Project FET Profundis

Minimization Module: User Manual
Version 0.1 — **DRAFT**

Gialuigi Ferrari Ugo Montanari Marco Pistore Roberto Raggi
Emilio Tuosto

Dipartimento di Informatica
Università di Pisa

June 9, 2002

Abstract

This document describes the application program interface relative to an implementation of a partition refinement algorithm. We show how the API is used in the implementation of the minimization algorithm. As a programming example we detail the realization of the standard automata case. Data format and a short programming guideline for HD-automata is detailed. More precisely, we consider HD-automata for the early semantics of π-calculus.

Contents

1 Overview **1**

2 API **2**
2.1 State . . . 2
type t . . . 3
id: *state*:t → string . . . 3
compare: *st1*:t → *st2*:t → int . . . 3
print: *state*:t → unit . . . 3
2.2 Arrow . . . 3
type statetype . . . 3
type labeltype . . . 3
type t . . . 3
source: *arw*:t → statetype . . . 3
target: *arw*:t → statetype . . . 4
label: *arw*:t → labeltype . . . 4
compare: *arw1*:t → *arw2*:t → int . . . 4
compose: *arw1*:t → *arw2*:t → t . . . 4
print: *arw*:t → unit . . . 4

2.3 Bundle . . . 4
type arrowtype . . . 4
type quadtype . . . 4
type statetype . . . 5
from_arrow_list: *arrows*:(arrowtype list) → quadtype list . . . 5
to_arrow_list: *state*:statetype → *bundle*:(quadtype list) → arrowtype list . . . 5
normalize: *red*:(quadtype list → quadtype list) → *bundle*:(quadtype list) → quadtype list . . . 5
minimize: *red*:(quadtype list → quadtype list) → *bundle*:(quadtype list) → quadtype list . . . 5
diff: *bl1*:(quadtype list) → *bl2*:(quadtype list) → quadtype list . . . 5
compare: *bl1*:(quadtype list) → *bl2*:(quadtype list) → int . . . 5
print: *bundle*:(quadtype list) → unit . . . 5
2.4 Automaton . . . 6
type statetype . . . 6
type arrowtype . . . 6
type bundletype . . . 6
type t . . . 6
create: *start*:statetype → *states*:statetype list → *arrows*:arrowtype list → t . . . 6
start: *a*:t → statetype . . . 6
states: *a*:t → statetype list . . . 6
arrows: *a*:t → arrowtype list . . . 6
bundle: *a*:t → *state*:statetype → bundletype . . . 6
print: *a*:t → unit . . . 6
2.5 Domination . . . 6
type quadtype . . . 7
dominated: *qd*:quadtype → *bundle*:(quadtype list) → quadtype option . . . 7
2.6 Bisimulation . . . 7
type bundletype . . . 7
type resulttype . . . 7
bisimilar: *bl1*:bundletype → *bl2*:bundletype → resulttype option . . . 7
2.7 Block . . . 7
type statetype . . . 7
type bundletype . . . 7
type buckettype . . . 7
type resulttype . . . 8
type t . . . 8
id: *block*:t → string . . . 8
states: *block*:t → statetype list . . . 8
cardinal: *block*:t → int . . . 8
norm: *block*:t → bundletype . . . 8
mem: *state*:statetype → *block*:t → bool . . . 8
from_states: *states*:(statetype list) → t . . . 8
to_state: *id*:int → *block*:t → statetype . . . 8
close_block: *env*:(statetype → t) → *buck*:buckettype → t . . . 8
next: *env*:(statetype → t) → *h_n*:(t → statetype) → *bundle*:bundletype → bundletype . . . 8
split: *bundle*:bundletype → *pred*:(statetype → resulttype option) → *block*:t → buckettype ×t 9
compare: *blk1*:t → *blk2*:t → int . . . 9

3 Reducer **10**

4 The standard automata case **12**
4.1 The implementation . . . 12
4.2 How to use the reducer . . . 17

5 The HD-automata case **18**
5.1 The HD-automata file format 18

1 Overview

This document describes the application program interface of an implementation of a partition refinement algorithm. The implementation is written in OCAML. The main features of OCAML exploited in our realization are polimorphism and encapsulation. Polimorphism is one of the intrinsic peculiarity of ML-language family, while encapsulation may be obtained in OCAML in two different ways; the first way is by using the object oriented features of the language, the second way is provided by modular programming features. More precisely, the module system separates the definition of interface specification, called *signatures* (i.e. definition of *abstract data types*) from their realizations, called *structures*. A structure may be parameterized using OCAML *functors*. Where a functor maps structures of a given signature on structures of other signatures.

In our opinion, object oriented programming simply adds to polimorphism and encapsulation (features already present in functional programming) hierarchical relations among abstract data types. However, in our case, those relations are meaningless and, therefore, have not been exploited.

Our tool allows the user to specify the automata type and, after having implemented some functionalities on such data structures, a general minimization algorithm is applied and the minimal realization of the automaton is returned.

The algorithm implementation is detailed in Section 3. Module *Reducer* is the only structure module. *Reducer* depends on the other signatures and details the constraints among the types of these modules. Using a type-theoretic notation, we write such dependencies as follows.

$$Reducer : \prod \textit{State, Arrow, Bundle, Block, Automaton, Bisimulation, Domination } \langle ... \rangle,$$

where the constraints are specified with the following equalities:

- $Arrow.statetype = Bundle.statetype = Automaton.statetype = Block.statetype = State.t$
- $Bundle.arrowtype = Automaton.arrowtype = Arrow.t$
- $Automaton.bundletype = Bisimultion.bundletype = Block.bundletype =$ $Bundle.quadtype$ `list`
- $Block.resulttype = Bisimulation.resulttype$
- $Domination.quadtype = Bundle.quadtype$

The structure of API permits us two facilities.

1. it is possible to apply the minimization algorithm to different class of automata, e.g. standard automata or HD-automata. Indeed, the OCAML module parameterization is exploited: a module may depend on other defined modules. For instance, the implementation module *Reducer* depends on many others and its implementation defines the constraints between them.

2. if the model (calculus and behavioural equivalence) changes, the programmer must re-implement only part of the interface. For instance, in the implementation of HD-automata minimization the behavioural equivalence is the early bisimulation. If we want to apply the algorithm to another observational equivalence for the same family of calculi, we have to supply the implementation of *Domination* and *Bisimulation*.

In Section 2 the whole interface is described by detailing all the types and functions. All those interfaces must be implemented in order to apply the algorithm for the minimization.

Section 3 comments on the implementation of the minimization algorithm and the main function of the tool. Moreover, Section 5.1 describes the syntactic format of the input in the case of HD-automata.

2 API

This Section describes all the signatures defined for the reducer. For each signature, types and functions are described.

2.1 State

The module State defines the interface of automata states.

type **t**

is the type of the states.

***id*: *state*:t → string**

returns the identifier of the state *state*.

Obs It is assumed that state identifiers are uniquely defined.

***compare*: *st1*:t → *st2*:t → int**

compares the two states *st1* and *st2*. The result is

- 0, if the arrows are equal,
- -1, if *st1* is less than *st2*,
- 1, otherwise.

Obs The comparison is meant to be a structural comparison. However, any function that does not equate two conceptually different states may be adopted.

***print*: *state*:t → unit**

prints *state* on the standard output.

2.2 Arrow

The interface for arrow is given below. Only the minimal features of arrows are included. This module specifies a type *t* which is a dependent type. Indeed, it depends on the type of the states and the type of the labels.

type **statetype**

is the type of states.

See: State

type **labeltype**

is the type of labels.

Obs In this prototype the type of label is a generic type. Indeed, no interface is defined for labels. Probably in future it would be refined.

type **t**

is the type of the arrows. It is not specified and depends on *statetype* and *labeltype*.

***source*: *arw*:t → statetype**

returns the source of the arrow *arw*.

***target*: *arw*:t → statetype**

returns the target (destination) of the arrow *arw*.

***label*: *arw*:t → labeltype**

returns the label of the arrow *arw*.

***compare*: *arw1*:t → *arw2*:t → int**

compares arrows *arw1* and *arw2*. The result is

- 0, if the arrows are equal,
- -1, if *arw1* is less than *arw2*,
- 1, otherwise.

Obs The comparison is meant to be a structural comparison. However, any function that does not equate two conceptually different states may be adopted.

***compose*: *arw1*:t → *arw2*:t → t**

returns an arrow from *source arw1* to *target arw2*, *arw1* and *arw2* are arrows.

Obs Up to now *compose* is not used; it has been included in the interface, because in future extensions of HD-automata it could be useful, e.g. for specifying weak version of semantics.

***print*: *arw*:t → unit**

prints the arrow *arw* on the standard output.

2.3 Bundle

The module *Bundle* defines the interface for bundle types. A bundle contains the information about the observables and future states carried out by the transitions starting from a given state. The module relies on *statetype*, *arrowtype* and *quadtype*. Usually, a bundle is computed from a list of arrows.

type **arrowtype**

is the type of the arrow.

See: *from_arrow_list*, *to_arrow_list* and *Automaton.bundle*.

type **quadtype**

is the type of the elements in the bundle.

Obs the bundle's type is a list of *quadtype*.

type **statetype**

is the type of the states.

***from_arrow_list*: *arrows*:(arrowtype list) → quadtype list**

creates a bundle from the list of transitions *arrows*.

***to_arrow_list*: *state*:statetype → *bundle*:(quadtype list) → arrowtype list**

returns a list of arrows with source *state* from *bundle*. This functions is used by *Reducer* to compute the arrows of the minimal automaton.

See: *Reducer*, *Block.norm*.

***normalize*: *red*:(quadtype list → quadtype list) → *bundle*:(quadtype list) → quadtype list**

returns a normalized bundle from *bundle* and the reduce function *red*.

See: *Bisimultation.bisimilar*

***minimize*: *red*:(quadtype list → quadtype list) → *bundle*:(quadtype list) → quadtype list**

returns a minimized bundle from *bundle* and the reduce function *red*. Function *red* is supposed to eliminate dominated transitions from a given bundle (see Domination) We underline that the minimization is parameterized by the *compare* functions on states, arrows and quadruples. *Reducer* uses *minimize* to compute the representative bundle of a block.

See: *Bisimultation.bisimilar* and *Block*

***diff*: *bl1*:(quadtype list) → *bl2*:(quadtype list) → quadtype list**

returns the bundle obtained by *bl1* minus all quadruples in *bl2*.

***compare*: *bl1*:(quadtype list) → *bl2*:(quadtype list) → int**

compares the two bundles *bl1* and *bl2*. The result is

- 0, if the bundles are equal,
- -1, if *bl1* is less than *bl2*,
- 1, otherwise.

Obs The comparison is meant to be a structural comparison. However, any function that does not equate two conceptually different states may be adopted.

***print*: *bundle*:(quadtype list) → unit**

prints *bundle* on the standard output.

2.4 Automaton

The following module defines the interface for automata type. An automaton is built out from states and arrows between states. However, also the type of a bundle should be provided for specifying automata. The functions of an automaton allows to extract the relevant information.

type **statetype**

is the type of the states in the automaton.

type **arrowtype**

is the type of the arrows in the automaton.

type **bundletype**

is the type of the bundle (*all observable actions*).

type **t**

is the type for automata.

***create*: *start*:statetype → *states*:statetype list → *arrows*:arrowtype list → t**

creates an automaton from *start*, *states* and *arrows*.

***start*: *a*:t → statetype**

returns the start state of the automaton *a*.

***states*: *a*:t → statetype list**

returns the list of states of the automaton *a*.

***arrows*: *a*:t → arrowtype list**

returns the list of arrows of the automaton *a*.

***bundle*: *a*:t → *state*:statetype → bundletype**

returns the bundle of the state *state* in automaton *a*.

***print*: *a*:t → unit**

prints the automaton on the standard output.

2.5 Domination

Automata may contain transitions that are "redundant", in the sense that, a transition *t* represent a state change (with a given observation) that is semantically covered by another transition *t'*. We say that *t'* dominates *t*. The module *Domination* defines the interface for such dominance relation.

type **quadtype**

is the type of quadruples.

***dominated*: *qd*:quadtype → *bundle*:(quadtype list) → quadtype option**

returns *Some(qd')* if *qd'* is in *bundle* and dominates *qd*, otherwise, *None* is returned.

2.6 Bisimulation

The Bisimulation module specifies the interface for expressing the behavioural equivalence the user is interested in. Note that this module depends on the type adopted for bundles and is also parameterized with respect to the type of the result. The idea is that, in some cases it is not enough to know that the relation holds for two bundle but also auxiliary informations may be useful. For instance, in the case of standard automata, *resulttype* could be simply *bool*, but automata for name passing calculi also has names appearing on bundles and name correspondences could be used for computing the minimal automaton.

type **bundletype**

is the type of bundles.

type **resulttype**

is the type of the result of *bisimilar*.

***bisimilar*: *bl1*:bundletype → *bl2*:bundletype → resulttype option**

returns an optional type; if the relation holds between *bl1* and *bl2*, then *res* should be *Some(r)* (for some *r* of type *resultype*) otherwise *None* is returned.

2.7 Block

The module *Block* is the signature for blocks. A block is the data structure which contains the states that are considered equivalent at a given iteration. The main operation on a block is the split operation that divide a block into buckets, i.e. quasi-blocks that have some components that should be uniformely computed at the end of the splitting phase. *Reducer* will return a list of blocks as result of each iteration. Such block represents the states of the current approximation of the minimal automaton.

type **statetype**

is the type of states.

type **bundletype**

is the type of bundles.

type **buckettype**

is the type of buckets.

type **resulttype**

is the type used by the splitting operation to separate the states of a block into different equivalence classes.

type **t**

is the type of a block.

***id*: *block*:t → string**

returns the name of *block*.

Obs It is assumed that blocks have unique identifiers.

***states*: *block*:t → statetype list**

returns the list of states in *block*.

***cardinal*: *block*:t → int**

returns the cardinality of the set of states in *block*.

***norm*: *block*:t → bundletype**

returns the normalized bundle of *block*.

***mem*: *state*:statetype → *block*:t → bool**

returns *true* if, and only if, *state* is member of the set of states in *block*.

***from_states*: *states*:(statetype list) → t**

builds a block out of a list of states.

Todo rename from_states to init/initialize

***to_state*: *id*:int → *block*:t → statetype**

converts a *block* into a state. Integer *id* is used to uniquely generate the name of *block*.

***close_block*: *env*:(statetype → t) → *buck*:buckettype → t**

converts *buck* into a block. The conversion requires an environment *env* that associates to a state its containing block.

***next*: *env*:(statetype → t) → *h_n*:(t → statetype) → *bundle*:bundletype → bundletype**

returns the application of *h_n*, the *n*-th approximation of the functor (see *FMP02*) to *bundle*. As for *to_state* and environment *env* is required in order to substitute the destination states on *bundle* with the block that contains them.

***split*: *bundle*:bundletype → *pred*:(statetype → resulttype option) → *block*:t → buckettype ×t**

separates the states of *block* whose normalized bundle is to *bundle* equivalent, according to *pred*. Indeed, predicate *pred* returns *None* if such equivalence does not hold, otherwise it returns *Some(r)*, where *r* establishes the correspondence between the two bundles. The result is a pair bucket-block, where the first component is the bucket made of the equivalent states and the second component is the *block* where such states are removed.

***compare*: *blk1*:t → *blk2*:t → int**

compares blocks *blk1* and *blk2*. The result is

- 0, if the blocks are equal,
- -1, if *blk1* is less than *blk2*,
- 1, otherwise.

Obs The comparison is meant to be a structural comparison. However, any function that does not equate two conceptually different states may be adopted.

3 Reducer

This Section deals with the implementation of the partitioning algorithm. In particular, the main part of *Reducer*'s code are described.

```
let partitioning aut =
  let start, states, arrows = (Automaton.start aut),
                              (Automaton.states aut),
                              (Automaton.arrows aut) in
```

Initially, the list of *blocks* is made of a single block that contains all automaton's states.

```
    blocks := [ (Block.from_states states) ] ;
```

split blocks block returns a pair (*bucket*,*block'*) where *bucket* contains all the states supposed equivalent at the current iteration and *block'* is obtained by removing those states from *block*.

```
    let split blocks block =
      try
```

minimal computes the minimal bundle of the first state of *block*. Note that to compute the *minimal* and the *normalized* bundle we use three auxiliary functions *red*, *env* and *h_n*. *red* is a filter function, e.g., for a given bundle *b* it returns the bundle obtained by removing from *b* all dominated quadruples (see *Domination.dominated*). *env* maps states to blocks; in particular given a state *q* returns the block that approximate *q* *h_n* maps blocks to states; given a block *b* returns the states *q* that represent *b* in the *n*-th approximation.

```
        let minimal =
          (Bundle.minimize red
             (Block.next
                (env blocks)
                (h_n blocks)
                (Automaton.bundle aut (List.hd (Block.states block))))) in
```

At this point, *block* is splitted in the pair (bucket, block'). More precisely, the function *Block.split* is invoked with a predicate that, for each state *q*, computes its normalized bundle *normal* and returns (*Bisimulation.bisimilar minimal normal*).

```
          Some (Block.split
                  minimal
                  (fun q ->
                     let normal =
                       (Bundle.normalize
                          red
                          (Block.next (env blocks)
                             (h_n blocks)
                             (Automaton.bundle aut q))) in
                       Bisimulation.bisimilar minimal normal)
                  block)
      with Failure e -> None in
```

split_iter f blks, using the split function *f*, recursively splits the blocks in the list *blks* into a list of buckets. Such splitting is performed as much as possible.

```
let rec split_iter f = function
  | [] → []
  | e :: els →
      match f e with
        | Some(bucket, continuation) →
            if (Block.states continuation) = []
            then bucket :: (split_iter f els)
            else bucket :: (split_iter f (continuation :: els))
        | _ → (split_iter f els) in

let stop = ref false in

  while ¬ ( !stop ) do
    begin
```

oldblocks records the blocks of the previous iteration.

```
      let oldblocks = !blocks in

      let buckets = split_iter (split oldblocks) oldblocks in
        begin
```

The buckets computed by splitting all the blocks are coerced to real blocks. Such coercion is performed by adding to buckets the new information obtained in the current iteration.

```
          blocks := (List.map (Block.close_block (env oldblocks)) buckets);
```

The termination condition is evaluated. The termination is reached when the current list of blocks *blocks* is isomorphic to the list of blocks of the previous iteration.

Note that if each block is not broken, then i-th block of the current approximation (*blocks*) exactly corresponds to the i-th block of previous approximation (*oldblocks*). Therefore, the comparison between *blocks* and *oldblocks* can be done position-wise.

```
          stop :=
              (List.length !blocks) = (List.length oldblocks) ∧
              (List.for_all2
                  (fun x y → (Block.compare x y) ≡ 0)
                  !blocks
                  oldblocks)

        end
    end
  done ;
  !blocks
end
```

4 The standard automata case

4.1 The implementation

In this Section we describe a simple implementation of the signatures for (ordinary) *Automaton*.

First states of automata must be implemented.

```
module State =
struct
```

the only information that we need to represent a state is its identifier.

```
  type t = State of string

  let id = function State(x) → x
  let create x = State(x)
  let compare = compare
  let print = function State(x) → Printf.fprintf stdout "State: %s\n" x
end
```

1. module *Arrow* =

Arrows are defined in this Section. We use OCAML functor (or parameterizers) to make the implementation independent from states.

The type *Arrow* depends on the type *State*

```
  functor(State : StateSig) →
struct

  type statetype = State.t
```

labeltype represents the observables associated with arrows.

```
  type labeltype = string
```

an *arrow* is described by a tuple (source, label, target)

```
  type t = Arrow of statetype × labeltype × statetype
```

this code provides all functions needed to accomplish with the *Reducer.Arrow* signature

```
  let create s l t = Arrow(s, l, t)
  let source = function Arrow(s, l, t) → s
  let target = function Arrow(s, l, t) → t
  let label = function Arrow(s, l, t) → l

  let compose ar1 ar2 =
    match ar1, ar2 with
      | Arrow(s1, l1, t1), Arrow(s2, l2, t2) →
          if (State.compare t1 s2) ≡ 0
          then Arrow(s1, l1 ^ l2, t2)
          else failwith "Error: not composable arrows"

  let compare = compare

  let print = function Arrow(s, l, t) →
    Printf.fprintf stdout "Arrow = " ;
```

```
    print_string "    ";
    State.print s ;
    print_string "    ";
    Printf.fprintf stdout "label = %s \n" l ;
    print_string "    ";
    State.print t ;
end
```

2. module *Bundle* =

Bundle depends on the types *State* and *Arrow*.

Note that *StateSig* is a subsignature of *State*, and *ArrowSig* is a subsignature of *Arrow*.

```
  functor (State : StateSig) →
  functor (Arrow : ArrowSig with type statetype = State.t) →
struct

  type statetype = State.t
  type arrowtype = Arrow.t
```

In the case of *Automaton* implementation the elements of *bundle* are the *arrows* of the *automaton*. Note that this simplifies the functions *from_arrow_list* and *to_arrow_list* (they are simply the identity functions) but complicates *compare*, because we must ignore the source of the arrows.

```
  type quadtype = arrowtype
  type t = quadtype list

  let from_arrow_list = function x → x
  let to_arrow_list q = function x → x

  let compare bl1 bl2 =
    let xx = State.create "dummy" in

    let bl1' =
      List.sort Arrow.compare
        (List.map
           (fun ar →
              Arrow.create xx (Arrow.label ar) (Arrow.target ar))
           bl1) in

    let bl2' =
      List.sort Arrow.compare
        (List.map
           (fun ar →
              Arrow.create xx (Arrow.label ar) (Arrow.target ar))
           bl2) in

      compare bl1' bl2'
```

normalization and minimization leave the bundle unchanged.

```
  let normalize = fun red x → x
  let minimize = fun red x → x

  let diff = list_diff
```

```
  let print bundle = List.iter Arrow.print (bundle)
end
```

3. module *Automaton* =

Automaton depends to the type *State*, *Arrow* and *Bundle*

```
  functor(State : StateSig) →
  functor(Arrow : ArrowSig with type statetype = State.t) →
  functor(Bundle : BundleSig with type arrowtype = Arrow.t) →
struct
  type statetype = State.t
  type arrowtype = Arrow.t
  type bundletype = Bundle.t
```

Automaton are represented as tuples (start, states, arrows)

```
  type t = Automaton of statetype × statetype list × arrowtype list

  let create start states arrows =
    Automaton(start, states, arrows)
```

we provide the projections

```
  let states = function Automaton(start, states, arrows) → states
  let arrows = function Automaton(start, states, arrows) → arrows
  let start = function Automaton(start, states, arrows) → start
```

the function that returns the bundle for the given state

```
  let bundle =
    function Automaton(start, states, arrows) →
      fun (q : statetype) →
        (Bundle.from_arrow_list
           (List.filter
              (fun a → State.compare q (Arrow.source a) ≡ 0) arrows))
```

and the print function

```
  let print (a : t) =
    List.iter State.print (states a);
    print_newline();
    List.iter Arrow.print (arrows a)
end
```

4. module *Bisimulation* =

This module depends to the type of the *Bundle*

```
  functor (Bundle : BundleSig) →
struct
  type bundletype = Bundle.t
```

in this case the extra information returned by *bisimilar* has type boolean.

```
  type resulttype = bool
```

two states are bisimilar if they have the same bundle in the current approximation.

```
  let bisimilar bundle1 bundle2 =
    if (Bundle.compare bundle1 bundle2 ≡ 0)
    then Some true (* NOTE: the extra information is ignored *)
    else None
end
```

5. module *Domination* =

This module depends on signature *Bundle*

```
  functor (Bundle : BundleSig) →
struct
  type quadtype = Bundle.quadtype
  type bundletype = quadtype list
```

does not exist a quadruple qd' in *bundle* that dominates *qd*

```
  let dominated qd bundle =
    None
end
```

6. module *Block* =

```
  functor(State : StateSig) →
  functor(Arrow : ArrowSig
            with type statetype = State.t) →
  functor(Bundle : BundleSig
            with type statetype = Arrow.statetype) →
  functor(Automaton : AutomatonSig
            with type statetype = State.t
            and type arrowtype = Arrow.t
            and type bundletype = Bundle.t) →
  functor(Bisimulation : BisimulationSig
            with type bundletype = Bundle.t) →

struct
```

this module depends on the type of the *State*, *Arrow*, *Bundle*, *Automaton* and *Bisimulation*.

```
  type statetype = State.t
  type bundletype = Bundle.t
  type automatontype = Automaton.t
  type resulttype = Bisimulation.resulttype
```

blocks are defined as a tuple (identifier, states, norm)

```
  type t = Block of string × statetype list × bundletype
```

There is no difference between buckets and blocks, because all information in the block depends only from the previous approximation.

```
  type buckettype = t

  let close_block env bucket =
```

```
    bucket
```

The constructors are

```
let from_states states =
    Block("", states, (Bundle.from_arrow_list []))

let create id states norm =
    Block(id, states, norm)

let to_state n block =
    State.create ("b" ^ (string_of_int n))
```

While projections are detailed below

```
let id =
    function Block(name, states, norm) → name

let states =
    function Block(name, states, norm) → states

let norm =
    function Block(name, states, norm) → norm

let cardinal block =
    List.length (states block)

let mem state block =
    List.mem state (states block)
```

this function provides the composition of *Automaton* with the previous approximation.

```
let next env h_n bundle =
    Bundle.from_arrow_list
      (unique
         (List.map
            (fun ar →
               (Arrow.create
                  (Arrow.source ar)
                  (Arrow.label ar)
                  (h_n (env (Arrow.target ar))))
            bundle)))
```

we split the block using the *List.partition*

```
let split minimal pred block =

    let (states', states'') =
      (List.partition
         (fun x → (pred x) ≠ None)
         (states block)) in

      ( (create "" states' minimal), (* the bucket *)
        (create (id block) states'' (norm block))) (* the remaining block *)

let compare block1 block2 =
    Bundle.compare (norm block1) (norm block2)

end
```

4.2 How to use the reducer

This Section aims at describing how it is possible to use the API's introduced so far. In order to do that, we describe the running example of instantiating the interfaces in the case of automata minimization.

Once all signatures (see Section 2) have been implemented, a new *Reducer* can be instantiated. The implementation of the signatures proceeds similarly to the case of ordinary automata.

The first step is the importing of *Reducer* and of all the implementation modules. In our running example

```
open Reducer

open Automaton_state
open Automaton_arrow
open Automaton_bundle
open Automaton_block
open Automaton_bisimulation
open Automaton_domination
open Automaton_
```

Then the structure module must be instantiated is such a manner that module dependencies are satisfied:

```
module AutomatonState = State
module AutomatonArrow = Arrow (AutomatonState)
module AutomatonBundle = Bundle (AutomatonState) (AutomatonArrow)
module MyAutomaton = Automaton (AutomatonState) (AutomatonArrow) (AutomatonBundle)
module AutomatonDomination = Domination (AutomatonArrow) (AutomatonBundle)
module AutomatonBisimulation = Bisimulation (AutomatonBundle)

module AutomatonBlock = Block (AutomatonState) (AutomatonArrow)
                              (AutomatonBundle) (MyAutomaton)
                              (AutomatonBisimulation)

module AutomatonReducer = Reducer (AutomatonState) (AutomatonArrow)
                                  (AutomatonBundle) (MyAutomaton)
                                  (AutomatonBisimulation) (AutomatonBlock)
                                  (AutomatonDomination)
```

At this point, *AutomatonReducer.reduce* can be invoked for reducing automata, as shown below.

```
let automaton = ... in
let reduced_automaton = (AutomatonReducer.reduce automaton) in
    ...
```

5 The HD-automata case

5.1 The HD-automata file format

The format for I/O data of *HDReducer.reducer* is described in this Section. Basically, such format mimics the scheme of the type of automata described in Section 2. Roughly, an automaton is a triple made of an initial state, a set of states and a set of arrows between states.

HD-automata extend ordinary automata in two ways:

1. states are equipped with local names and group of symmetries (permutations) on names. Names are supposed to be totally ordered,

2. a transition $s \overset{l\ \pi,\ \sigma}{\rightarrow} d$ exposes names π of the source state s, and has a function σ that maps names of the destination state d into the name of s, or in a distinguished name $\star$.

In our data model names are represented as integers, $\star$ is represented as $*$ or as 0. Moreover, if a state has n names we represent them with the segment of integers $1, ...n$. Note that this is consistent because names have local meaning.

A symmetry over n names may be simply expressed by means of a list $\rho = [i_1; ...; i_n]$ of distinct integers, where each i_j is in $1, ..., n$; the convention is that ρ represents the permutation that maps each j in i_j. For instance, $[2; 1; 3]$ represents a permutation of 3 elements: in particular it is the permutation that exchanges 1 and 2, and leaves 3 unchanged. The permutation group is specified as a list of permutations. Such convention is also adopted for representing other functions on names, e.g. σ's.

Given the above assumptions, we describe the format by commenting on the following example:

```
    start q0

    state q0 3
    state q1 3
    state q2 2
    state q3 2    [ [1;2] ; [2;1] ]
    state q4 3

#   SOURCE      TARGET     PI_LABEL          SIGMA
      q0    ->    q1     out[ 1 ; 2 ]    [ 1 ; 2 ; 3 ]
      q0    ->    q1     out[ 2 ; 1 ]    [ 1 ; 2 ; 3 ]
      q0    ->    q0     tau             [ 2 ; 1 ; 3 ]

      q1    ->    q2     in[ 1 ; 1 ]       [ 1 ; 2 ]
      q1    ->    q3     in[ 1 ; 2 ]       [ 1 ; 2 ]
      q1    ->    q4     in[ 1 ; 3 ]     [ 1 ; 2 ; 3 ]
      q1    ->    q4     bin[ 1 ]        [ 1 ; 2 ; * ]

      q2    ->    q2     bout[ 1 ; 2 ]     [ 1 ; 2 ]
```

`start` denotes the initial state (the name of the state is a string)

```
    start q0
```

then the list of states is given. For each state it is mandatory to specify

- the name of the state
- the number of local names of the state. Indeed, note that a group of symmetries has been explicitly specified only for `q3`. For all other cases, it is assumed to be the group made of the identity permutation over the names of the state.

On the other hand, the list of permutations of the state may be optionally specified.

Finally, the list of arrows of the automaton is given (the line starting with `#` is a comment). Columns `SOURCE` and `TARGET` are (the names of) the initial and final states of transitions. Column `PI_LABEL` is one of the strings `out`, `in`, `tau`, `bin`, `bout` followed by the local names exposed in the transition. Column `SIGMA` represents the σ-component of the transition. Such a function is represented as a list of integers whose length is the number of names of the target, while the elements are integers ranging from 1 to the number of names of the source state moreover also 0 or '*" may appear in the list (see transition from `q1` to `q4`).