
       tick ->
          ([frame wr sec post sec = (sec  + 1) mod 60]);(
             [sec = 0]& inc -> Skip
             []
             [sec <> 0]& Skip
          )
    ) ; X))
    [| {sec} | {| ans, inc, minsReq |} | {min} |]
    (mu X @ ((
       inc ->
          ([frame wr min post min = (min  + 1) mod 60])
       []
       minsReq -> ans!min -> Skip
    ) ; X))
    ))
)\\ {| ans, inc, minsReq |})

(([frame wr min  pre true post min = 0] ;
    ([frame wr sec pre true post sec = 0] ;
    ((mu X @ ((
       time -> minsReq -> ans?mins -> out!mins!sec -> Skip
       []
       tick ->
          ([frame wr sec post sec = (sec  + 1) mod 60]);(
             [sec = 0]& inc -> Skip
             []
             [sec <> 0]& Skip
          )
    ) ; X))
    [| {sec} | {| ans, inc, minsReq |} | {min} |]
    (mu X @ ((
       inc ->
          ([frame wr min post min = (min  + 1) mod 60])
       []
       minsReq -> ans!min -> Skip
    ) ; X))
```

```
    ))
)\\ {| ans, inc, minsReq |})
```

Next, we apply the law `Distribute sequential composition over parallelism on the left hand side` to the sequential composition of the second specification statement and the parallel action to move the specification statement inside the parallel action.

```
(([frame wr min  pre true post min = 0] ;
    (([frame wr sec pre true post sec = 0] ;
    (mu X @ ((
       time -> minsReq -> ans?mins -> out!mins!sec -> Skip
       []
       tick ->
          ([frame wr sec post sec = (sec  + 1) mod 60]);(
            [sec = 0]& inc -> Skip
            []
            [sec <> 0]& Skip
          )
    ) ; X)))
    [| {sec} | {| ans, inc, minsReq |} | {min} |]
    (mu X @ ((
      inc ->
        ([frame wr min post min = (min~ + 1) mod 60])
      []
      minsReq -> ans!min -> Skip
    ) ; X))
    ))
)\\ {| ans, inc, minsReq |})
```

```
(([frame wr min pre true post min = 0] ;
    (([frame wr sec pre true post sec = 0] ;
    (mu X @ ((
       time -> minsReq -> ans?mins -> out!mins!sec -> Skip
       []
       tick ->
          ([frame wr sec post sec = (sec  + 1) mod 60]);(
            [sec = 0]& inc -> Skip
            []
            [sec <> 0]& Skip
          )
```

```
    ) ; X)))
    [| sec | {| ans, inc, minsReq |} | min |]
    (mu X @ ((
      inc ->
        ([frame wr min post min = (min  + 1) mod 60])
      []
      minsReq -> ans!min -> Skip
    ) ; X))
    ))
)\\ {| ans, inc, minsReq |})
```

Similarly, we apply the law `Distribute sequential composition over parallelism on the right hand side` to the sequential composition of the remaining specification statement and the parallel action to move the specification statement to the right hand side of the parallel action.

```
(((
  ([frame wr sec  pre true post sec = 0] ;
  (mu X @ (
    (time -> minsReq -> ans?mins -> out!mins!sec -> Skip
    []
    tick -> [frame wr sec post sec = (sec~ + 1) mod 60] ; (
      [sec = 0]& inc -> Skip
      []
      [sec <> 0]& Skip)
    ) ; X)
  ))
  [| {sec} | {| ans, inc, minsReq |} | {min} |]
  ([frame wr min pre true post min = 0] ;
  (mu X @ (
    (inc ->
      ([frame wr min post min = (min  + 1) mod 60])
    []
    minsReq -> ans!min -> Skip) ;
  X)))
)) \\ {| ans, inc, minsReq |})

(((
  ([frame wr sec pre true post sec = 0] ;
  (mu X @ (
    (time -> minsReq -> ans?mins -> out!mins!sec -> Skip
    []
```

```
    tick -> [frame wr sec post sec = (sec  + 1) mod 60] ; (
      [sec = 0]& inc -> Skip
      []
      [sec <> 0]& Skip)
    ) ; X)
  ))
  [| {sec} | {| ans, inc, minsReq |} | {min} |]
  ([frame wr min pre true post min = 0] ;
  (mu X @ (
    (inc ->
      ([frame wr min post min = (min  + 1) mod 60])
    []
    minsReq -> ans!min -> Skip) ;
  X)))
)) \\ {| ans, inc, minsReq |})
```

Finally, we apply the law `Copy Rule from Right to Left` to the specification statements: the first with name `SecInit`, the second with name `incSec`, the third with name `MinInit`, and the fourth with name `incMin`.

```
(((
  (SecInit ;
  (mu X @ (
    (time -> minsReq -> ans?mins -> out!mins!sec -> Skip
    []
    tick -> incSec ; (
      [sec = 0]& inc -> Skip
      []
      [sec <> 0]& Skip)
    ) ; X)
  ))
  [| {sec} | {| ans, inc, minsReq |} | {min} |]
  (MinInit ;
  (mu X @ (
    (inc ->
      (incMin)
    []
    minsReq -> ans!min -> Skip) ;
  X)))
)) \\ {| ans, inc, minsReq |})

(((
  ({SecInit ;
```

```
  (mu X @ (
    (time -> minsReq -> ans?mins -> out!mins!sec -> Skip
    []
    tick -> incSec ; (
      [sec = 0]& inc -> Skip
      []
      [sec <> 0]& Skip)
    ) ; X)
  ))
  [| {sec} | {| ans, inc, minsReq |} | {min} |]
  (MinInit ;
  (mu X @ (
    (inc -> incMin
    []
    minsReq -> ans!min -> Skip) ;
  X)))
)) \\ {| ans, inc, minsReq |})
```

We also apply the law `Copy Rule from Right to Left` to the body of the recursions introducing the action calls, for the left hand side, `RunSec` and, for the right hand side, `RunMin`. This steps produce the following action, which apart from parentheses is the same as the main action of the concrete process, thus demonstrating that one is the refinement of the other.

```
(((
  (SecInit ; (mu X @ (RunSec ; X)))
  [| {sec} | {| ans, inc, minsReq |} | {min} |]
  (MinInit ; (mu X @ (RunMin ; X)))
)) \\ {| ans, inc, minsReq |})
```

This example has been fully developed in the refinement plugin, but no attempt was made to verify the provisos that were not discharged automatically by the refinement tool.

3.2 Distributed mini-Mondex

Unlike the previous case study, the mini-Mondex example cannot be fully developed using the refinement plugin. It is, nevertheless, an important case study in that it demonstrates the suitability of the refinement calculus for the verification of emergent properties.

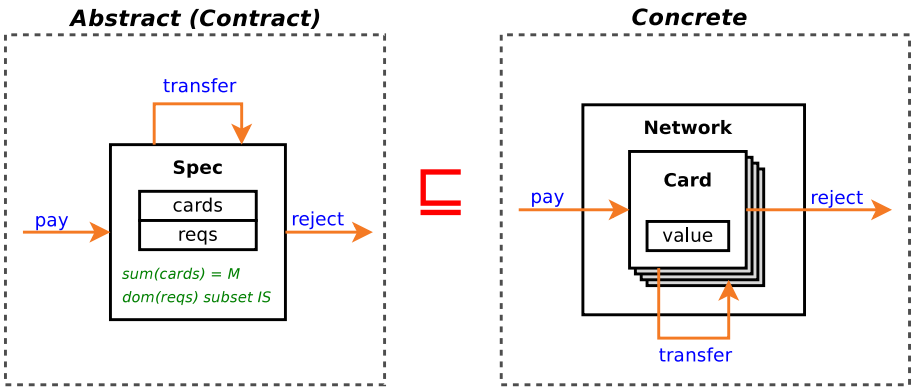

Figure 9: The Mini-Mondex process refinement

The Mini-Mondex case study models a card system, where the main desirable properties are that (1) money is not created or lost during the operation of the system, and (2) that the implementation of the system be based on fully distributed cards, where each card records its available balance.

Whilst it is feasible to specify property 1 in a centralised specification as an invariant of the state, it is not possible to do so in a distributed specification in which the amount of money available to each card is only visible to the card itself.

Our approach to obtain a distributed system that preserves property 1 is to start from a centralised specification that enforces this property and refine it into the distributed version, whilst preserving the invariants, as shown in Figure 9. The centralised specification is shown below.

```
values
 N: nat = 2
 V: nat = 10
 M: nat = N*V
```

It declares the number `N` of cards, the initial balance of each card `V`, the total amount of money in the system `M`.

```
types
  Index = nat
    inv i == i in set {1,...,N}
  Money = nat
    inv m == m in set {0,...,M}

  TransReq = Index * Index * Money
```

Next, three types are declared: `Index` encodes the indices of the cards, `Money` the possible balance values, and `TransReq` encodes a transference request formed by the index of the payer, the index of the payee, and the amount of money being transferred.

```
functions
  initseq: nat -> seq of nat
  initseq(n) == [V | i in set {1,...,n}]

  sum: seq of int -> int
  sum(xs) == if (xs = []) then 0 else hd(xs) + sum(tl(xs))
    measure sumM
```

Two auxiliary functions are declared. The first initialises a sequence with the constant `V`, and the second sums the elements of a sequence.

```
channels
  pay, transfer: Index * Index * Money
```

The specification uses three channels: `pay`, `transfer` and `reject`. The channel `pay` is used to request a payment to be made, `transfer` is used to complete a payment request, and `reject` is used to indicate that the payment cannot be completed.

The process that specifies the system declares two state components: `cards` and `reqs`. The first is a sequence of `Money` and stores the balance of each card (of type `Index`), and the second is a mapping of `TransReq` that records the transference requests that have not yet been completed or rejected. The state invariant is specified that requires the sum of the balances of all cards to be equal to `M`, that the originator of a transfer request to be one of the cards, and that it is not possible for a card to transfer money to itself.

```
process Spec =
begin
  state
    cards : seq of Money := initseq(N)
    reqs  : map Index to TransReq := {|->}
  inv
    sum(cards) = M and
    dom(reqs) subset inds cards and
    forall mk_(i,j,-) in set rng reqs @ i <> j
```

Three auxiliary operations are declared to simplify the manipulation of the state. `AddTranReq` adds a transference request to the map `reqs`, `RemReq` removes a

request from `reqs`, and `MkTran` executes a transference request by withdrawing money from the payer's card and adding it to the payees' card.

```
operations
  AddTranReq: Index * Index * Money ==> ()
  AddTranReq(i, j, n) == reqs:=reqs++{i|->mk_(i, j, n)}
  pre i <> j

  RemReq: Index ==> ()
  RemReq(i) == reqs := {i} <-: reqs

  MkTran: Index ==> ()
  MkTran(i) == atomic(cards(i) := cards(i) - reqs(i).#3;
    cards(reqs(i).#2) := cards(reqs(i).#2) + reqs(i).#3);
    RemReq(i)
```

Next, three auxiliary actions are declared, each specifying the possible interactions with the system. The first, `RecTranReq`, offers the possibility of requesting a payment through the channel `pay` from a card `i`, `RejTran` rejects any requests from card `i` that cannot be completed due to insufficient funds, and `EffTran` executes any requests from card `i` that can be completed.

```
actions
  RecTranReq = i: Index @ [i not in set dom(reqs)] &
    pay.i?j:(i <> j)?n:(n > 0) -> AddTranReq(i, j, n)

  RejTran    = i: Index @
    [i in set dom(reqs) and reqs(i).#3 > cards(i)] &
      reject!i -> RemReq(i)

  EffTran    = i: Index @
    [i in set dom(reqs) and reqs(i).#3 <= cards(i)] &
      transfer.i.(reqs(i).#2).(reqs(i).#3) -> MkTran(i)
```

Finally, the overall behaviour of the process is specified by a recursive action that at each step offers, for each of the cards, a choice of the three actions above: receiving, rejecting or executing a request.

```
  @ mu X @ (
    ([] i : Index @
      RecTranReq(i) [] RejTran(i) [] EffTran(i)
    ); X
  )
end
```

As a first step of the refinement, we simplify the model using the constant values such as N and V.

```
process Spec1 =
begin
  state
    cards : seq of Money := [10,10]
    reqs  : map Index to TransReq := {|->}
  inv
    cards(1) + cards(2) = M and
    dom(reqs) subset  {1,...,N}
    forall mk_(i,j,-) in set rng reqs @ i <> j

  operations

    AddTranReq: Index * Index * Money ==> ()
    AddTranReq(i, j, n) == reqs:=reqs++{i|->mk_(i, j, n)}

    RemReq: Index ==> ()
    RemReq(i) == reqs := {i} <-: reqs

    MkTran: Index ==> ()
    MkTran(i) == atomic(cards(i) := cards(i) - reqs(i).#3;
      cards(reqs(i).#2) := cards(reqs(i).#2) + reqs(i).#3);
      RemReq(i)

  actions
    RecTranReq = i: Index @ [i not in set dom(reqs)] &
            pay.i?j:(i <> j)?n:(n > 0) -> AddTranReq(i, j, n)

    RejTran    = i: Index @
      [i in set dom(reqs) and reqs(i).#3 > cards(i)] &
        reject!i -> RemReq(i)

    EffTran    = i: Index @
      [i in set dom(reqs) and reqs(i).#3 <= cards(i)] &
        transfer.i.(reqs(i).#2).(reqs(i).#3) -> MkTran(i)

  @ mu X @ (
    ([] i : Index @
      RecTranReq(i) [] RejTran(i) [] EffTran(i)
    ); X
  )
end
```

Next, we data refine the specification. The abstract state is

```
[cards: seq of Money, reqs: map Index to TransReq]
```

The concrete state encodes the sequence of size 2 of the abstract models as two state components `card1` and `card2`, and the map as eight state components that identify whether there is a transaction request associated with the card N (`hasN`), who is the source (`srcN`), who is the target (`tgtN`), and how much money is to be transferred (`mN`). It is shown below

```
[card1: Money, card2: Money, has1: bool, src1: Index,
tgt1: Index, m1: Money, has2: bool, src2: Index,
tgt2: Index, m2: Money]
```

The retrieve relation formalises the relationship between abstract and concrete states and is as follows.

```
reqs = {1 |-> mk_(1,y,z) | y <- Index, z <- Money @
          has1 and y = tgt1 and z = m1}
       munion
       {2 |-> mk_(2,y,z) | y <- Index, z <- Money @
          has2 and y = tgt2 and z = m2}

cards = [card1,card2]
```

The retrieve relation specifies that a request from card 1 is in the map `reqs` if and only if the concrete component `has1`, and in this case the second component of the request is equal to `tgt1` and the third component equals `m1`. The case for requests from card 2 is similarly specified. The sequence of cards is simply the concatenation of the concrete components `card1` and `card2`. The result of data refining the process is shown below.

```
process Spec2 =
begin

  state
    card1: Money := 10
    card2: Money := 10

    has1: bool := false
    src1: Index
    tgt1: Index
    m1: Money

    has2: bool := false
```

```
    src2: Index
    tgt2: Index
    m2: Money

  inv
    card1 + card2 = 20 and
    (has1 => src1 in set {1,...,2}) and
    (has2 => src2 in set {1,...,2}) and
    (has1 => src1 <> tgt1) and (has2 => src2 <> tgt2)

  operations
    AddTranReq: Index * Index * Money ==> ()
    AddTranReq(i, j, n) == cases i:
      1 -> has1 := true; src1 := i; tgt1 := j; m1 := n,
      2 -> has2 := true; src2 := i; tgt2 := j; m2 := n
    end

    RemReq: Index ==> ()
    RemReq(i) == cases i:
      1 -> has1 := false,
      2 -> has2 := false
    end

    MkTran: Index ==> ()
    MkTran(i) == cases i:
      1 -> card1 := card1 - m1;
         cases tgt1:
              1 -> card1 := card1 + m1,
              2 -> card2 := card2 + m1
         end; RemReq(1),
      2 -> card2 := card2 - m2;
         cases tgt2:
              1 -> card1 := card1 + m2,
              2 -> card2 := card2 + m2
         end; RemReq(2)
    end

  actions
   RecTranReq = i: Index @
     [i = 1 => not has1 and i = 2 => not has2] &
       pay.i?j:(i <> j)?n:(n > 0) -> AddTranReq(i, j, n)

    RejTran    = i: Index @
```

```
      [((i = 1 and has1) or (i = 2 and has2)) and
        ((i = 1 and m1 > card1) or
        (i = 2 and m2 > card2))] &
          reject!i -> RemReq(i)

    EffTran     = i: Index @
      [((i = 1 and has1) or (i = 2 and has2)) and
        ((i = 1 and m1 <= card1) or
        (i = 2 and m2 <= card2))] &
          ([i = 1] & transfer.(1).tgt1.m1 -> MkTran(1)
        []
        [i = 2] & transfer.(2).tgt2.m2 -> MkTran(2))

  @ mu X @ (
    ([] i : Index @
      RecTranReq(i) [] RejTran(i) [] EffTran(i)
    ); X
  )
end
```

Next, by applying the copy-rule to the operation calls and further simplifying the model, we obtain the following actions.

```
...
actions
  RecTranReq = i: Index @
    [i = 1 => not has1 and i = 2 => not has2] &
        pay.i?j:(i <> j)?n:(n > 0) -> (
        cases i:
            1 -> has1 := true; src1 := i; tgt1 := j; m1 := n,
            2 -> has2 := true; src2 := i; tgt2 := j; m2 := n
          end
        )

    RejTran     = i: Index @
      [((i = 1 and has1) or (i = 2 and has2)) and
        ((i = 1 and m1 > card1) or
        (i = 2 and m2 > card2))] &
        reject!i -> (
          cases i:
              1 -> has1 := false,
              2 -> has2 := false
            end
        )
```

```
    EffTran    = i: Index @
      [((i = 1 and has1) or (i = 2 and has2)) and
        ((i = 1 and m1 <= card1) or
        (i = 2 and m2 <= card2))] &
          ([i = 1] & transfer.(1).tgt1.m1 ->
                card1 := card1 - m1;
            cases tgt1:
              1 -> card1 := card1 + m1,
              2 -> card2 := card2 + m1
             end; has1 := false
          []
          [i = 2] & transfer.(2).tgt2.m2 ->
                card2 := card2 - m2;
            cases tgt2:
              1 -> card1 := card1 + m2,
              2 -> card2 := card2 + m2
             end; has2 := false
            )

  @ mu X @ (
    ([] i : Index @
      RecTranReq(i) [] RejTran(i) [] EffTran(i)
    ); X
  )
end
```

At this point, we expand the external choices over the set of indices, apply the copy rule to replace the calls to the auxiliary actions, and further simplify the specification obtaining the following main action.

```
  @ mu X @ ((
    [not has1] & pay.1?j:(1 <> j)?n:(n > 0) ->
      has1 := true; src1 := 1; tgt1 := j; m1 := n
    []
      [has1 and m1 > card1] & reject!1 -> has1 := false
      []
      [has1 and m1 <= card1] & transfer.(1).tgt1.m1 ->
        card1 := card1 - m1; (
         [tgt1 = 1] & card1 := card1 + m1; has1 := false
         []
         [tgt1 = 2] & card2 := card2 + m1; has1 := false
      )
      []
```

```
      [not has2] & pay.2?j:(2 <> j)?n:(n > 0) ->
        has2 := true; src2 := 2; tgt2 := j; m2 := n
      []
      [has2 and m2 > card2] & reject!2 -> has2 := false
      []
      [has2 and m2 <= card2] & transfer.(2).tgt2.m2 ->
        card2 := card2 - m2; (
         [tgt2 = 1] & card1 := card1 + m2; has2 := false
         []
         [tgt2 = 2] & card2 := card2 + m2; has2 := false
      )
); X)
```

Next, we use the invariant that forbids self requests to remove the choices guarded by `tgt1 = 1` and `tgt2 = 2`. This is necessary because the expansion of the operations includes options that do not occur under certain circumstances, but cannot be eliminated without explicit reference to the invariant. In order to eliminate the unreachable operations, we introduce the invariant as an assumption and distribute it through the action until it reaches the guards introduced by the expansion of the operations. If the assumption falsifies the guard, it is possible to remove that branch of the external choice.

```
@ mu X @ ((
  [not has1] & pay.1?j:(1 <> j)?n:(n > 0) ->
    has1 := true; src1 := 1; tgt1 := j; m1 := n
[]
  [has1 and m1 > card1] & reject!1 -> has1 := false
    []
    [has1 and m1 <= card1 and tgt1 = 2] &
      transfer.(1).(2).m1 -> card1 := card1 - m1;
      card2 := card2 + m1; has1 := false
    []
    [not has2] & pay.2?j:(2 <> j)?n:(n > 0) ->
      has2 := true; src2 := 2; tgt2 := j; m2 := n
    []
    [has2 and m2 > card2] & reject!2 -> has2 := false
    []
    [has2 and m2 <= card2 and tgt2 = 1] &
      transfer.(2).(1).m2 -> card2 := card2 - m2;
      card1 := card1 + m2; has2 := false
  ); X)
```

In the next step, we parallelise the assignment to `card1` and `card2` of the two choices prefixed by `transfer`.

Law seq-par-intro

```
c -> v1 := e1; v2 := e2
[=
(c -> v1 := e1 [|{v1}|{|c|}|{v2}|] c -> v2 := e2)

provided
  v1 <> v2, v1 not in set usedV(e2) and
  v2 not in set usedV(e1)
```

The novel refinement law above is used twice to complete this step.

```
  @ mu X @ ((
    [not has1] & pay.1?j:(1 <> j)?n:(n > 0) ->
      has1 := true; src1 := 1; tgt1 := j; m1 := n
    []
    [has1 and m1 > card1] & reject!1 -> has1 := false
    []
    [has1 and m1 <= card1 and tgt1 = 2] & (
      transfer.(1).(2).m1 -> card1 := card1 - m1
      [|{card1}|{|transfer|}|{card2}|]
      transfer.(1).(2)?x -> card2 := card2 + x
    ); has1 := false
    []
    [not has2] & pay.2?j:(2 <> j)?n:(n > 0) ->
      has2 := true; src2 := 2; tgt2 := j; m2 := n
    []
    [has2 and m2 > card2] & reject!2 -> has2 := false
    []
    [has2 and m2 <= card2 and tgt2 = 1] & (
      transfer.(2).(1).m2 -> card2 := card2 - m2
      [|{card2}|{|transfer|}|{card1}|]
      transfer.(2).(1)?x -> card1 := card1 + x
    ); has2 := false
  ); X)
```

Next, we distribute the guard and sequential composition over the parallelism using the following new laws.

Law guard-par-dist

```
[g]&(
  c->A
  [|{|c|}|]
  c-> B
```

```
)
[=
([g]&c->A)
[|{|c|}|]
c-> B
```

Law par-seq-dist

```
(c -> A [|ns1|{|c|}|ns2|] c-> B); C
[=
c -> (A; C) [|ns1|{|c|}|ns2|] c -> B

provided
  FV(B) inter FV(C) = {}
```

The resulting main action is as follows.

```
  @ mu X @ ((
    [not has1] & pay.1?j:(1 <> j)?n:(n > 0) ->
      has1 := true; src1 := 1; tgt1 := j; m1 := n
    []
    [has1 and m1 > card1] & reject!1 -> has1 := false
    []
    ([has1 and m1 <= card1 and tgt1 = 2] &
      transfer.(1).(2).m1 -> card1 := card1 - m1;
      has1 := false
    [|{card1}|{|transfer|}|{card2}|]
    transfer.(1).(2)?x -> card2 := card2 + x)
    []
    [not has2] & pay.2?j:(2 <> j)?n:(n > 0) ->
      has2 := true; src2 := 2; tgt2 := j; m2 := n
    []
    [has2 and m2 > card2] & reject!2 -> has2 := false
    []
    ([has2 and m2 <= card2 and tgt2 = 1] &
      transfer.(2).(1).m2 -> card2 := card2 - m2;
      has2 := false
    [|{card2}|{|transfer|}|{card1}|]
    transfer.(2).(1)?x -> card1 := card1 + x)
  ); X)
```

Next, we apply the following novel law to transform a recursion of external choices into a parallelism of recursions.

Law rec-ext-par-rec

```
mu X @ ((
        a1 -> A1 [] a2 -> A2
        []
        (b -> B1 [| wrtV(B1) | {|b|} | wrtV(B2) |] b -> B2)
        []
        c1 -> C1 [] c2 -> C2
); X)
[=
mu X @ ((a1 -> A1 [] c1 -> C1 [] b -> B1); X)
  [| wrtV(A1,B1,C1) | {b} | wrtV(A1,B1,C1) |]
mu X @ ((a2 -> A2 [] c2 -> C2 [] b -> B2); X)

provided
        (FV(A1) union FV(B1) union FV(C1))
        inter
        (FV(A2) union FV(B2) union FV(B2))
        =
        {}
```

The result action is as follows.

```
  @ (
    mu X @ ((
      [not has1] & pay.1?j:(1 <> j)?n:(n > 0) ->
        has1 := true; src1 := 1; tgt1 := j; m1 := n
      []
      [has1 and m1 > card1] & reject!1 -> has1 := false
      []
      [has1 and m1 <= card1 and tgt1 = 2] &
        transfer.(1).(2).m1 -> card1 := card1 - m1;
        has1 := false
      []
      transfer.(2).(1)?x -> card1 := card1 + x
    ); X)
    [|
      {card1,has1,src1,tgt1,m1}|
      {|transfer|}
      |{card2,has2,src2,tgt2,m2}
    |]
    mu X @ ((
      transfer.(1).(2)?x -> card2 := card2 + x
      []
      [not has2] & pay.2?j:(2 <> j)?n:(n > 0) ->
        has2 := true; src2 := 2; tgt2 := j; m2 := n
```

```
      []
      [has2 and m2 > card2] & reject!2 -> has2 := false
      []
      [has2 and m2 <= card2 and tgt2 = 1] &
        transfer.(2).(1).m2 -> card2 := card2 - m2;
        has2 := false
    ); X)
  )
```

Finally, we apply a process refinement law to split the process into the parallel composition of two other processes `Card1` and `Card2`.

```
process Card1 =
begin
  state
    card1: Money := 10
    has1: bool := false
    src1: Index
    tgt1: Index
    m1: Money

  @ mu X @ ((
    [not has1] & pay.1?j:(1 <> j)?n:(n > 0) ->
      has1 := true; src1 := 1; tgt1 := j; m1 := n
    []
    [has1 and m1 > card1] & reject!1 -> has1 := false
    []
    [has1 and m1 <= card1 and tgt1 = 2] &
      transfer.(1).(2).m1 -> card1 := card1 - m1;
      has1 := false
    []
    transfer.(2).(1)?x -> card1 := card1 + x
  ); X)
end

process Card2 =
begin
  state
    card2: Money := 10
    has2: bool := false
    src2: Index
    tgt2: Index
    m2: Money
```

```
  @ mu X @ ((
    transfer.(1).(2)?x -> card2 := card2 + x
    []
    [not has2] & pay.2?j:(2 <> j)?n:(n > 0) ->
      has2 := true; src2 := 2; tgt2 := j; m2 := n
    []
    [has2 and m2 > card2] & reject!2 -> has2 := false
    []
    [has2 and m2 <= card2 and tgt2 = 1] &
      transfer.(2).(1).m2 -> card2 := card2 - m2;
      has2 := false
  ); X)
end

process CSpec = Card1  [|{|transfer|}|] Card2
```

The final process `CSpec` is a distributed process that refines the original specification. Further refinement steps can be applied to rename the state components of the processes `Card1` and `Card2` and replace them by a parametrised process `Card`.

Whilst this development was carried out for a small number of cards, the general strategy can be applied for other values of `N`. Furthermore, a similar strategy can be applied to refine a version of the model parametrised by the number of cards. This development however requires refinement laws that deal with iterated operators, as well as extensions to the language to support partitioning of sequences and mappings in parallel operator. Alternatively, it may be possible to verify the parametrised model using a strategy based on promotion as discussed in [Oli06]

We believe that the strategy used to verify this case study provides a general treatment of SoS contracts and their refinement into a concrete distributed SoS model. The overall approach is to first model the contract as a single process encapsulating a monolithic state over which we can assert global invariants that must be preserved in the SoS. This provides an *Olympian view* of the SoS topology, and due to its centralised state supports the use of traditional verification techniques such as simulation, theorem proving, model checking etc., and therefore can be subject to rigorous analysis, which would otherwise by difficult in a large distributed SoS.

A verified contract can then be refined into a collection of processes, which map onto the topology of the contract. As a consequence of refinement, properties verified for the abstract model also hold in the concrete one. Therefore, we also

provide an approach for the verification of emergent properties, which are specified as the invariants in the SoS contract.

4 Technical Overview

4.1 Functionality and workflow

The core of the refinement tool consists of an interface for pattern matching a fragment of the CML AST and execution of applicable laws to update the AST at this location. A refinement law is essentially a functional rewrite rule over a particular kind of AST node, possibly with meta-variables and side-conditions (provisos). The main control loop follows this sequence:

1. select a fragment of the AST where the user has selected or placed his cursor. The AST node selected is calculated as the least common ancestor for the nodes at the beginning and end of the selection. For example, in `Skip ; Stop ; Skip` if we select between the `t` in `Stop` and the `i` in the second `Skip` then we will select node `Stop ; Skip`;
2. check for applicability of all known refinement laws to this AST fragment (usually via a pattern match, but potentially programmatically);
3. display applicable laws in a list for the user to select;
4. when a user selects a law request any meta-variables required to complete the refinement step;
5. apply selected law to the AST fragment, updating the CML document tree and adding any necessary provisos to the list of refinement proof obligations. This step is performed via a text edit to the CML file.

Refinement provisos usually take the form of propositional formulae which must be discharged by a theorem prover for the refinement to be correct. Consider the following law for example:

`[pre p post q] [= Skip` **provided** `p => q`

Here the refinement proviso is $p \Rightarrow q$, which should be discharged by the CML theorem prover [FP13]. If it cannot be discharged, the refinement is unsubstantiated and future refinements are therefore invalid. Nevertheless, refinement without discharge can be useful for experimental and speculative refinement where a particular result is desired. Currently discharge of provisos is unsupported by the tool, though a link to the theorem prover exists in Symphony and we propose to

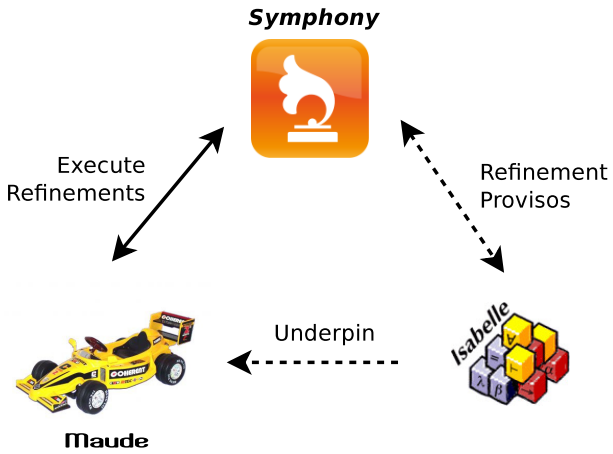

Figure 10: Overview of the interaction in the refinement tool

enable its integration in the future. Moreover, the potential exists for using other components of Symphony to discharge provisos, such as the COMPASS model checker.

Certain refinement laws are also parametric, meaning the right-hand side contains meta-variables not mentioned in the left-hand side. Therefore, in order to apply these laws it is necessary for the user to fill in values for the variables. This being the case, the tool will, upon application of the law, pop up a series of dialog boxes requesting values for the variables. Meta-variables can either have the type CML expression or CML action, and must correctly parse as such. Once all variables and have been entered and accepted, the refinement is executed with the variables substituted.

It should be noted that as a further step to our refinement tool, each of the refinement laws we have implemented should be separately proved as theorems within the Isabelle/UTP semantic framework [FZW14], within the context of the theory of CML, as illustrated in Figure 10 for Maude-based refinement. This will ensure that our refinements are truly mathematically and foundationally sound. Many of our current laws have previously been proved in the context of *Circus* either by hand or in ProofPower-Z [Oli06].

4.2 Programmatic refinement laws

Central to refinement API is the `IRefine` interface, which all refinement law classes must implement. A refinement law, in general, consists of the following

elements:

- a name string (`getName()`);
- an explanatory detail string (`getDetail()`);
- an applicability condition (`isApplicable(INode node)`), which checks if the refinement law can be applied to the given AST node;
- a list of required meta variable parameters (`getMetas()`);
- an application method (`apply(Map<String, INode> metas, INode node, int offset)`) which applies a refinement law, under a suitable map of meta-variable substitutions, to a given AST node with a specified textual offset (to represent any indentation of code). The method returns text which can be spliced into the CML file.

A programmatic refinement law simply consists of a Java class that implements this interface. Typically `isApplicable` will check if the passed AST nodes are of a particular class using `instanceof`, and `apply` will manipulate the input AST node. Programmatic refinement laws are useful for when a complex imperative manipulation is needed, or access to data not contained in the AST fragment is required, such as external AST nodes or other resources. However they also require the most effort to implement as pattern matching and extraction must be written manually.

4.3 Maude refinement laws

4.3.1 Introduction to Maude

The Maude System [CDE+99, CDE+07] is an engine for executing specifications written using rewrite logic. A user typically specifies a collection of sorts, term constructors, equations for these terms, and finally rewrite laws which specify how terms of a particular form can be rewritten to other terms. Maude can then be used to explore possible rewrites a system can undertake, which could for instance specify the behaviour of an abstract machine. Due to Maude's powerful search facilities for possible rewrites, we considered that it would make an excellent platform for implementing refinement laws, inspired by similar work with the rCOS language [GLMW13].

We briefly introduce the Maude language with some examples. A Maude file consists of a collection of *modules*, each of which contains a number of definitions. There are several types of modules, but the two we focus on are called *functional*

modules and *system modules*. The former contain functional specifications, including sorts, term constructors, variables, and equations, whilst the latter also contains rewrite rules. A simple functional module for natural number arithmetic follows.

```
fmod NAT-ARITH is
  sort Nat .
  op 0   : -> Nat [ctor] .
  op S   : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat [prec 20] .

  vars x y : Nat .

  eq 0 + x = x .
  eq S(x) + y = S(x + y) .
endfm
```

We declare the module with name `NAT-ARITH`, which contains a sort for natural numbers, and three operators: `0`, `S`, and `+`. The former two, which construct natural numbers, are declared as constructor operations by use of the `ctor` directive. The third operator, an infix plus operator, represented a function on two naturals. We specify the reduction behaviour for + by means of two equations, which use two variables `x` and `y`, which we declare to have type `Nat`. The standard definition for natural number addition then follows. We can then use Maude to perform reductions using our equational laws using the `red` command:

```
Maude> red S(S(0)) + S(0) .
```

... and Maude responds thus ...

```
reduce in NAT-ARITH : S(S(0)) + S(0) .
rewrites: 3
result Nat: S(S(S(0)))
```

Three applications of the equational laws show that 2 + 1 = 3. System modules are similar to functional modules, but in addition can also specify rewrite laws. For example:

```
mod CLIMATE is
  sort weathercondition .
  op sunnyday : -> weathercondition .
  op rainyday : -> weathercondition .
  rl [raincloud] : sunnyday => rainyday .
endm
```

This simple specification declares a sort, `weathercondition`, and two possible states, `sunnyday` and `rainyday`. We then declare a rewrite law, with the name `raincloud`, which transforms a `sunnyday` to a `rainyday`. Maude can then be used to apply this law, which it does by pattern matching against the left-hand side and then rewriting. Rewrite laws in Maude can also be conditional, depending on a true valuation for a given Boolean formula written over the variables in the rule left-hand side. A more complex example follows, representing a simple vending machine. The vending machine produces tea for one coin, and coffee for two coins.

```
mod VENDING is
  pr NAT-ARITH .

  sorts Button Hatch State .

  op Unpress   : -> Button .
  op Coffee    : -> Button .
  op Tea       : -> Button .

  op Nothing   : -> Hatch .
  op CupCoffee : -> Hatch .
  op CupTea    : -> Hatch .

  vars H : Hatch .
  vars B : Button .
  vars x y : Nat .

  op V[_,_,_,_]  : Nat Nat Button Hatch -> State .

  rl V[S(x), 0, Coffee, Nothing]
    => V[x, S(0), Coffee, Nothing] .
  rl V[S(x), S(0), Coffee, Nothing]
    => V[x,S(S(0)), Coffee, Nothing] .
  rl V[x, S(S(0)), Coffee, Nothing]
    => V[x, 0, Unpress, CupCoffee] .

  rl V[S(x), 0, Tea, Nothing]
    => V[x, S(0), Tea, Nothing] .
  rl V[x, S(0), Tea, Nothing]
    => V[x, 0, Unpress, CupTea ] .
```

endm

The module VENDING imports the previous functional module NAT-ARITH. It declares three sorts to represent the the button, hatch, and state of the vending machine. The button can be in three states, unpressed, pressed coffee, and pressed tea. The hatch can contain either nothing, a cup of coffee, or a cup of tea. We declare some variables for use in our rewrite laws. Then we create a constructor for the state of the vending which includes a natural number to represent the coins inserted, a natural number for the coins accepted, the button pressed, and the contents of the hatch.

We then create five rewrite laws to specify the behaviour of the vending machine. For coffee we have two laws which transfer two coins from inserted to accepted state, and for tea one. The remaining two laws dispense coffee or tea once enough money has been inserted. We can then used Maude to explore possible behaviours. We first exemplify the rew command, which recursively applies all possible rewrite laws until termination.

```
Maude> rew V[S(0), 0, Tea, Nothing] .
```

Here we ask Maude to fully rewrite the vending machine where we have inserted one coin and requested tea. The rew command is also aware of any equations we have specified and will also rewrite the terms appropriately using them. After entering this command, Maude responds:

```
rewrite in VENDING : V[S(0),0,Tea,Nothing] .
rewrites: 2
result State: V[0,0,Unpress,CupTea]
```

Two rewrite laws have been applied, and the final state is that no coins remain in the system and a cup of tea has been dispensed. If we insert an extra coin we get a similar result:

```
Maude> rew V[S(S(0)), 0, Tea, Nothing] .
rewrite in VENDING : V[S(S(0)),0,Tea,Nothing] .
rewrites: 2
result State: V[S(0),0,Unpress,CupTea]
```

However, at the end a coin remains in the system. In addition to the rewrite command, Maude includes a lower level search command which can be used to search for applicable laws.

```
Maude> search V[S(0),0,Tea,Nothing] =>1 V[x,y,B,H] .
search in VENDING : V[S(0),0,Tea,Nothing]
  =>1 V[x,y,B,H] .
```

```
Solution 1 (state 1)
states: 2  rewrites: 1
x --> 0
y --> S(0)
B --> Tea
H --> Nothing
```

We here request that Maude give us all possible *single* step rewrites (indicated by the `=>1`) into a state matching the pattern `V[x,y,B,H]`. Maude responds stating that there is one solution, and gives values for the variables in our result pattern. In this case a single coin has been accepted. Alternatively we can request the reflexive transitive closure of the rewrite relation:

```
Maude> search V[S(S(0)), 0, Tea, Nothing]
  =>* V[x,y,B,H] .
search in VENDING : V[S(S(0)),0,Tea,Nothing]
  =>* V[x,y,B,H] .

Solution 1 (state 0)
states: 1  rewrites: 0
x --> S(S(0))
y --> 0
B --> Tea
H --> Nothing

Solution 2 (state 1)
states: 2  rewrites: 1
x --> S(0)
y --> S(0)
B --> Tea
H --> Nothing

Solution 3 (state 2)
states: 3  rewrites: 2
x --> S(0)
y --> 0
B --> Unpress
H --> CupTea

No more solutions.
states: 3  rewrites: 2
```

There are three possibilities, each corresponding to one of the states of vending machine operation. In the first solution, nothing has as yet occurred, in the second one coin has been accepted and so on. It is this search functionality that we make most use of in our refinement tool for finding applicable laws. For more background to Maude, please see the manual[2].

4.3.2 Refinement in Maude

A partial CML AST was implemented in Maude during the COMPASS project by Prof. Musab AlTurki from KPUFM in Saudi Arabia[3]. This mimics, as much as possible, the syntax of CML as given by the Symphony tool so that applied refinements do not need much conversion. We have extended this AST with further constructs from the CML language and implemented a number of syntax level functions, such as the free variable function `FV`, the written variables function `wrtV`, and used channel function `usedC`, both of which are required by several refinement laws for provisos. For more information about side-conditions in refinement laws, please refer to [CSW03, Oli06]. We have also implemented some basic congruence laws for CML expressions and actions using equations, which means Maude can also prove simple theorems during refinement, and therefore discharge provisos.

Refinement laws are encoded in Maude using a pair of rewrite rules:

- the precondition rule, which tests whether a given refinement law is applicable and if so returns information about the law;
- the refinement rule, which encodes the step of the refinement.

In Maude these are encoded using rules of the following form, respectively:

```
refs[A] => rinf[NM, DS, LD, INP]  .
refine[NM, < A | M | p >] => < A' | M' | p' > .
```

Where `A`, `A'` are CML AST nodes; `NM`, `DS`, and `LD` are information strings encapsulating the law name, description, and content; `INP` is a set of possible input variables; `M`, `M'` are meta-variable maps; `p`, `p'` are proviso expressions. The control loop for Maude refinement then is:

1. Take the selected AST node and feed it through the Maude pretty printer visitor, which converts it to the format expected by Maude;

[2] `http://maude.cs.uiuc.edu/primer/maude-primer.pdf`
[3] Personal website: `http://www.ccse.kfupm.edu.sa/~musab/`

2. search for applicable laws for this node using the precondition rules, which is then returned to Symphony for display in the refinement law list:
3. once a law is applied, apply the corresponding refinement law in Maude to get the resultant AST node. The returned AST mode goes through a transformation to remove Maude specific encoding information.

To exemplify refinement laws in Maude, we illustrate the encoding of the following law combination of guards:

```
[g1] & [g2] & A [= [g1 and g2] & A
```

This law simply states that two guards can be composed to produce a single guard through conjunction. In Maude we implement this using the following two rewrite laws:

```
rl [pre-guard-combination] :
  refs[[g1] & #paren([g2] & A)] =>
         rinf[ "guard-combination"
             , "Guard combination"
             , "[g1] & [g2] & A [= [g1 and g2] & A"
             , ] .

rl [guard-combination] :
  refine[ "guard-combination"
        , < [g1] & #paren([g2] & A) | M | p >]
  => < [g1 and g2] & A | M | p > .
```

The first law is used to encode that the refinement is applicable for nodes of the form `[g1] & ([g2] & A)`. Specifically, when the refinement tool searches for applicable laws for a given AST node `A` it will ask for possible rewrites for `refs[A]` from Maude using the `search` command. The list returned is then all possible refinements steps. Application of `refs` returns a quadruple, consisting of the law code, law name, a textual description of the law, and a set of meta-variables which are required for the law to be applied.

The second law defines the actual rewrite the refinement law performs. Once law with code `l` has been chosen, the tool will send to Maude a request to rewrite `refine[l, < A | M | p >]`, where `M` is a map from meta-variable names to values (currently either an action or expression), and `p` is a set of provisos. Maude will then construct a new AST fragment, using the meta-variables when necessary, and a set of additional provisos required to support the law. In the case of our example law, no meta-variables are required and no provisos, so the rewrite law simply performs the node update.

To illustrate these laws in action, we exemplify the process in Maude. The user has requested to refine the AST node "`[x >= 0] & [x <= 5] & Skip`". The first step is to search for applicable laws using the Maude search command.

```
Maude> search refs[[#nm("x") >= #n(0)] &
                   #paren([#nm("x") <= #n(5)] & Skip)]
               =>* rinf[NM, DS, LD, INP ]  .
Solution 1 (state 1)
states: 2  rewrites: 1
NM --> "guard-combination"
DS --> "Guard combination"
LD --> "[g1] & [g2] & A [= [g1 and g2] & A"
INP -->

Solution 2 (state 2)
states: 3  rewrites: 2
NM --> "guard-weaken"
DS --> "Guard Weakening"
LD --> "[g1] & A [= [g2] & A provided g2 => g1"
INP --> #meta("g2", "expression")

No more solutions.
states: 3  rewrites: 2
```

We encode the AST node in Maude using the visitor, and then search for possible rewrites which give refinement information. In this case we get two applicable laws, the first is the application of the guard combination law, and the second the guard weakening law (which we shortly exemplify). The information output is then sent back to Symphony as a text block, and assuming the first law is picked we execute it using the `refine` command:

```
search in ACTION-REFINE :
    refine["guard-combination",< [#nm("x") >= #n(0)]&
    #paren([#nm("x") <= #n(5)]& Skip) | M | #b(true) >]
    =>* < A | M | p > .

Solution 1 (state 1)
states: 2  rewrites: 1
A --> [#nm("x") <= #n(5) and #nm("x") >= #n(0)]& Skip
M --> M
p --> #b(true)
```

```
No more solutions.
```

We specify the refinement law code to be executed (`guard-combination`) and the AST node to refine, and the result consists of the new node, the meta-variable map (here irrelevant), and the proviso of the law, which in this case is just true.

A slightly more complicated law encoding follows for the guard weakening law:

```
rl [pre-guard-weaken] :
  refs[ [g1] & A ] =>
    rinf[ "guard-weaken"
        , "Guard Weakening"
        , "[g1] & A [= [g2] & A provided g2 => g1"
        , #meta("g2", "expression")] .

crl [guard-weaken] :
  refine["guard-weaken", < [g1] & A | M | p >] =>
  < [getExp(M["g2"])] & A
  | empty
  | g1 => getExp(M["g2"]) >
  if M["g2"] =/= undefined and isExp(M["g2"]) .
```

In this example the precondition states that a single meta-variable, `g2`, is required of type "expression". The actual refinement law is a *conditional* rewrite law (indicated by **crl** instead of **rl**), since it relies on this variable being defined and having the correct type. The law body uses map lookup, written `M[k]`, to get the value of the meta-variable `g2` and then extracts the expression within. This law also has the proviso that $g1 \Rightarrow g2$, and this is also encoded. The condition of the law (stated after the **if**) states that there must exist a value for `g2` in `M` (it cannot be undefined), and further that this value must be an expression.

Of the two kinds of law, Maude refinement laws are the easiest to implement, once the refinement law DSL encoded above is understood. They are not as expressive as programmatic laws, the latter being computationally complete, and currently cannot make use of other external AST nodes. Nevertheless, for the majority standard refinement laws the Maude approach is best. Thus far we have only implemented a small number of laws ($\sim$ 20) but along with the programmatic refinement laws this is sufficient to support the example in Section 3.1.

5 Related Work

Various tools have been previously developed to support formal refinement. The following descriptions are based on [CHN+94] and [ZOC12].

RED [Vic90] is a refinement tool that supports the application of Morgan's refinement calculus, but does not support the verification of proof obligations. RED is no longer available.

The tool described in [Nic93] and [GNU92] supports Morgan's refinement calculus, and provides some support for discharging proof obligations. This tool is also no longer available.

Centipede [BHS92] supports the refinement of specifications and guarded commands extended with features of action systems. Verification of proof obligations is delegated to the theorem prover HOL. This tool is also no longer available.

The tool described in [CR88] supports only a limited procedural language and is also no longer available.

A number of HOL based refinement tools have been developed by [BW90] and [Gru92]. [BW90] supports Back's refinement calculus. Neither of these tools seem to be available.

To the best of our knowledge, the only refinement tools (for calculation) currently available are Refine [OXC04] and the related tool CRefine [OGdC08]. While Refine supports Morgan's refinement calculus, CRefine supports refinement of *Circus* specifications. They only support partial verification of proof obligations and do not seem to have support for user defined refinement laws.

Finally, the *Circus* language and the refinement tactic language ArcAngel have been mechanised in the theorem prover ProofPower-Z, thus providing support for mechanised refinement. While new rules can be added and the tools support discharging proof obligations, there is currently no specific user interface for doing this.

6 Conclusions

We have presented our implementation of the refinement tool for CML. This tool allows an abstract model to be transformed into a concrete, possibly executable,

model through application of a collection of refinement laws. We have implemented an interface for searching for, and then applying suitable laws to, nodes of the CML AST. Refinement laws can be specified either programmatically in Java, or else using a simple DSL we have implemented in the Maude rewrite system. We have also applied this tool to a simple example, and believe in the future this tool could be applied both to refinement of constituent system models. Moreover, through implementation of further refinement laws for CML it can also be applied to proving how a System of Systems satisfies its contract, as demonstrated in Section 3.2.

Nevertheless, the tool currently faces a number of limitations which we leave for future work. The refinement provisos can currently not be discharged, though a link to the theorem prover plugin exists which should enable this. However, complex refinement provisos including further refinements will require a complete theory of CML mechanised in Isabelle/UTP which is still a work in progress. In a related note, the refinement laws implemented in the tool should also be mechanised in Isabelle/UTP to ensure that the refinements produced by our tool are correct and well-founded. The number of laws implemented is also limited, and more should be implemented in the future to enable more complex refinements. Indeed it is likely necessary that a complete set of laws specifically for SoS refinement is required to enable a more comprehensive approach to verification against a CML contract.

The refinement laws we have implemented are adapted from the previous laws for *Circus* [CSW03], which have been separately mechanised in ProofPower-Z [Oli06]. However these laws have not as yet been separately validated within the context of our own CML mechanisation based in Isabelle/UTP [FZW14, FP13]. Since the semantic model of CML does substantially differ from that of *Circus*, this is a necessary step to gain complete confidence in refinements. Being a deep embedding, Isabelle/UTP gives us the basis to precisely specify and prove these laws correct with respect to the underlying theory of CML. Once proven these laws could then be used to further certify refinements produced by our tool through a process of reconstruction, where each law application is substituted for an Isabelle theorem, the proven provisos acting as inputs where necessary. This then will allow us to fully follow the approach of [GLMW13] where Isabelle and Maude work in tandem to provide fully verified, and yet highly automated model refinements.

In terms of our Maude integration, the integration is stable but lacking support for a number of things. The CML AST representation is only partially complete, and in particular there is little support for indexed operations. Moreover we currently do not generate and maintain an AST representation of a particular CML file in

Maude. This is necessary if we wish to make use of contextual information in refinements. We will also need some form of edit model for the AST, so that the underlying Maude file can be separately maintained. Our use of Maude in the refinement tool is currently limited to the use of Maude's unification algorithm and basic search facilities. A desirable future feature would be automation of multiple refinement steps that can be implemented through Maude's powerful search facilities.

The interface we currently have is quite simple. In particular there is currently no management of the refinement state, means for instance it is currently impossible to reverse refinement steps properly. We also need to be able to save the refinement state so that it can be continued, and rolled back later if necessary. Also we do not currently have integration with other tools, for example the theorem prover and the model checker, the latter of which would be used to discharge certain kinds of action refinements.

References

[BHS92] R. J. R. Back, J. Hekanaho, and K. Sere. Centipede-a program refinement environment. 1992.

[BW90] R.J.R. Back and J. Wright. Refinement concepts formalised in higher order logic. *Formal Aspects of Computing*, 2(1):247–272, 1990.

[BW98] Ralph-Johan Back and Joakim Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.

[CDE+99] M. Clavel, F. Durn, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and J. F. Quesada. The maude system. In *Rewriting Techniques and Applications*. Springer, LNCS1631, 1999.

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes of Computer Science*. Springer-Verlag, 2007.

[CHN+94] David Carrington, Ian Hayes, Ray Nickson, Geoffrey Watson, and Jim Welsh. A review of existing refinement tools. Technical Report 94-8, Software Verification Research Centre, Department of Computer Science, The University of Queensland, 1994.

[CR88] D. A. Carrindgton and K. A. Robinson. A prototype program refinement editor. In *Australian Software Engineering Conference*, pages 45–63. ACS, 1988.

[CSW03] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2–3):146–181, 2003.

[FP13] Simon Foster and Richard J. Payne. Theorem proving support - developers manual. Technical report, COMPASS Deliverable, D33.2b, September 2013.

[FZW14] S. Foster, F. Zeyda, and J. Woodcock. Isabelle/UTP: A mechanised theory engineering framework. In *5th Intl. Symposium on Unifying Theories of Programming*, 2014.

[GLMW13] Andreas Griesmayer, Zhiming Liu, Charles Morisset, and Shuling Wang. A framework for automated and certified refinement steps. *Innovations in Systems and Software Engineering*, 9(1):3–16, 2013.

[GNU92] Lindsay Groves, Raymond Nickson, and Mark Utting. A tactic driven refinement tool. In *5th Refinement Workshop*, pages 272–297. Springer, 1992.

[Gru92] Jim Grundy. A window inference tool for refinement. In CliffB. Jones, RogerC. Shaw, and Tim Denvir, editors, *5th Refinement Workshop*, Workshops in Computing, pages 230–254. Springer London, 1992.

[Mor90] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, London, UK, 1990.

[Nic93] R. Nickson. *Tool Support for the Refinement Calculus*. PhD thesis, Victoria University of Wellington, 1993.

[OGdC08] M. V. M. Oliveira, A. C. Gurgel, and C. G. de Castro. CRefine: Support for the *Circus* Refinement Calculus. In Antonio Cerone and Stefan Gruner, editors, *6th IEEE International Conferences on Software Engineering and Formal Methods*, pages 281–290. IEEE Computer Society Press, 2008. IEEE Computer Society Press.

[Oli06] M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, Department of Computer Science - University of York, UK, 2006. YCST-2006-02.

[OXC04] M. V. M. Oliveira, M. Xavier, and A. L. C. Cavalcanti. Refine and Gabriel: Support for Refinement and Tactics. In Jorge R. Cuellar and Zhiming Liu, editors, *2nd IEEE International Conference on Software Engineering and Formal Methods*, pages 310 – 319. IEEE Computer Society Press, September 2004. IEEE Computer Society Press.

[Vic90] Trevor Vickers. An overview of a refinement editor. In *Fifth Australian Software Engineering Conference 1990: Proceedings, The*, page 39. Institution of Radio and Electronic Engineers, Australia, 1990.

[WC01] J. C. P. Woodcock and A. L. C. Cavalcanti. A Concurrent Language for Refinement. In A. Butterfield and C. Pahl, editors, *IWFM'01: 5th Irish Workshop in Formal Methods*, BCS Electronic Workshops in Computing, Dublin, Ireland, July 2001.

[WSC+08] Jim Woodcock, Susan Stepney, David Cooper, John A. Clark, and Jeremy Jacob. The Certification of the Mondex Electronic Purse to ITSEC Level E6. *Formal Aspects of Computing*, 20(1):5–19, 2008.

[ZOC12] Frank Zeyda, Marcel Oliveira, and Ana Cavalcanti. Mechanised support for sound refinement tactics. *Formal Aspects of Computing*, 24(1):127–160, 2012.