Automatic Refinement Checking for B*

Michael Leuschel[1,2] and Michael Butler[1]

[1] School of Electronics and Computer Science
University of Southampton
Highfield, Southampton, SO17 1BJ, UK
{mjb,mal}@ecs.soton.ac.uk
[2] Institut für Informatik, Heinrich-Heine Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
leuschel@cs.uni-duesseldorf.de

Abstract. While refinement is at the heart of the B Method so far no automatic refinement checker has been developed for it. In this paper we present a refinement checking algorithm and implementation for B. It is based on using an operational semantics of B, obtained in practice by the ProB animator. The refinement checker has been integrated into ProB toolset and we present various case studies and empirical results in the paper, showing the algorithm to be surprisingly effective. The algorithm checks that a refinement preserves the trace properties of a specification. We also compare our tool against the refinement checker FDR for CSP and discuss an extension for singleton failure refinement.
Keywords: B-Method, Tool Support, Refinement Checking, Model Checking, Animation, Logic Programming, Constraints.

1 Introduction

The B-method is a well-established theory and methodology for the rigorous development of computer systems and programs. B was originally devised by Abrial [1] and has been applied for a wide range of safety critical applications.

B is based on the notion of *abstract machine*. The variables of an abstract machine are typed using set theoretic constructs such as sets, relations and functions. Each machine has a certain number of operations that can update the variables of the machine, as well as an invariant specified using predicate logic.

Refinement is a key concept in the B-Method. It allows one to start from a high-level specification and then gradually refine it into an implementation, which can then be automatically be translated into executable code. While there is tool support for proving refinement via semi-automatic proof (within Atelier-B [22], the B-toolkit [17], and now also B4Free), there has been up to now no automatic refinement checker in the style of FDR [10] for CSP [13, 19]. Thus, especially the development of B refinements has been a labour intensive activity. Indeed, when a refinement does not hold it may take a while for a B user to

* This research is being carried out as part of the EU funded projects: IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems) and IST-2001-38059 ASAP (Advanced Specialization and Analysis for Pervasive Systems).

realise that the proof obligations cannot be proven, resulting in a lot of wasted effort. In this paper we wish to speed up B development time by providing an automatic refinement checker that can be used to locate errors before any formal refinement proof is attempted. In some cases the refinement checker can actually be used as an alternative to the prover,[1] but in general the method presented in this paper is complementary to the traditional B tools.

In this paper we formalise the notion of refinement checking and present an algorithm which is at the heart of an automatic refinement checker. This new refinement checker has been implemented and integrated within the ProB validation tool for the B method [14]. At the heart of ProB is a fully automatic animator implemented mainly in SICStus Prolog. The undecidability of animating B is overcome in ProB by restricting animation to finite sets and integer ranges, while efficiency is achieved by delaying the enumeration of variables as long as possible. ProB comprises various visualization facilities [16] to display the state space in a user-friendly way. ProB also contains a model checker [7] which tries to find a sequence of operations that, starting from an initial state, leads to a state which violates the invariant (or exhibits some other error, such as deadlocking, assertion violations, or abort conditions). To compute the set of reachable states of a B machine the model checker makes use of the same underlying interpreter as the animator. In fact, the ProB interpreter can be viewed as providing the operational semantics of a B machine. In this paper we will re-use the same ProB interpreter as the foundation of the refinement checker. In case refinement is violated, the refinement checker displays a sequence of operations that can be performed by the "refinement" machine but not by the specification.

2 Scheduler Example

In this section we present a small example of a specification and its refinement in B to help motivate the work. Later we will be more precise about the meaning of refinement and refinement checking. Familiarity with B notation is assumed in the remainder of the paper.

Figure 1 presents a B specification ($Scheduler0$) of a system for scheduling processes on a single resource. In this model, each process has a state which is either *idle*, *ready* to become active or *active* whereby it controls the resource. The current set of processes is modelled by the variable *proc* and the *pst* variable maps each current process to a state. There is a further invariant stating that there should be no more than one active process ($pst^{-1}[\{active\}]$, the image of $\{active\}$ under the inverse of *pst*, represents the set of active processes). $Scheduler0$ contains events for creating new processes, making a process ready, allowing a process to take control of the resource (*enter*) and allowing a process to relinquish control (*leave*). Each of these events is appropriately guarded by a

[1] Namely when all sets and integer ranges are already finite to start with and do not have to be reduced to make animation by ProB feasible.

WHEN clause[2]. In particular, the *enter* event is enabled for a process p when p is ready and no other process is active.

MACHINE $Scheduler0$
SETS
  $PROC$;
  $STATE = \{idle, ready, active\}$
VARIABLES $proc, pst$
INVARIANT
  $proc \in \mathbb{P}(PROC) \wedge$
  $pst \in proc \rightarrow STATE \wedge$
  $card(pst^{-1}[\{active\}]) \leq 1$

INITIALISATION $proc, pst := \{\}, \{\}$

OPERATIONS
$new(p : PROC) \hat{=}$
  WHEN
    $p \in PROC \setminus proc$
  THEN
    $pst(p) := idle \parallel$
    $proc := proc \cup \{p\}$
  END;

$ready(p : PROC) \hat{=}$
  WHEN
    $pst(p) = idle$
  THEN
    $pst(p) := ready$
  END;

$enter(p : PROC) \hat{=}$
  WHEN
    $pst(p) = ready \wedge$
    $pst^{-1}[\{active\}] = \{\}$
  THEN
    $pst(p) := active$
  END;

$leave(p : PROC) \hat{=}$
  WHEN
    $pst(p) = active$
  THEN
    $pst(p) := idle$
  END

Fig. 1. Scheduler specification

Figure 2 presents a B refinement of $Scheduler0$. In this refinement, instead of mapping each current process to a state, we have a pool of idle processes *idleset* and a queue of ready processes *readyq*. We also have a flag indicating whether or not there is a process currently active ($activef$). When $activef$ is true, the identity of the currently active process is stored in *activep*. The queue of ready processes means that processes will become active in the order in which they became ready[3]. Now the *enter* event is enabled for process p when p is the first element in the queue and there is no active process.

We expect that $Scheduler1$ is a valid refinement of $Schduler0$ since any sequence of operations in $Schduler0$ should also be possible in $Schduler1$. Refinement checking of $Schduler0$ against $Scheduler1$ with our tool for a maximum of three processes ($PROC = \{p1, p2, p3\}$) finds no counterexamples. If we were to weaken the guard of the refined *enter* event, removing the clause $activef = FALSE$, this weaker refinement would allow more than one process to take control of the single resource. In terms of operation sequences, it would allow sequences in the refinement in which, for example, $enter(p1)$ is followed

[2] WHEN is the the Event-B syntax for the SELECT clause of classical B.
[3] In the *ready* event, $readyq \leftarrow p$ represents appending of p to the end of *readyq*.

MACHINE $Scheduler1$
REFINES $Scheduler0$
VARIABLES
$proc, idleset, readyq, activep, activef$
INVARIANT
$idleset \in \mathbb{P}(PROC) \wedge$
$readyq \in seq(PROC) \wedge$
$activep \in PROC \wedge$
$activef \in BOOL$

INITIALISATION
$proc := \{\} \;||\; readyq := [\,] \;||$
$activep :\in PROC \;||$
$activef := FALSE \;||$
$idleset := \{\}$

OPERATIONS

$new(p : PROC) \mathrel{\hat{=}}$
WHEN
$p \in PROC \setminus proc$
THEN
$idleset := idleset \cup \{p\} \;\;||$
$proc := proc \cup \{p\}$
END;

$ready(p : PROC) \mathrel{\hat{=}}$
WHEN
$p \in idleset$
THEN
$readyq := readyq \leftarrow p \;||$
$idleset := idleset \setminus \{p\}$
END;

$enter(p : PROC) \mathrel{\hat{=}}$
WHEN
$readyq \neq [\,] \wedge$
$p = first(readyq) \wedge$
$activef = FALSE$
THEN
$activep := p \;||$
$readyq := tail(readyq) \;||$
$activef := TRUE$
END;

$leave(p : PROC) \mathrel{\hat{=}}$
WHEN
$activef = TRUE \wedge$
$p = activep$
THEN
$idleset := idleset \cup \{p\} \;||$
$activef := FALSE$
END

Fig. 2. Refinement of the scheduler

by $enter(p2)$ without $leave(p1)$ occurring in between. It would thus be an incorrect refinement. The following counterexample is generated by ProB for the incorrect refinement:

$$new(p1),\ new(p2),\ ready(p1),\ ready(p2),\ enter(p1),\ enter(p2)$$

In ProB this counterexample is represented in diagrammatic form (see Figure 4 in the appendix). This counterexample discovered by ProB is a trace allowed by the incorrect refinement that is not a trace of the specification $Scheduler0$.

3 Refinement Checking for B

In this section we outline the B notion of refinement. We also outline the trace behaviour of B machines and trace refinement for B machines and relate it to standard B refinement.

Classical B distinguishes between an enabling condition (guard) and a precondition. ProB allows both to be used, but for refinement checking purposes,

it treats preconditions as guards. If we ignore preconditions but allow for guards, then all B operations have a normal form defined by a predicate P relating before state v and after state v' as follows [1, Chapter 6]:

$$\text{ANY } \; v' \; \text{ WHERE } \; P(v, v') \; \text{ THEN } \; v := v' \; \text{ END}$$

Classical B refinement is expressed in terms of a gluing invariant (also called linking invariant) which links concrete states to abstract states. The meaning of operations in B is defined in terms of weakest precondition formulae as are the refinement proof obligations for B. In this paper we will find it more convenient to take a standard relational view of operations and gluing invariants. This view is easily reconciled with the generalised substitution notation by treating the predicate P in the normal form for operations above as characterising a relation between before and after states.

The proof obligations for B correspond to the standard relational definition of forward simulation[4]. Let R be the gluing relation, AI and CI be the abstract and concrete initial states respectively and AOP and COP stand for corresponding abstract and concrete operations. The usual relational definition of forward simulation is as follows [12]:

– Every initial concrete state must be related to some initial abstract state:

$$c \in CI \quad \Longrightarrow \quad \exists a \in AI \; \cdot \; c \; R \; a$$

– If states are linked and the concrete one enables an operation, then the abstract state should enable the corresponding abstract operation and both operations should result in states that are linked:

$$c \; R \; a \; \wedge \; c \; COP \; c' \quad \Longrightarrow \quad \exists a' \; \cdot \; a \; AOP \; a' \; \wedge \; c' \; R \; a'$$

The proof obligations for refinement are automatically generated by, e.g., AtelierB or the BToolkit. The user can then try to prove them using the semi-automatic provers provided by those systems. If the proof obligations are all proven, we know that every execution sequence performed by the refinement machine can be matched by the abstract machine [6]. Automatic refinement checkers work on directly on the execution sequences and try to *disprove* refinement by finding traces that can be performed by the refinement machine but not by the specification. For this we need to formalise the notions of execution sequences (traces) for B.

Traces The use of event traces to model system behaviour is well-known from process algebra, especially CSP [13]. Although event traces are not part of the standard semantic definitions in B, many authors have made the link between B machines and event traces including [6, 8, 21].

[4] This is easy to demonstrate by using the normal form for operations characterised by a before-after predicate and the weakest precondition rules for B operations and refinement.

For a B operation of the form $X \longleftarrow op(Y) \hat{=} S$, we regard execution of operation op with input value a resulting in output value b as corresponding to the occurrence of event $op.a.b$. An event trace is a sequence of such events and the behaviour of a system may be defined by a set of event traces. For example, the following is a possible trace of the scheduler specification of Figure 1:

$$\langle\ new.p1,\ new.p2,\ ready.p1,\ ready.p2,\ enter.p1,\ leave.p1\ \rangle$$

The normal form for a B operation operating on variables v with inputs x and outputs y is characterised by a predicate $P(x, v, v', y)$. Characterising a B operation of the form $X \longleftarrow op(Y)$ as a predicate in this way gives rise to a labelled transition relation on states: state s is related to state s' by event $op.a.b$, denoted by $s \rightarrow^M_{op.a.b} s'$, when $P(a, v, v', b)$ holds. This transition relation $\rightarrow^M_e$ is lifted to traces using relational composition:

$$\begin{array}{rcl} \rightarrow^M_{\langle\rangle} & = & ID \\ \rightarrow^M_{\langle e\rangle t} & = & \rightarrow^M_e \ ;\ \rightarrow^M_t \end{array}$$

Now t is a possible trace of machine M if $\rightarrow^M_t$ relates some initial state to some state reachable through trace t:

$$t \in traces(M) \ = \ \exists c, c' \cdot c \in CI \ \wedge \ c \rightarrow^M_t c'$$

Trace Refinement Checking A machine M is trace refinement of machine N if any trace of N is a trace of M, that is, any trace that is possible in the concrete system is also possible in the abstract system. It is straightforward to show by induction over traces that if we can exhibit a forward simulation between M and N with some gluing relation, then M is trace refined by N. It is known that forward simulation is not complete, i.e., there are systems related by trace refinement for which it not possible to find a forward simulation. The related technique of backward simulation together with forward simulation make simulation complete [12]. A backward simulation is defined as follows:

$$\begin{array}{l} c \in CI \ \wedge \ c \ R \ a \quad \Longrightarrow \quad a \in AI \\ c \ COP \ c' \ \wedge \ c' \ R \ a' \ \Longrightarrow \ \exists a \cdot c \ R \ a \ \wedge \ a \ AOP \ a' \end{array}$$

The B tools produce proof obligations for forward simulation only. Backwards refinement is required when an abstract operation is refined by a concrete operation that is *less* deterministic. Typical developments B involve the reduction of nondeterminism in operations so that forward simulation is sufficient in most cases.

A single complete form of simulation can be defined by enriching the gluing structure. Gardiner and Morgan [11] have developed a single complete simulation rule by using a predicate transformer for the gluing structure. Such a predictate transformer characterises a function from *sets* of abstract states to *sets* of concrete states. Refinement checking in ProB works by constructing a

gluing structure between the concrete and abstract states as it traverses the states spaces of both systems. So that we have a complete method of refinement checking, the ProB checking algorithm constructs a gluing structure similar to the structure used in [11]. The structure constructed by ProB is a relation between concrete states and sets of abstract states[5]: $R \in C \leftrightarrow \mathbb{P}(A)$. On successful completion of an exhaustive refinement checking run the constructed gluing structure R will relate each individual concrete initial state to the set of abstract initial states and for each pair of corresponding concrete and abstract states, the following simulation condition will be satisfied:

$$c\ R\ as\ \wedge\ c\ COP\ c' \implies \exists as' \cdot as\ AOP\ as'\ \wedge\ c'\ R\ as'$$

Here as and as' represent sets of abstract states and $as\ AOP\ as'$ is defined as $AOP[as] = as'$. It can be shown by induction over traces that this entails trace refinement.

4 The Algorithm

We now present an algorithm to perform refinement checking. The gluing structure discussed in the previous Section is stored in $Table$, and for every entry (c, A) the algorithm checks whether all operations of the concrete state c can be matched by some abstract state in A; if not, a counter example has been found, otherwise all concrete successor states are computed and put into relation with the corresponding abstract successor states. To ensure termination of the algorithm it is crucial to recognise when the same configuration is re-examined. This is done by the check "(ConcNode,AbsNodes) ∈ Table". If that check succeeds we know that we can safely stop looking for a counter example. Indeed, if one counter example exists we know that we can find a shorter version starting from the configuration that is already in the Table.

In the previous section we have introduced the relation $\rightarrow^M$, where $s \rightarrow^M_{op.a.b} s'$ signifies that the operation op can be performed with inputs a and outputs b in state s, resulting in a new state s' of the machine M. For the algorithm below it is convenient to also model the initalisations by adding a special state $root$, and extending $\rightarrow^M$ such that $root \rightarrow^M_{initialise\_machine} s$ holds for all valid initial states s of the machine M.

Algorithm 4.1[*Refinement Checking*]

Input: An abstract machine M_A and a refinement machine M_R
Table := {} ; Res := refineCheck(root,{root});
if Res = ⟨⟩ **then println** 'Refinement OK'
else println('Counter Example:',Res)
end if

[5] Our structure is not isomorphic to a predicate transformer structure as it doesn't satisfy closure properties of monotonic predicate transformers.

```
function refineCheck(ConcNode,AbsNodes)
 if (ConcNode,AbsNodes) ∈ Table then
   return ⟨⟩
 else
   Table := Table ∪ {(ConcNode,AbsNodes)};
   for all CSucc,Op such that ConcNode →_Op^{M_R} CSucc do
      TraceS := concat(Trace,[(Op,CSucc)]);
      ASuccs := {as | ∃an ∈ AbsNodes ∧ an →_Op^{M_A} as};
      if ASuccs = ∅ then
         return TraceS
      else
         Res := refineCheck(CSucc,ASucss,TraceS);
         if Res ≠ ⟨⟩ then  return Res; end if
      end if
   end for
 end if
end function
```

Implementation We have actually performed two implementations of the above algorithm. The first one is implemented inside the ProB toolset, i.e., using SICStus Prolog. The tabling is done by maintaining a Prolog fact database which is updated using `assert/1`. The second implementation has been done in XSB Prolog. The code of the XSB refinement checker is almost identical, but instead of using a Prolog fact database it uses XSB's efficient tabling mechanism [20]. As we will see later, this implementation is faster than the SICStus Prolog one, but the overhead of starting up a new XSB Prolog process and loading the states space is only worth the effort for larger state spaces with no or difficult to find counter examples. From a pragmatic point of view, this approach also requires the ProB user to separately install XSB Prolog.

For both implementations the abstract state space currently has to be computed beforehand (using ProB). To ensure completeness of the refinement checking, it should be fully computed. However, our refinement checker also allows the abstract state space to be only partially computed. In that case, the refinement checker will detect whether enough of the state space has been computed to decide the refinement (and warn the user if not).

For the SICStus Prolog implementation the state space of the implementation can, but does not have to be computed beforehand. In other words, the implementation state space will be expanded *on-the-fly*, depending on how the refinement checking algorithm proceeds. This is of course most useful when counter examples are found quickly, as in those cases only a fraction of the state space will have to be computed. In future work we plan to enable this on-the-fly expansion also for the abstract state space. For the XSB implementation, running separateley from ProB, this interaction is currently not possible, and hence both the abstract and implementation state space have to be computed beforehand.

5 Experiments

To test our refinement checker we have conducted a series of experiments with various models. As well as using the scheduler example from Section 2, we have experimented with a much larger development of a mechanical by press by Abrial [2]. The development of the mechanical press started from a very abstract model and went through several refinements. The final model contained "about 20 sensors, 3 actuators, 5 clocks, 7 buttons, 3 operating devices, 5 operating modes, 7 emergency situations, etc." [2]. Figure 5 in Appendix A shows the first three machines in the refinement hierarchy, as displayed by ProB. We were able to apply our new refinement checker to successfully validate various refinement relations. Furthermore, as no abstraction was required for ProB (i.e., all sets were already finite to start with), the refinement checker can actually be used in place of the traditional B refinement provers. In other words, were thus able to automatically prove refinement using our new tool. To check the ability of our tool to find errors we have also applied it to an erroneous refinement (m2_err.ref), and ProB was able to locate the problem in a few seconds. We have also experimented with a simple example of a server allowing clients to log in. Precise timings and results for these and other experiments are presented in the next subsections.

Consistency checking In a first phase we have performed classical consistency and deadlock checking on our examples using ProB's model checker. The results can be found in Table 1, and give an indication of the size of the state space and how expensive it is to compute the operational semantics. The experiments were all run on a PowerPC G5 Dual 2.5 GHz, running Mac OS X 10.3.9, SICStus Prolog 3.12.1 and ProB version 1.1.5. Note, while the machine had 4.5 Gigabyte of RAM, only 256 Megabyte are available in SICStus Prolog 3.12 for dynamic data (such as the state space of B machines). scheduler0.mch and scheduler1.ref are the machines presented above in Section 2 for 3 processes, while scheduler0_6.mch and scheduler1_6.ref are the same machines but for 6 processes. The machines m0.mch, m1.ref, m2.ref, m2_err.ref, and m3.ref are from the mechanical Press example discussed above. Server.mch is a simple B machine describing the server example, while ServerR.ref is a refinement thereof.

Refinement checking Table 2 are the results of performing various refinement checks on these machines. Entries marked with an asterisk mean that no previous consistency checking was performed, i.e., the operational semantics of the implementation machine was computed on-the-fly, as driven by the refinement checker. For entries without an asterisk the experiment was run straight after the consistency checking of Table 1, i.e., the operational semantics was already computed and the time is thus of the refinement checking proper. The figures show that our checker was very effective, especially if counter examples existed.

In Table 3 we have conducted some of the experiments where the refinement checker is run as a separate process using XSB Prolog [20], rather than inside ProB under SICStus Prolog. Our experiments confirm that XSB's tabling mechanism leads to a more efficient refinement checking (cf. the third column).

Table 1. ProB consistency checking and size of state space

Machine	Time	States	Transitions
Server.mch	0.013 s	5	9
ServerR.ref	0.05 s	14	39
scheduler0.mch	46 s	55	190
scheduler1.ref	0.93 s	145	447
scheduler0_6.mch	41.37 s	2,188	14,581
scheduler1_6.ref	501.61 s	37,009	145,926
m0.mch	3.19 s	65	9,924
m1.ref	20.38 s	293	47,574
m2.ref	44.29 s	393	59,588
m2_err.ref	31.51 s	405	61,360
m3.ref	364.90 s	2,693	385,496

However the time to start up XSB and load the state space is not negligible, meaning that the XSB approach does not always pay off. This can be seen in the fourth column which contains the total time for loading and checking: e.g., the approach pays off for the m2.ref check against m1.ref (overall gain of 30 seconds) but not for the smaller examples nor when a counter example is found quickly.

Comparison with FDR We have compared our new refinement checker against the most widely known refinement checker, namely FDR [10]. FDR is a commercial tool for the validation of CSP specifications [13]. While B machines cannot easily be translated into CSP, the state space explored by ProB can easily be translated into a CSP specification using just choice and process definitions. While this automatically generated CSP is not a typical CSP specification, it is still useful for two purposes. First, it allows us to evaluate our refinement Algorithm 4.1 against the counterpart in FDR. Second, we can determine whether it would make sense, from an implementation point of view, to outsource the refinement checks to FDR, rather than using our own algorithm.

The experiments were conducted as follows. After consistency checking (Table 1) the state space was saved as a simple CSP file using an export facility added to ProB. Basically, every state was encoded as a CSP Process and defined by an external choice of all the outgoing transitions. Operation arguments were flattened out, e.g., something like $new(p1)$ got translated into a new CSP channel new_p1.[6] As an illustration, here are the first few lines for m0.mch:[7]

```
Nroot = initialise_machine->N3448 [] initialise_machine->N3449 []
        initialise_machine->N3450 [] initialise_machine->N3451
```

[6] This had no impact for the mechanical press examples written in Event B style, as there are no operation arguments in any of the machines anyway.

[7] We have also used internal rather than external choice, to see whether this improved FDR's performance. Unfortunately, FDR was unable to compile those models (producing a "failed to compile ISM" error).

Table 2. ProB refinement checking and size of refinement relation

Refinement	Specification	Time	Size of table
Successful refinements:			
ServerR.ref	Server.mch*	0.05 s	14
”	Server.mch	0.00 s	”
scheduler1.ref	scheduler0.mch*	0.73 s	145
”	scheduler0.mch	0.00 s	”
scheduler1_6.ref	scheduler0_6.mch	3.80 s	37,009
m1.ref	m0.mch*	25.4 s	585
”	m0.mch	6.28 s	”
m2.ref	m0.mch	8.10 s	785
m2_err.ref	m0.mch	8.13 s	809
m2.ref	m1.ref	70.57 s	3,804
m3.ref	m0.mch	51.96 s	5,345
m3.ref	m1.ref	429.37 s	24,039
m3.ref	m2.ref	333.85 s	21,205
Counter examples found:			
scheduler1err.ref	scheduler0.mch*	0.12 s	19
scheduler1err_6.ref	scheduler0_6.mch*	1.80 s	121
m1.ref	m2.ref	0.01 s	13
m2_err.ref	m1.ref*	4.22 s	92
”	m1.ref	0.03 s	”

```
N3448 = demarrer_presse->N3452 [] presse_descend->N3450 []
    occuper_main_gauche->N3453 [] occuper_main_droite->N3453 [] ...
```

FDR 2.8.1 was then run on the same hardware as for the earlier experiments to check CSP trace refinement and the results can be found in Table 4. Timings do not include the time needed to load the CSP file, but include the compilation, normalization and checking time of FDR. Due to a small bug in the TclTk interface of FDR timings were not displayed for ServerR.ref and scheduler1.ref; but the response was very quick. For scheduler1_6.ref FDR ran for 2.812 hours (CPU usage) and then generated a “failed to compile ISM” error message. For the other examples FDR spent most of the time on compilation and normalization of the CSP model. This means subsequent refinement checks of the same combination of machines would have been substantially faster. In practice, however, there is only one refinement check that one is interested in for any two machines (namely that the “refinement” machine is a refinement of the specification machine).

We have also modelled the scheduler and its refinement in CSP by hand using what we believe is a natural CSP style using CSP constructs such as parallelism and synchronization wherever possible. These are named scheduler1.csp and scheduler0.csp and are restricted to 3 processes each (see Fig. 3 in the appendix). We had to manually limit the size of the communication queue for FDR to terminate, but after that refinement checking was very quick. Unfortunately, due to the timing bug mentioned above we could not evaluate whether this was

Table 3. ProB refinement checking using XSB Prolog

Refinement	Specification	Checking Time	Total Time
Successful refinements:			
ServerR.ref	Server.mch	0.00 s	0.06 s
scheduler1.ref	scheduler0.mch	0.00 s	0.11 s
m1.ref	m0.mch	2.85 s	13.76 s
m2.ref	m0.mch	3.61 s	18.15 s
m2.ref	m1.ref	26.66 s	40.24 s
m3.ref	m2.ref	136.12 s	219.03 s
Counter examples found:			
m1.ref	m2.ref	0.00 s	22.68 s
m2_err.ref	m1.ref	0.01 s	12.79 s

faster than checking the automatic translation. We have then tried to check the natural CSP specification for 6 processes, but unfortunately FDR was unable to deal with that, generating a "failed to compile ISM" message after 11 s.

For checking m1.ref against m0.mch it can be noted that our algorithm is about 16 times faster than FDR (one has to compare the 101 s against 6.28 s as the computation of the operational semantics has in both cases been done beforehand by ProB; arguably FDR is at a slight disadvantage as the state space is in CSP form rather than stored as facts in a Prolog database; still the CSP processes are very simple and thus should not cause a large overhead). Our relatively simple refinement checking algorithm thus proves surprisingly effective in practice (or, depending on your point of view, FDR proves to be surprisingly ineffective at checking CSP models which directly encode fully expanded transition systems). When counter examples exist the difference is even more dramatic, as FDR spends a lot of effort on compiling and normalising the CSP specification before (quickly) finding the counter example. We believe that this difference is at least partly explained by the fact that our algorithm normalises the abstract state space on the fly rather than beforehand. Indeed, Algorithm 4.1 can also be viewed as linking concrete states with sets of abstract states, by exploring in parallel all possible alternatives of the abstract machine. This corresponds to normalisation in FDR [10], but done *on-the-fly rather* than beforehand. Thus, when a counter example is found (quickly) only a fraction of the abstract space will have been normalised. Furthermore, even when no counter example is found, only that part of the abstract system is normalised which is in common with the implementation. As the implementation often has more restricted behaviour, this can result in big reductions and could explain why our tool can handle much bigger examples (e.g., scheduler1_6.ref). FDR's approach would only pay off if one did many refinement checks of the same abstract system, covering a large part of its state space, and if one has enough memory to normalise the entire abstract system. In our particular case studies, this was not the case. We plan to undertake a more thorough comparison of ProB and FDR by comparing more examples modelled naturally in CSP and naturally in B. Still, as a preliminary conclusion, we can state that our algorithm compares favourably with FDR.

Table 4. Refinement checking on the already expanded state space with FDR

Refinement	Specification	Time	States	Transitions
Successful refinements:				
ServerR.ref	Server.mch	< 1s	5	9
scheduler1.ref	scheduler0.mch	< 1s	69	205
scheduler1_6.ref	scheduler0_6.mch	2.812 hours	error	error
scheduler1.csp	scheduler0.csp	< 1 s	68	204
m1.ref	m0.mch	101 s	447	71,910
m2.ref	m0.mch	130 s	613	92,768
m2.ref	m1.ref	152 s	3,239	492,401
Counter examples found:				
m1.ref	m2.ref	120 s	2	8
m2_err.ref	m1.ref	150 s	464	71,107

6 Extensions

Singleton Failures We have extended our refinement checking algorithm to also check singleton failure refinement (see, e.g., [4]). A *singleton failure trace* is a pair consisting of a trace t as defined earlier and either the empty set or singleton set containing a single operation F (with arguments). The intuitive meaning of $(t, \{F\})$ is that the machine can perform all the operations in the trace t and then be in a state where the operation F is not enabled, i.e., can be refused. The meaning of $(t, \varnothing)$ is that the machine can perform all the operations in t and then all operations are enabled for all possible arguments. A machine m_1 is said to be a *singleton failure refinement* of m_0 iff all singleton failure traces of m_1 are also singleton failure traces of m_0. Singleton Failures refinement can be situated in between trace refinement and CSP's failure refinement as implemented in FDR (in the latter, rather than talking about an single operation that can be refused one talks about sets of possible combinations of operations that can be refused).

To implement singleton failure refinement checking, Algorithm 4.1 has been extended to check for an additional condition when a counter example is found. More precisely, the function $\mathit{refineCheck}(\mathit{ConcNode}, \mathit{AbsNodes})$ also looks for operation calls that are possible in all states in $\mathit{AbsNodes}$ but not in $\mathit{ConcNode}$.

As B does not have the distinction between internal and external choice, singleton failure refinement is mainly useful for refinements that should not decrease the choices offered by the machine, e.g., data refinement or when moving non-deterministic choices later. Note, however, when treating parameters of B operations, a choice of input values could be treated as an external choice, while a choice of output values could be treated as internal. In future, we plan to make this distinction. (It is not necessary to make this distinction for trace refinement, since the traces model does not distinguish internal and external choice.)

Some empirical results can be found in Table 5. All experiments were run on-the-fly, i.e., the implementation transition was not computed beforehand. As one can see, several refinement checks that were successful using trace refinement now yield a counter example. For example, for the m1.ref vs m0.mch check the algorithm finds the counter example $(\langle \mathit{initialise\_machine} \rangle, \mathit{demarrer\_presse})$,

meaning that there is an initial state of m1.ref where *demarrer_presse* is not enabled, but in all initial states of $m0.mch$ this operation is enabled.

Table 5. ProB refinement checking using singleton failures

Refinement	Specification	Time	Size of table
Successful refinements:			
ServerR.ref	Server.mch*	0.07 s	14
Counter examples found:			
scheduler1.ref	scheduler0.mch*	0.06 s	9
m1.ref	m0.mch*	0.05 s	2
m2.ref	m0.mch*	0.06 s	2
m2.ref	m1.ref *	0.07 s	2
m3.ref	m2.ref *	0.08 s	2

Application to B and CSP In recent work [5] we have shown how to combine B and CSP for specification purposes (a specification is partly written in B and partly in CSP) or for property checking of B machines (the CSP is used as a temporal property that a B machine must satisfy). Our new refinement checker is language independent, in the sense that any interpreter plugged into the ProB toolset can be used. In practice this means that we can check whether a B machine is a refinement of a CSP machine, or the other way around. For example, the mutual exclusion property of the scheduler of Section 2 can be specified as the following CSP process: $LOCK = enter?p \rightarrow leave.p \rightarrow LOCK$. We can check that both B schedulers (Figures 1 and 2) are trace refinements of the $LOCK$ CSP process. We can also check whether a combined B/CSP specification is a refinement of another combined specification. One can even use other formalisms, such as Object Petri nets as implemented in [9]. All this opens up new possibilities for validation.

7 Related and Future Work, and Conclusion

The idea of using (tabled) logic programming for verification is not new (see, e.g., [18]). The inspiration for the current refinement checker came from the earlier developed CTL model checker presented in [15]. Another related work is [3], which presents a bisimulation checker written in XSB Prolog.

In future, we plan to extend the refinement checker to also allow on-the-fly expansion of the abstract state space. We also wish to move away from the pure depth first strategy that it currently employs; using a similar mixed breadth-first depth-first strategy as the ProB model checker. This should allow the refinement checker to be applied when the abstract and implementation state spaces are big or even infinite. Another extension is to actually check whether a gluing invariant provided by the user can be satisfied. To improve the scalability of ProB we are also looking at symbolic state space reduction techniques.

We have presented the first automatic refinement checker for B. The checker is implemented within ProB and has been applied to various case studies. Our experiments have shown that, at least for the case studies under consideration, the

algorithm is very effective and surprisingly competitive. Its ability to normalise the abstract machine on-the-fly seemed to be a key ingredient of its success.

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. J.-R. Abrial. Case study of a complete reactive system in Event-B: A mechanical press controller. *Tutorial at ZB'2005*. Available at `http://www.zb2005.org/`.
3. S. Basu, M. Mukund, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. M. Verma. Local and symbolic bisimulation using tabled constraint logic programming. In *Proceedings ICLP'01*, LNCS 2237, pages 166–180, 2001. Springer.
4. C. Bolton and J. Davies. A comparison of refinement orderings and their associated simulation rules. *Electr. Notes Theor. Comput. Sci.*, 70(3):440 –453, 2002.
5. M. Butler and M. Leuschel. Combining CSP and B for specification and property verification. In *Proceedings FM 2005*, Newcastle upon Tyne, 2005. In press.
6. M. J. Butler. An approach to the design of distributed systems with B AMN. In J. P. Bowen, M. G. Hinchey, and D. Till, eds, *Proceedings ZUM '97*, LNCS 1212, pages 223–241. Springer, 1997.
7. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
8. S. Dunne and S. Conroy. Process refinement in B. In H. Treharne, S. King, M. C. Henson, and S. Schneider, eds, *Proceedings ZB 2005*, LNCS 3455, pages 45–64. Springer, 2005.
9. B. Farwer and M. Leuschel. Model checking object Petri nets in Prolog. In *Proceedings PPDP '04*, pages 20–31, 2004. ACM Press.
10. Formal Systems (Europe) Ltd. *Failures-Divergence Refinement – FDR2 User Manual*.
11. P. H. B. Gardiner and C. Morgan. A single complete rule for data refinement. *Formal Asp. Comput.*, 5(4):367–382, 1993.
12. J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In *Proceedings ESOP 86*, LNCS 213, pages 187–196. Springer, 1986.
13. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice–Hall, 1985.
14. M. Leuschel and M. Butler. ProB: A Model Checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, eds, *Proceedings FME 2003*, LNCS 2805, pages 855–874. Springer, 2003.
15. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, ed, *Proceedings of LOPSTR'99*, LNCS 1817, pages 63–82, 2000.
16. M. Leuschel and E. Turner. Visualizing larger states spaces in ProB. In H. Treharne, S. King, M. Henson, and S. Schneider, eds, *Proceedings ZB'2005*, LNCS 3455, pages 6–23. Springer-Verlag, April 2005.
17. B-Core UK Limited. B-toolkit manuals. 1999.
18. C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, Y. Dong, X. Du, A. Roychoudhury, V. N. Venkatakrishnan. XMC: A logic-programming-based verification toolset. In *Proceedings CAV 2000*, pages 576-580.
19. A. Roscoe. *The Theory and Practice of Concurrency*. Prentice–Hall, 1998.
20. K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proceedings SIGMOD'94*, pages 442–453, May 1994. ACM.
21. S. Schneider and H. Treharne. Communicating B machines. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, eds, *Proceedings ZB 2002*, LNCS 2272, pages 416–435. Springer, 2002.
22. Steria. Atelier B, user and reference manuals. 1997.

A Appendix (for referees)

```
-- A CSP specification and refinement of a scheduler

psize = 3
PID = {1..psize}
channel new : PID
channel ready : PID
channel enter : PID
channel leave : PID

-- Specification

NEWPROC(p) = new.p -> PROC(p)

PROC(p) =
    ready.p -> enter.p -> leave.p -> PROC(p)

MUTEX = enter?p -> leave.p -> MUTEX

SCHEDULER0 =
   (||| p:PID @ NEWPROC(p)) [| {| enter, leave |} |]  MUTEX

-- Refinement

QUEUE(q) =
       #q<psize & ready?p -> QUEUE(q^<p>)
    [] q!=<> & enter.head(q)   -> QUEUE(tail(q))

SCHEDULER1 =
   ((||| p:PID @ NEWPROC(p)) [| {| enter, leave |} |]  MUTEX)
      [| {| ready, enter |} |] QUEUE(<>)

assert SCHEDULER0 [T= SCHEDULER1
```

Fig. 3. CSP specification and refinement of a scheduler

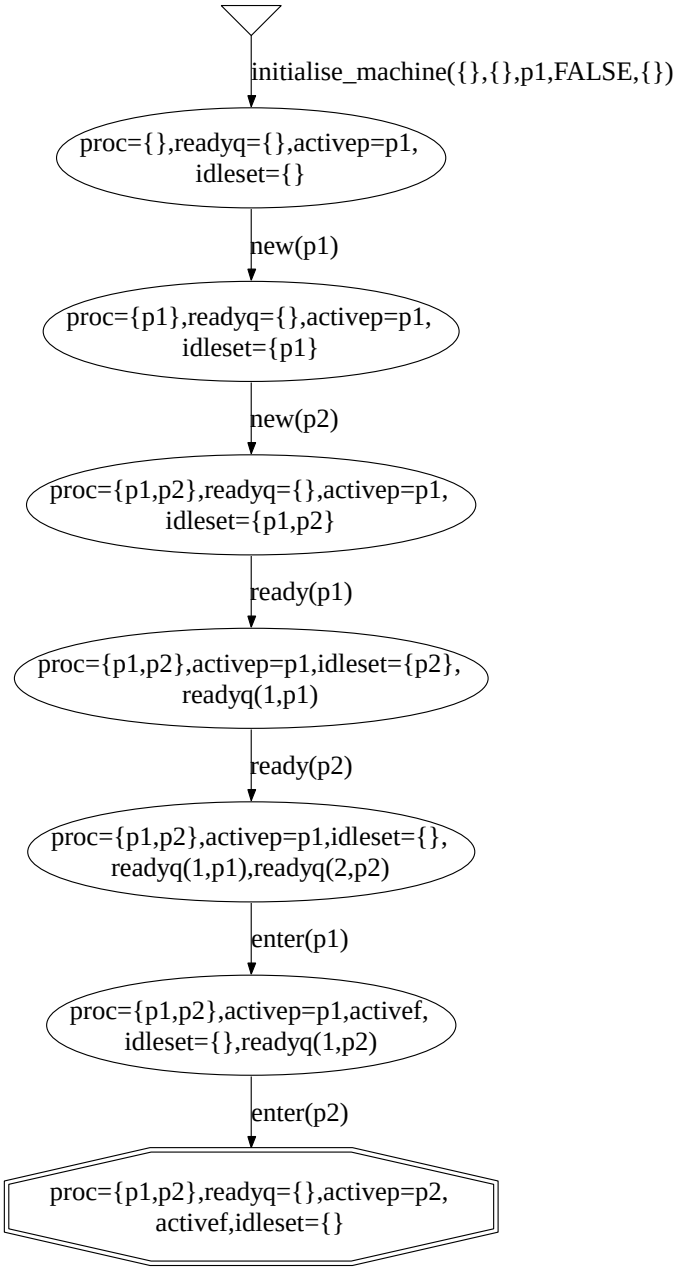

Fig. 4. Counterexample for incorrect refinement of scheduler as displayed by ProB

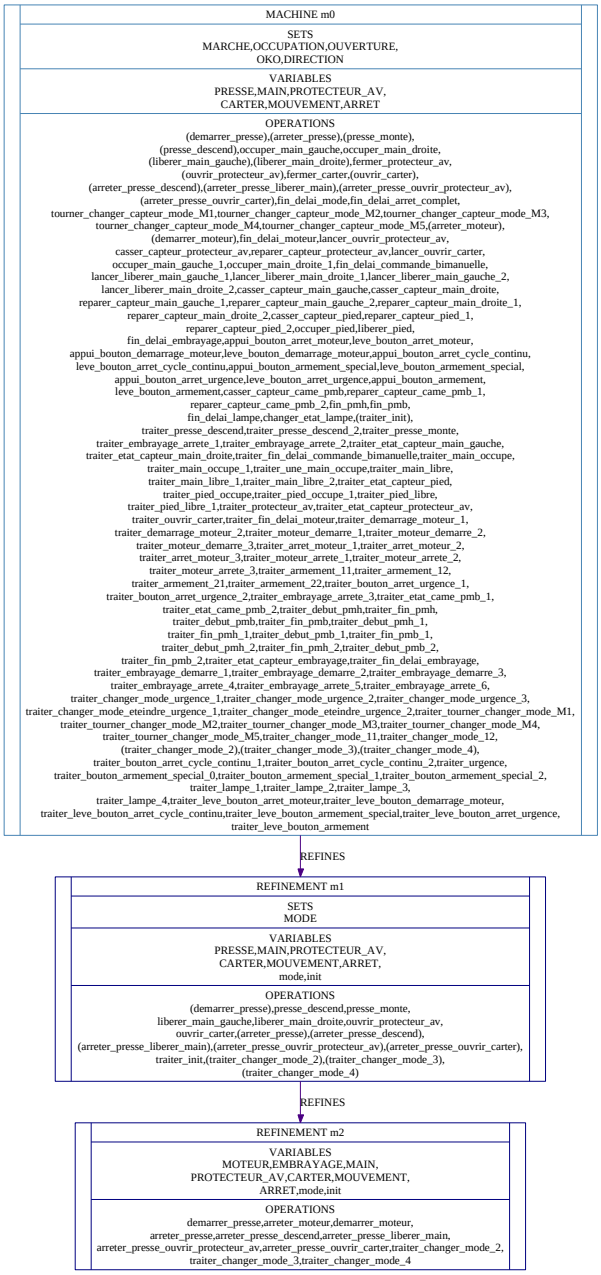

Fig. 5. Mechanical Press: Top-level Machine Hierarchy displayed by ProB