# Petri Nets to B-Language Transformation in Software Development

**Štefan Korečko, Branislav Sobota**

Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9, 041 20 Košice, Slovakia, stefan.korecko@tuke.sk, branislav.sobota@tuke.sk

*Abstract: Petri nets and B-Method represent a pair of formal methods, for computer systems engineering, with interesting complementary features. Petri nets have nice graphical representation, valuable analytical properties and can express concurrency. B-Method supports verified software development. To gain from these complements, a mapping from Petri nets to the language of B-Method has been defined and its correctness proved. This paper presents, by means of a case study, the usefulness of incorporation of Petri net designs in a software application developed by B-Method. Modifications of this mapping intended for the Event-B method and treatment of concurrency are also discussed.*

*Keywords: Petri nets; B-Method; Event-B; refinement; software development; concurrency*

## 1 Introduction

Petri nets (PN) [8] are a formal language able to naturally express behaviour of non-deterministic, parallel and concurrent systems. PN offer an easy to understand graphical notation and analytical methods, which, for example, allows one to derive invariant properties from the structure of the net. There are many types of PN with different expressional and modelling power. PN can be used for modelling, analysis and simulation of systems from various areas, including network protocols, operating systems, workflow management and business processes [5] and robotic manufacturing systems [18, 19]. On the other hand, the B-Method (B) [1] is a state based, model-oriented formal method intended for software development. Its strength lies in a well-defined development process, which allows one to specify a software system as a collection of components called B-machines and refine such an abstract specification to a concrete one. The concrete specification can be automatically translated to ADA, C or another programming language. An internal consistency of the abstract specification and correctness of each refinement step are verified by proving a set of predicates called proof obligations (PObs). The whole development process, including proving, is supported by an industrial-strength software tool called Atelier B.

Important properties of these methods are complementary: in PN invariant properties can be derived from the structure of the net; in B we have to specify invariants manually. B has verified development process but is intended only for sequential systems; PN can express concurrency but lack a development process. This led to an idea of integration of these methods. The idea was realized in the form of semantics-preserving mappings from the language of B-Method (B-language) to Coloured Petri nets [14] and from Evaluative Petri nets (EvPN) to the B-language [13, 15]. The second mapping transforms each EvPN to a computationally equivalent (bisimilar) B-machine. Its formal definition and proof of correctness was presented in [13, 15] and it has even also been shown that it can be used for an additional analysis of PN in B-Method [16]. However, its usefulness for software development has yet to be treated. Therefore, in this paper we present a case study that demonstrates how a B-machine, obtained by the second mapping, can be used as a component of a software system, developed by B-Method. We also outline an approach to reflect concurrent aspects of PN models in B and describe how the mapping can be adapted for a new version of B, called Event-B. The case study uses Place-transition nets, which can be regarded as a subclass of EvPN, so both PN and the mapping are treated to this extent only.

The rest of the paper is organized as follows. Sections 2 and 3 provide necessary information about Place-transition (PT) nets and B. Section 4 defines the mapping from PT nets to B-Method and shows its application to both the “classical” B and Event-B. Section 5 presents the case study and section 6 discusses the approach to reflect concurrent aspects. Section 7 describes related work and in the conclusions we deal with plans for future research and development.

## 2 Place-Transition Nets

Place-transition nets (PT nets) [8], also called Generalised Petri nets [10], are one of the most commonly used and researched type of PN. A PT net is defined [10] as a 5-tuple

$$N=(P, T, pre, post, m_0), \tag{1}$$

where $P=\{p_1,...p_k\}$ is a finite set of places, $T=\{t_1,..., t_n\}$, is a finite set of transitions, *pre:* $P\times T\rightarrow \mathbb{N}$ is a preset function, post: $P\times T\rightarrow \mathbb{N}$ is a postset function and $m_0\in \mathbb{N}^{|P|}$ is the initial marking. $\mathbb{N}$ is the set of natural numbers, including $0$.

PT net is usually depicted as an oriented graph with places (circles or ellipses) and transitions (bold lines or rectangles) as vertices (Fig. 1). Arcs are defined by the functions *pre* and *post*: When $pre(p,t) \neq 0$, then there is an arc from $p$ to $t$, when $post(p,t) \neq 0$, then there is an arc from $t$ to $p$. If the value of $pre(p,t)$ or $post(p,t)$ is greater than 1 then it is written next to the corresponding arc. For example, in the net in Fig. 1 we have $pre(p_2,t_1)=10$, $pre(p_1,t_1)=0$, $post(p_1,t_1)=1$ and $post(p_4,t_2)=1$.

With each transition *t* we can associate two sets: a set of pre-places of *t* ( $^{\bullet}t$ ) and a set of post-places of *t* ( $t^{\bullet}$ ). They are defined as follows:

$$^{\bullet}t = \{p \mid p \in P \wedge pre(p,t) \neq 0\}\,,\ \ t^{\bullet} = \{p \mid p \in P \wedge post(p,t) \neq 0\}\,. \tag{2}$$

A *marking* of PT net *N* is a function $m\colon P \rightarrow \mathbb{N}$. Value of $m(p)$ is the number of tokens in the place *p*. Markings represent states of a Petri net. Markings are often written as vectors, $m=(m(p_1),\ldots m(p_k))$. For example the initial marking of the net from Fig. 1 is $m_0=(0,10,0,10)$. A transition $t \in T$ is enabled (feasible) in marking *m*, if and only if

$$\forall p \in {}^{\bullet}t : m(p) \geq pre(p,t)\,, \tag{3}$$

When *t* is enabled, it can be executed (fired). The result of its execution is a new marking $m'(p) \in \mathbb{N}^{|P|}$:

$$m'(p) = m(p) - pre(p,t) + post(p,t)\,, \tag{4}$$

A marking, which can be reached from the initial marking of some PT net *N* by firing some sequence of enabled transitions, is called a reachable marking of *N*.

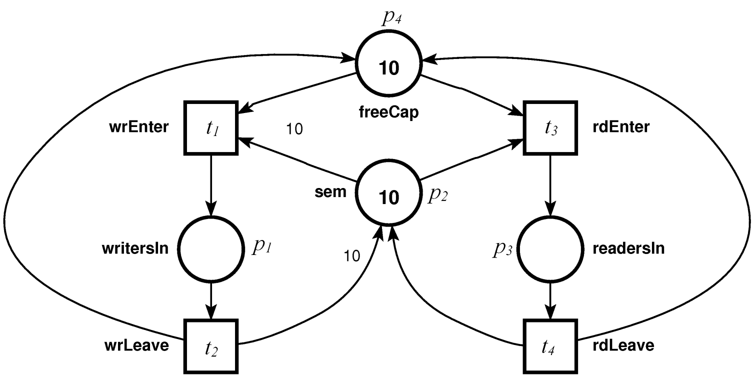

Figure 1
PT net representing limited variant of RW problem

An example of PT net can be seen in Fig. 1. The net specifies a solution of a limited variant of the so-called readers-writers (RW) problem. The RW problem can be described as follows: We have shared contents (a library) that can be accessed concurrently by two kinds of processes: readers, which only read the contents and writers, which also modify it. The problem is to ensure that no reader will access the contents while some writer is modifying it. In the limited variant of the problem, the library capacity, that is, the maximal number of processes accessing concurrently, is limited. The capacity is set to *10* in our example. The number of tokens in *writersIn* is the number of writers in the library (i.e. of processes modifying it) and the number of tokens in *readersIn* is the number of readers in the library. Firing of *wrEnter* means that a writer process started accessing the library and *wrLeave* that it finished accessing the library; *rdEnter* and *rdLeave* have similar meaning for readers. We mentioned earlier that a handful of analytical methods are available for PN. One of them is a computation of so-called place invariants and by applying it to our net we get equations (5) and (6), which hold in any reachable marking *m* of the net. The equations prove that the net has required properties, such as the mutual exclusion of readers and writers.

$$m(writersIn)+ m(freeCap)+ m(readersIn) = 10 \tag{5}$$
$$10 \cdot m(writersIn)+ m(sem)+ m(readersIn) = 10 \tag{6}$$

## 3 B-Method

As it was written above, the *B-Method* (*B*) [1] allows us to specify a software system as a collection of *B-machines* and to refine such an abstract specification to an implementable one, while providing formal means to prove that both abstract specification and its refinements are consistent. All components in B are written in its own *B-language* (also called B-AMN), which is based on Zermelo-Fraenkel set theory and E.W. Dijkstra's Guarded Command Language [9].

Table 1

General structure of B-machine (left) and Refinement (right)

| | |
|---|---|
| `MACHINE M(p)`<br>`CONSTRAINTS C`<br>`SETS St`<br>`CONSTANTS k`<br>`PROPERTIES Bh`<br>`VARIABLES v`<br>`DEFINITIONS D`<br>`INVARIANT I`<br>`ASSERTIONS A`<br>`INITIALISATION T`<br>`OPERATIONS`<br>`  y←op(x) =`<br>`    PRE P THEN S END`<br>`  …`<br>`END` | `REFINEMENT R(p)`<br>`REFINES M`<br>`SETS St1`<br>`CONSTANTS k1`<br>`PROPERTIES Bh1`<br>`VARIABLES w`<br>`DEFINITIONS D1`<br>`INVARIANT J`<br>`ASSERTIONS A1`<br>`INITIALISATION T1`<br>`OPERATIONS`<br>`  y←op(x) =`<br>`    PRE P1 THEN S1 END`<br>`  …`<br>`END` |

Each B-machine consists of several clauses (Table 1). The most important are the `MACHINE` clause with a name `M` of the machine and a list `p` of its formal parameters, the `VARIABLES` clause with a list v of state variables, `INVARIANT` with properties `I` of the state variables, `INITIALISATION` with an operation `T` that establishes an initial state of the machine, and `OPERATIONS` that contains its operations. The B-Method obeys the encapsulation principle, so only operations of the given machine can modify its variables. We say that the machine is internally consistent if `I` holds in each of its states. `St` is a list of deferred and enumerated sets. These are regarded as new types. Constants of the machine are listed in `k` and a predicate `Bh` defines properties of `St` and `k`. `D` is a list of macro definitions and `A` is a list of lemmas used to simplify proof of PObs. Only the `MACHINE` clause is mandatory.

Table 2
Selected generalized substitutions and their intuitive meaning

| **GS** | **meaning of GS** |
|---|---|
| `skip` | Empty GS (do nothing). |
| $x := e$ | Assignment of value of expression *e* to variable *x*. |
| $S_1 ; S_2$ | Sequential composition (do GS $S_1$, then GS $S_2$). |
| $S_1 \parallel S_2$ | Parallel composition (do $S_1$ and $S_2$ at once). |
| `CHOICE` $S_1$ `OR` $S_2$ `END` | Do $S_1$ or $S_2$. |
| `PRE` *E* `THEN` $S_1$ `END` | If predicate *E* holds, do $S_1$. Otherwise, do anything. |
| `SELECT` *E* `THEN` $S_1$ `END` | If *E* holds, do $S_1$. Otherwise, do not execute. |
| `IF` *E* `THEN` $S_1$ `ELSE` $S_2$ `END` | If *E* holds, do $S_1$. Otherwise, do $S_2$. |
| `VAR` *v* `IN` $S_1$ `END` | For any values of local variables from the list *v* do $S_1$. |
| `ANY` *v* `WHERE` *E* `THEN` $S_1$ `END` | For any values of variables from *v* that satisfy *E* do $S_1$. If no values satisfy *E*, do not execute. |

Every operation has two parts: a header and a body. The header includes its name (`op`) and optional input and output parameters (`x`, `y`). The body is written in the *Generalized Substitution Language* (*GSL*), a part of the B-language. GSL contains several constructs, or "commands", called *generalized substitutions* (*GS*). Some of them are listed in Table 2. The formal semantics of GSL is defined by the weakest pre-condition calculus [9]. Standardly, the body has the form of `PRE` GS, however if `P` ≡ `TRUE` then it consists only of `S`. The PObs for B-machine assert that `T` always establishes an initial state in which `I` holds and that for each operation `op` it holds that if `op` is executed from a state satisfying `I` and `P` then it always terminates in a state satisfying `I`.

One of the valuable assets of B-Method is its verified stepwise refinement process. This means that an abstract specification, consisting of B-machines (*MM*), can be modified in one or more steps into a form of concrete, implementable, specification. There are two additional components used during the refinement process – *Refinement* (*RR*) and *Implementation* (*II*). Structures of *MM*, *RR* (Table 1) and *II* are similar, but there are some differences. For example GS "`;`" and loops are not allowed in *MM* and "||", `PRE`, `SELECT` and `CHOICE` are not allowed in *II*. A *RR* or *II* can refine only one *MM* or *RR* but one *MM* or *RR* can be refined by more *RR* or *II*. To refine means to modify data or operations. Interfaces (i.e.

parameters and operation headers) of a refining and a refined component have to be the same. The structure of *RR* can be seen in Table 1. Invariant `J` of the refinement `R` defines properties of `w` but also a relation between `v` and w. Whether `J` is established by `T1` and maintained by all operations of `R` is, again, verified by proving the PObs.

We stated earlier that specification in B usually consists of more than one component. To access contents of one component from another, several composition mechanisms can be used. For example, `SEES` and `USES` allow different level of read-only access, `INCLUDES` allow to call operations of accessed component in the accessing one and `IMPORTS` replaces `INCLUDES` in implementations. These mechanisms are usually defined right after the `CONSTRAINTS` or `REFINES` clause.

## 3.1 Event-B

In the late 1990s a development of a new version of B-Method, called *Event-B* [2], started. Event-B was meant to be a reinvention of B-Method (now also called the classical B), based on existing experiences with the practical use of B and a wide variety of research results related to B. It has a broader scope – it is intended for computer system modelling and development in general and is not only meant for software. Specifications are called models and composition is possible via `SEES` and `EXTENDS` mechanisms. We have two types of specification components in Event-B: *Context* with sets, constants and their properties and *Machine* with everything else. Machines can be refined, and refined components are called machines, too. Operations are replaced by events and the initialisation is now one of them. Event-B uses a modified version of B-language. Most of GS have been dropped and each event has the form

`any` *v* `where` *E* `then` *S* `end`,

where *v* is a list of local variables, *E* is a list of predicates, called guards and *S* is a list of (possibly multiple and non-deterministic) assignments, called actions. All its guards have to hold for an event to be executable (enabled) and when executed all its actions are run at once. On the other hand, there are some new additions to B-language that allows one to specify names and additional properties of specification components and their parts. The concept of refinement has been modified, too; in Event-B it is possible to refine one event into several events.

One may wonder how a sequential program can be described in Event-B without the sequential composition, conditional statements and loops. But the general model of an Event-B model execution is such that all its events, except of the initialisation, are executed in a loop and the loop terminates if no event is enabled. If there are more enabled events, one of them is selected non-deterministically. So, by a careful design of guards and actions we can ensure that events of a model will be executed in desired order. This is covered in more detail in [2].

## 4 Petri Net to B-Machine Transformation

The transformation has been originally designed for a Turing-complete low-level type of PN, called Evaluative Petri Nets (EvPN). As the type of PN used in our case study is PT nets, we present here a simplified version of the original mapping, namely the mapping $\pi$ from the class $PT_{cl}$ of PT nets to the class $PT_lBM_{cl}$ of PT-like B-machines (Definition 1). The structure of PT-like B-machine ($PT_lBM$) is given in (7). In the resulting $PT_lBM$ $M$, $M= \pi(N)$, there will be one state variable $sv_i$ for each place $p_i$ from the PT net $N$ and one operation $op_j$ for each transition $t_j$ from $N$.

***Definition 1.*** Let $N$ be a PT-net $N=(P,T,pre,post,m_0)$, where $P=\{p_1,\ldots, p_k\}$, $T=\{t_1,\ldots, t_n\}$, and $\pi$ be a mapping $\pi: PT_{cl} \rightarrow PT_lBM_{cl}$. Then the image of $N$ under $\pi$ is the PT-like B-machine $M$, $M = \pi(N), M \in PT_lBM_{cl}$, with the structure

```
MACHINE M
VARIABLES sv_1, …, sv_k
INVARIANT sv_1:NATURAL & … & sv_k:NATURAL
INITIALISATION sv_1 := iv_1|| …|| sv_k := iv_k
OPERATIONS                                                  (7)
  op_1 = SELECT PCond_1 THEN Sub_1 END;
  …
  op_n = SELECT PCond_n THEN Sub_n END
END
```

and elements defined as follows:

$$\forall i(1 \le i \le k): iv_i = m_0(p_i) \tag{8}$$

$$\forall j(1 \le j \le n): \text{Pcond}_j = prc(p_1,t_j) \,\&\ldots\&\, prc(p_k,t_j) \tag{9}$$

$$\forall p,t(p \in P, t \in T): prc(p,t)=\begin{cases}\pi_P(p) \ge pre(p,t) & if\ p \in {}^{\bullet}t \\ \texttt{TRUE} & if\ p \notin {}^{\bullet}t\end{cases} \tag{10}$$

$$\forall j(1 \le j \le n): Sub_j = asg(p_1,t_j)||\ldots||asg(p_k,t_j) \tag{11}$$

$$\forall p,t(p \in P, t \in T):$$
$$asg(p,t)=\begin{cases}\pi_P(p) := \pi_P(p) - pre(p,t) + post(p,t) & if\ p \in {}^{\bullet}t \cup t^{\bullet} \\ \texttt{skip} & if\ p \notin {}^{\bullet}t \cup t^{\bullet}\end{cases} \tag{12}$$

The $\pi_P$ is an auxiliary mapping:

$$\pi_P: P \rightarrow \{sv_1,\ldots,sv_k\}, \forall i(1 \le i \le k): \pi_P(p_i) = sv_i \tag{13}$$

The bisimilarity between $N$ and $\pi(N)$ is not hard to see. The construction of the `INITIALISATION` clause in (7) and the formula (8) ensure that the initial value of each $sv_i$ will be the same as $m_0(p_i)$. In $op_j$ the predicate $PCond_j$, specified according to (9) and (10), is similar to the enabling condition (3) and GS $Sub_j$, defined by (11), (12), is equivalent to the new marking computation formula (4) (both for $t_j$). The `SELECT` GS is used because it is not executable when its condition ($PCond_j$ here) is false.

## 4.1 B-Machine for RW Problem

The PT-like B-machine `RWlimited`, transformed from the net in Fig. 1 by the mapping $\pi$, is shown in Fig. 2. In the machine we named variables and

```
MACHINE RWlimited
VARIABLES writersIn, sem, readersIn, freeCap
INVARIANT
  writersIn:NATURAL & sem:NATURAL & readersIn:NATURAL &
  freeCap:NATURAL
INITIALISATION
  writersIn := 0 || sem := 10 || readersIn := 0 || freeCap := 10
OPERATIONS
 wrEnter = SELECT sem >= 10 & freeCap >= 1 THEN
   writersIn:=writersIn+1 || sem:=sem-10 || freeCap:=freeCap-1 END;
 wrLeave = SELECT writersIn >= 1 THEN
   writersIn:=writersIn-1 || sem:=sem+10 || freeCap:=freeCap+1 END;
 rdEnter = SELECT sem >= 1 & freeCap >= 1 THEN
   sem:=sem-1 || readersIn:=readersIn+1  || freeCap:=freeCap-1 END;
 rdLeave = SELECT readersIn >= 1 THEN
   sem:=sem+1 || readersIn:=readersIn-1  || freeCap:=freeCap+1 END
END
```

Figure 2
B-machine RWlimited

operations in the same way that places and transitions are named in Fig. 1. Statements "`skip`" and "`TRUE`" are omitted as for each generalized substitution $S$ it holds that $S||$`skip` $\equiv S$ and for each predicate $P$ that $P\wedge$`TRUE` $\equiv P$. The symbol "`:`" stands for "belongs to" and "`NATURAL`" is the set of natural numbers.

## 4.2 Transformation to Event-B

The concept of events in the Event-B model being executed in a loop while at least one of them is executable is essentially the same as the original concept of Petri net execution: Petri net is also firing transitions until none of them are enabled. And if more transitions are enabled simultaneously, one of them is selected randomly and fired. To adjust our transformation for Event-B we just need to rename operations to events, delete "||" and "$\wedge$" symbols, add names for predicates and actions, replace `SELECT` by `where` and move the initialisation to `events`. We will not define the transformation formally here; we only show how the Event-B version of `RWlimited` looks like (Fig. 3).

```
machine RWlimitedEvB
variables writersIn sem
          readersIn freeCap
invariants
  @inv1 writersIn : NATURAL
  @inv2 sem : NATURAL
  @inv3 readersIn : NATURAL
  @inv4 freeCap : NATURAL
events
  event INITIALISATION
    then
     @act1 writersIn := 0
     @act2 sem := 10
     @act3 readersIn := 0
     @act4 freeCap := 10
  end
  event wrEnter
    where
     @grd1 sem >= 10
     @grd2 freeCap >= 1
    then
     @act1 writersIn :=
           writersIn + 1
     @act2 sem := sem - 10
     @act3 freeCap := freeCap-1
  end
  event wrLeave …
  event rdEnter …
  event rdLeave …
end
```

Figure 3
Event-B machine RWlimitedEvB

As it can be seen, only the `then…end` part of an event is mandatory. The `where…then…end` command is semantically identical to the `SELECT` GS. Only `INITIALISATION` and `wrEnter` events are shown as the rest is created in the same way.

## 5 Application in Software Development

The form of operations introduced in Definition 1 is perfect for an analysis of Petri nets by means of B, for example to prove deadlock freeness [16]. But it is not good for software development as the `SELECT` GS is not feasible when its condition doesn't hold. And only completely feasible operations can be refined. Because of this, when using $PT_1BM$ for software development we replace the form of $op_j$ from (7) by (14) or (15). The form (15) is used if there is a need to report a success of corresponding state change back to a caller of $op_j$.

$$op_j = \texttt{IF } PCond_j \texttt{ THEN } Sub_j \texttt{ END} \tag{14}$$

$$\begin{aligned} ok{<}{-}{-}op_j = \texttt{IF } PCond_j \texttt{ THEN } & Sub_j \texttt{||ok:=TRUE} \\ & \texttt{ELSE ok:=FALSE END} \end{aligned} \tag{15}$$

This replacement doesn't change the bisimilarity relation between markings of *N* and states of *π(N)* as the state of *π(N)* (i.e. the values of its state variables) is changed by $op_j$ only if $PCond_j$ is true. If $PCond_j$ is false, $op_j$ is executed but doesn't change the state of *π(N)* at all.

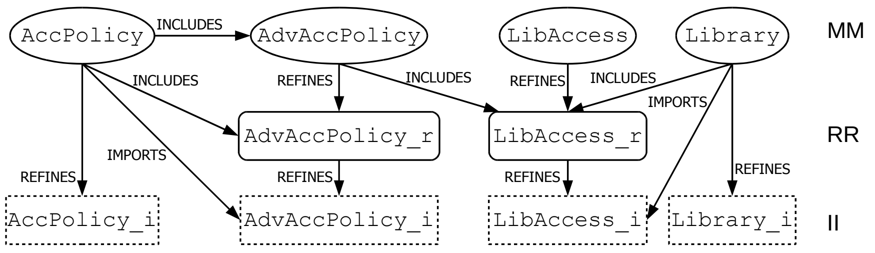

Figure 4
Components of B specification case study

In the rest of this section we present a case study that demonstrates how a B-machine, obtained from a Petri net, can be refined to a more feature-rich form and how this form can be used in other specification components in B. The structure of our specification is shown in Fig. 4. The machine translated from PT net is `AccPolicy`, which is in fact a slightly modified version of `RWlimited`. This is then included (imported) into the `AdvAccPolicy` machine, its refinement `AdvAccPolicy_r` and implementation `AdvAccPolicy_i`, in order to define a more sophisticated access policy component based on the limited RW problem solution. The `Library` machine represents shared contents and `LibAccess` with its refinement and implementation provide access to the shared contents using the policy defined by `AdvAccPolicy`.

The `AccPolicy` (Fig. 5) primarily differs from `RWlimited` in that it uses the form (15) for operations and that the abstract type `NATURAL` is replaced by an implementable type `NAT`. The second modification is an introduction of the parameter `cap`, which represents the capacity of the library and replaces the value "`10`" from `RWlimited`. This makes the machine more usable without affecting any of its properties. Finally, the third change is an addition of formulas equivalent to (5) and (6) to its invariant (in italic in Fig. 5). This addition was necessary for proving that variables of the machine will not exceed the limit of the type `NAT`. Fig. 5 doesn't show bodies of `rdEnter` and `rdLeave`, as they are similar to those of the previous operations.

```
MACHINE AccPolicy (cap)
CONSTRAINTS cap:NAT & cap>0
VARIABLES writersIn, sem, readersIn, freeCap
INVARIANT
  writersIn:NAT & sem:NAT & readersIn:NAT & freeCap:NAT &
  writersIn+freeCap+readersIn=cap & cap*writersIn+sem+readersIn=cap
INITIALISATION
  writersIn := 0 || sem := cap || readersIn := 0 || freeCap := cap

OPERATIONS
 ok<--wrEnter =
   IF sem >= cap & freeCap >= 1 THEN
```

```
    writersIn:= writersIn+1 || sem:= sem-cap || freeCap:=freeCap-1
    || ok:=TRUE
   ELSE ok:=FALSE END;
 ok<--wrLeave =
   IF writersIn >= 1 THEN
    writersIn:= writersIn-1 || sem:= sem+cap || freeCap:= freeCap+1
    || ok:=TRUE
   ELSE ok:=FALSE END;
 ok<--rdEnter = ... END;
 ok<--rdLeave = ... END
END
```

Figure 5

B-machine AccPolicy

The crucial difference between `AccPolicy` and `AdvAccPolicy` (and corresponding refined components) is that the latter contain variables `reading` and `writing`. These are used to register processes that are currently editing or reading the shared contents. Both variables are subsets of the set `PROCESSES`, which represents all processes that could possibly access the contents. One member of `PROCESSES`, stored in the `noPr` variable, is reserved for the null process. Introduction of `reading` and `writing` allowed us to check by PObs whether our advanced procedure really obeys the access policy defined by the original PT net (and `AccPolicy`): In `AdvAccPolicy` (Fig. 6) we prove that it is impossible to write and read at the sametime, and in `AdvAccPolicy_r` (Fig. 7) we show that the number of writing and reading processes is always the same as in `AccPolicy`. Related parts of their invariants are written in italic. The operation `reqRAcc` corresponds to `rdEnter`, `reqWAcc` to `wrEnter` and `leave` unites `rdLeave` and `wrLeave`. They call the corresponding operations from `AccPolicy`. The proper order of calling is established in the refinement `AdvAccPolicy_r` since it is impossible to use the sequential composition in B-machines. The `scs` output parameter indicate whether a request to access or leave the shared contents was successful and `pr` holds assigned process id. The null process (`noPr`) is returned if the access is not granted. There are three extra operations, `canRead` and `canWrite` to return process status and `getNullPr` to get the value used as the null process. The last operation is necessary for the final stage of development as it is forbidden to read variables directly in implementations.

```
MACHINE AdvAccPolicy(prCap)
CONSTRAINTS prCap:NAT & prCap>0
INCLUDES AccPolicy(prCap)
SETS PROCESSES
VARIABLES reading, writing, noPr
INVARIANT reading <: PROCESSES & writing <: PROCESSES &
          noPr : PROCESSES & reading /\ writing = {} &
          {noPr} /\ reading = {} & {noPr} /\ writing = {}
INITIALISATION reading:= {} || writing:= {} || noPr::PROCESSES
```

```
OPERATIONS
 pr,scs<--reqRAcc=
  IF PROCESSES-(reading\/writing\/{noPr}) /= {} THEN
   ANY pp WHERE pp:PROCESSES-(reading\/writing\/{noPr}) THEN
     CHOICE reading := reading \/ {pp} || pr:=pp OR pr:=noPr END
     || scs<--rdEnter
   END
  ELSE scs:=FALSE || pr:=noPr END;
 pr,scs<--reqWAcc=
  IF PROCESSES-(reading\/writing\/{noPr}) /= {} THEN
   ANY pp WHERE pp:PROCESSES-(reading\/writing\/{noPr}) THEN
     CHOICE writing := writing \/ {pp} || pr:=pp OR pr:=noPr END
     || scs<--wrEnter
   END
  ELSE scs:=FALSE || pr:=noPr END;
 scs<--leave(pr)=
  PRE pr: reading\/writing THEN
   IF pr:reading THEN
     CHOICE reading := reading - {pr} OR skip END || scs<--rdLeave
   ELSE
     CHOICE writing := writing - {pr} OR skip END || scs<--wrLeave
   END
  END;
 yes<--canRead(pr)= PRE pr:PROCESSES THEN
    IF pr:reading THEN yes:=TRUE ELSE yes:=FALSE END
 END;
 yes<--canWrite(pr)= PRE pr:PROCESSES THEN
    IF pr:writing THEN yes:=TRUE ELSE yes:=FALSE END
 END;
 npr <-- getNullPr = BEGIN npr:=noPr END
END
```

Figure 6

B-machine AdvAccPolicy

In Fig. 6 and the following ones the symbol "`<:`"means "is subset or equal", "`\/`" is the set union, "`/\`" the set intersection, "`{}`" the empty set set and "`{x}`" is a set with `x` as its only member. The symbol "`/=`" stands for "not equal" and "`xx::SS`" is a special kind of the `ANY` GS with the meaning "assign any arbitrary selected value from a set `SS` to a variable `xx`".

```
REFINEMENT AdvAccPolicy_r(prCap)
REFINES AdvAccPolicy
INCLUDES AccPolicy(prCap)
VARIABLES reading, writing, noPr
INVARIANT card(writing)=writersIn & card(reading)=readersIn
INITIALISATION reading:= {}; writing:= {}; noPr::PROCESSES

OPERATIONS
 pr,scs<--reqRAcc=
  IF PROCESSES-(reading\/writing\/{noPr}) /= {} THEN
   VAR acd,pp IN
```

```
      acd <--rdEnter;
      pp::PROCESSES-(reading\/writing\/{noPr});
      IF acd=TRUE THEN
        reading := reading \/ {pp}; pr:=pp; scs:=TRUE
      ELSE scs:=FALSE; pr:=noPr END
    END
   ELSE scs:=FALSE; pr:=noPr END;
 pr,scs<--reqWAcc=
   IF PROCESSES-(reading\/writing\/{noPr}) /= {} THEN
    VAR acd,pp IN
      acd <--wrEnter;
      pp::PROCESSES-(reading\/writing\/{noPr});
      IF acd=TRUE THEN
       writing := writing \/ {pp}; pr:=pp; scs:=TRUE
      ELSE scs:=FALSE; pr:=noPr END
    END
   ELSE scs:=FALSE; pr:=noPr END;
 scs<--leave(pr)=
   IF pr: reading\/writing THEN
     VAR acd IN
       IF pr:reading THEN
         acd <--rdLeave;
         IF acd=TRUE THEN reading := reading - {pr}; scs:=TRUE
          ELSE scs:=FALSE END
       ELSE
         acd <--wrLeave;
         IF acd=TRUE THEN writing := writing - {pr}; scs:=TRUE
          ELSE scs:=FALSE END
       END
     END
   ELSE scs:=FALSE END;
 yes<--canRead(pr)= IF pr:PROCESSES THEN
    IF pr:reading THEN yes:=TRUE ELSE yes:=FALSE END
 ELSE yes:=FALSE END;
 yes<--canWrite(pr)= IF pr:PROCESSES THEN
    IF pr:writing THEN yes:=TRUE ELSE yes:=FALSE END
 ELSE yes:=FALSE END;
 npr <-- getNullPr = BEGIN npr:=noPr END
END
```

Figure 7

Refinement AdvAccPolicy_r

```
MACHINE Library
VARIABLES contents
INVARIANT contents:NAT
INITIALISATION contents:=0
OPERATIONS
 lcnt <-- read = lcnt:=contents;
 write(ncnt) = PRE ncnt:NAT THEN contents:=ncnt END
END
```

Figure 8

B-machine Library

The machine `Library` (Fig. 8) represents shared contents that is accessed by processes and defines operations over it. For the sake of simplicity, the content is only a natural number here.

An access to the shared contents is provided by the `LibAccess` (Fig. 9) component and its refinement. In fact, `LibAccess` defines only heads of operations and the type `RETCODE`, while the real functionality is encoded in its refinement `LibAccess_r` (Fig. 10). The reason why we have to use the refinement is the restriction on the use of "`;`", again.

```
MACHINE LibAccess(lCap)
CONSTRAINTS lCap:NAT & lCap>0
SETS RETCODE={ok,failEnter, failLeave, failWrite}
OPERATIONS
 cnt, rc<--libRead= BEGIN rc::RETCODE || cnt::NAT END;
 rc<--libWrite(cnt)=PRE cnt:NAT THEN rc::RETCODE END
END
```

Figure 9
B-machine LibAccess

To read the contents, one has to call the operation `libRead`, which first checks whether it is possible to read by calling `reqRAcc` from `AdvAccPolicy` then reads (by calling `read` from `Library`) and, finally, calls `leave` to announce that the reading is over. For editing the `libWrite` operation is used, which works in the similar way.

We decided to not describe the four implementation components of our case study in this paper as they are similar to the corresponding refinements or machines.

```
REFINEMENT LibAccess_r(lCap)
REFINES LibAccess
INCLUDES AdvAccPolicy(lCap), Library

OPERATIONS
 cnt,rc<--libRead=
   VAR acd, prId IN
     prId,acd <--reqRAcc;
     IF acd=TRUE & prId /=noPr THEN
       cnt<--read; acd <--leave(prId);
       IF acd=TRUE THEN rc:=ok ELSE rc:=failLeave END
     ELSE cnt:=0; rc:=failEnter END
 END;

 rc<--libWrite(cnt)=
   IF cnt:NAT THEN
     VAR acd, prId IN
       prId,acd <--reqWAcc;
       IF acd=TRUE & prId /=noPr THEN
         write(cnt); acd <--leave(prId);
```

```
        IF acd=TRUE THEN rc:=ok ELSE rc:=failLeave END
      ELSE rc:=failEnter END
     END
ELSE rc:=failWrite END
END
```

Figure 10

Refinement LibAccess_r

## 6 Utilization in Concurrent Environment

While the verification mechanisms of B proved to be sufficient to ensure that properties of the design (`AccPolicy` machine) are maintained in components refining and directly including it, we still cannot call the resulting implementation safe for use in a concurrent environment. B doesn't take concurrency into account, so to improve the situation, extensions to both its language and tools are necessary. The B-language can be extended by annotations allowing one to label operations that cannot be run in parallel at all or within some group of operations. Then modified compilers for B will translate these annotations to equivalent constructs of target programming languages. However, one critical question remains open: *Can the process of annotating of operations and of verifying their consistency be automated?*

The use of machines translated from Petri nets provides a partial answer here: Assuming that all concurrency issues are treated in machines transformed by $\pi$ and that these machines are separately implemented (like `AccPolicy` in `AccPolicy_i`), we can automatically annotate operations in them and mark all operations that directly or indirectly call their operations as candidates for concurrent execution. However, it is very probable that the final decision about the calling operations will require certain amount of manual checking.

The automatic annotating of machines translated from PN can be easily implemented, obeying the following rule: operations created from transitions with common pre-places cannot be run at once. This is because there is a risk that their enabling conditions (*PCond* in (7)) will be evaluated at once and, as they read some common variables, will lead to faulty execution of their bodies. In machines in Fig. 2 and Fig. 5 `wrEnter` and `rdEnter` are such operations as their counterparts in Fig. 1 have common pre-places *sem* and *freeCap*. Separate and careful development of these machines is critical: we can introduce new variables and add new functionality to its refinements and implementation, but what was defined in the machine must stay intact.

Regarding the calling operations, the question is whether they call those originated from Petri nets properly. In our case study, the calling ones can be found in `AdvAccPolicy(_r)` (the first three ones) and in `LibAccess_r` (both). In the case of `AdvAccPolicy_r`, its invariants and corresponding PObs help to resolve the situation and they can be allowed to run concurrently. Operations from `LibAccess(_r)` are also the calling ones as they call the first three from `AdvAccPolicy`, but invariants and PObs are of no help here. This is because operations from `AdvAccPolicy` are called in sequence in `libRead` and `libWrite` and PObs only check states before and after an operation execution. Again, the situation can be improved by introducing annotations to define order of execution of operations in machines obtained from PN and a procedure that will check whether this order is maintained within every calling operation.

## 7 Related Work

The problem of Petri nets and B-Method integration attracted other researchers as well, but, to our knowledge, all of these works have been published after the initial version of our transformation [13] and approach the problem from a more or less different perspective.

The work [3] presents an encoding of PT nets and high-level PN (HLPN, tokens have values assigned in these types of PN) to the Event-B language. Each Petri net is represented by a specification consisting of two machines. The first machine contains constants, sets and variables that define the concrete Petri net. The second one contains one event for transition firing and in the case of HLPN also events for actions associated with places and transitions. The second machine is identical for all PT nets. The author uses the Atelier B version of Event-B syntax, which is much closer to the classical B-language than to the "official" version, presented in [2]. The essential difference between our approach and [3] is that we translate each transition to a separate operation (event). This is more natural and usable for software development. The author of [3] claims that his primary motivation is analysis; however our practical experience shows that the data representation chosen in [3] is usually more difficult for the Atelier B prover to handle than the one used in our approach.

In [11, 12] a mapping of a subset of the SYNTESIS scripting language, which is similar to HLPN, is presented. The target specification is the Refinement component of B-Method. In principle, the approach is close to ours: places are mapped to variables and transitions to operations. What differs is that variables in [11, 12] are sets and structure of operations is more complicated as high-level PN have individualised tokens. The purpose of the mapping is an analysis of scripts in SYNTESIS by means of B-Method. A similar transformation is used in a railway safety-related case study in [7] to translate a simple Coloured Petri net (a kind of

HLPN) specification to the Event-B language with the Atelier B syntax. The B specification in [7] uses a special machine that implements multi-sets and the purpose of this transformation is a further development of the specified system. The mapping of PN, defined in [13, 15] and used in this paper, can be quite easily modified for HLPN by adopting principles of these two approaches. However, we found the PT nets and other PN types with undistinguishable tokens (so-called low-level PN) more suitable for the role of the most abstract specification of a development. They provide analytical methods that are only hardly usable for HLPN (e.g. derivation of invariants) and an additional functionality can be added later, on the side of B-Method. On the other hand, we are aware that utilization of HLPN can lead to significant reduction in size of PN models and allow parameterisation of models that is impossible for low-level PN.

There have been also researches that combined B with other formal methods for specification of concurrent systems, primarily with the Communicating Sequential Processes (CSP) [6]. The most significant approaches are csp2B [4] and CSP||B. Both support certain subsets of CSP. The csp2B provides a method of transformation of CSP specifications to B-language. The transformation translates implicit states of CSP processes to variables and process events to operations and is supported by the csp2B tool. The CSP||B is a method that combines specifications in CSP and B-language in such a way that CSP controllers manage concurrently running B-machines. CSP||B has been introduced in [21] and further developed in subsequent works, such as [22] or [23] where it was adjusted for Event-B. Both approaches also deal formally with refinement of translated or combined specifications, so their results will be considered when designing mechanisms for maintaining some consistency aspects of B-machines created by our transformation.

## Conclusions

The approach to implementation of PN specifications, shown here, looks promising and brings qualities that B doesn't provide out of the box, namely some control over concurrency aspects. PN have an easy-to-understand graphical form, so they may be more attractive for developers to use than text-based methods for concurrent systems, such as CSP. By the mapping $\pi$ we have been able to properly transfer a design created and verified on the Petri nets' side to B-Method. However, as B-Method has been designed solely for development of sequential systems, its verification system is only of limited use when checking concurrency-related aspects of translated PN and components that access or refine them. Possible solution of this problem is outlined in section 6 and its realization should be one of the primary tasks for future research and development. The implementation platform will most probably be the BKPI compiler [17], developed at the home institution of authors, which can parse specifications in B-language and translate implementation components to Java and C#. Lessons learned by other teams when integrating other formal methods for concurrent systems with B (e.g. csp2B and CSP||B) will certainly be considered here. On the

PN side the demonstrated approach can also benefit from cooperation with other formalisms, for example with the linear logic [20] or artificial intelligence concepts such as cognitive maps, including their fuzzy variants [24].

Considering the strong relation between computational concepts of Petri nets and Event-B, it will also be worth to explore possibilities of incorporation of transformed PN to Event-B models. It should be also useful to integrate Petri nets in similar way to other industrially-used formal methods for software development, such as VDM or language Perfect. For the reasons mentioned in the previous section we also plan to extend the mapping $\pi$ to support core features of HLPN. It is relatively easy to specify such an extension and prove its correctness, provided that the language used to handle tokens in HLPN is a subset of the B-language.

The transformation of the Evaluative PN and PT nets has been already implemented in the experimental mFDTE/PNtool2 software, which is available by request from the authors.

**Acknowledgement**

This work was supported by KEGA grant project No. 050TUKE-4/2012: “Application of Virtual Reality Technologies in Teaching Formal Methods”.

## References

[1] J. R. Abrial: The B-Book: Assigning Programs to Meaning, Cambridge University Press, Cambridge, 1996

[2] J. R. Abrial: Modeling in Event-B: System and Software Engineering, Cambridge University Press, Cambridge, 2010

[3] J. Ch. Attiogbé: Semantic Embedding of Petri Nets into Event B, In: International Workshop on Integration of Model-based Methods and Tools IM FMT'09 at IFM'09 Conference, Düsseldorf Germany, February 2009, available from: http://pagesperso.lina.univ-nantes.fr/info/perso/permanents/attiogbe/B_Petri/

[4] M. J. Butler: csp2B: A Practical Approach to Combining CSP and B., Formal Aspects of Computing, Vol. 12, 2000, pp. 182-196

[5] H. Ehrig, W. Reisig, G. Rozenberg, H. Weber (Eds.): Petri Net Technology for Communication-based Systems, LNCS vol. 2472, Springer, 2003

[6] C. A. R. Hoare: Communicating Sequential Processes, Prentice Hall, ISBN 0-13-153289-8, 1985, available from: http://www.usingcsp.com/

[7] F. Defossez, P. Bon, S. Collart-Dutilleu: Taking Advantage of some Complementary Modelling Methods to Meet Critical System Requirement Specifications, In: Safety and Security in Railway Engineering, WIT Press, 2010, pp. 119-128

[8] J. Desel, W. Reisig: Place/Transition Petri Nets. In: W. Reisig and G. Rozenberg (eds.) Petri Nets, LNCS Vol. 1491, Springer, 1998, pp. 122-173

[9] E. W. Dijkstra: A Discipline of Programming, Prentice Hall, Englewood Cliffs, ISBN 0-13-215871-X, 1976

[10] Š. Hudák: Reachability Analysis of Systems Based on Petri Nets, Elfa, Košice, 1999

[11] L. A. Kalinichenko, S. A. Stupnikov, N. A. Zemtsov: Extensible Canonical Process Model Synthesis Applying Formal Interpretation. In: Proc. of the East-European Conference ADBIS'05, Talin, Estonia, September 2005, pp. 183-198

[12] L. A. Kalinichenko, S. A. Stupnikov, N. A. Zemtsov: Synthesis of the Canonical Models for the Integration of Heterogeneous Information Resources. M.: IPI RAN, 2005, 87 pp (In Russian)

[13] Š. Korečko, Š. Hudák: Implementing Petri nets via B-Method. In: Proc. of 6th International Scientific Conference Electronic Computer and Informatics, ECI 2004, September 2004, pp. 103-110

[14] Š. Korečko, Š. Hudák: S. Šimoňák: Analysis of B-machine based on Petri Nets, Proceedings of CSE 2008 International Scientific Conference on Computer Science and Engineering, September 2008, pp. 24-33

[15] Š. Korečko: From Petri Nets to B-Method, Technical report DCI 1/2009, Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, 2009, available from: http://hornad.fei.tuke.sk/~korecko/pblctns/trEvPN_B.pdf

[16] Š. Korečko, B. Sobota: Building Parallel Raytracing Simulation Model with Petri Nets and B-Method, In: Proc. of the 7th EUROSIM Congress on Modelling and Simulation, Prague, Czech Republic, 2010, ISBN 978-80-01-04589-3, 7pp

[17] Š. Korečko, M. Dancák: Some Aspects of BKPI B Language Compiler Design, Egyptian Computer Science Journal, Vol. 35, No. 3, 2011, pp. 33-43

[18] L. Madarász, J. Vaščák, R. Andoga, T. Karoľ: Decision Making, Complexity and Uncertainty: Theory and Practice, elfa s.r.o., Košice, 2010, ISBN 978-80-8086-142-1 (in Slovak)

[19] L. Madarász, J. Živčák (Eds.): Aspects of Computational Intelligence: Theory and Applications, Topics in Intelligent Engineering and Informatics, Vol. 2, Springer, 2013

[20] D. Mihályi, V. Novitzká, V. Slodičák: From Petri Nets to Linear Logic, In: Proc. of CSE'2008, Stará Lesná, September 2008, pp. 48-56

[21] S. Schneider, H. Treharne: How to Drive a B Machine. In: Proc. of ZB 2000, LNCS Vol. 1878, Springer, 2000, pp. 188-208

[22] S. Schneider, H. Treharne: Verifying Controlled Components. In: Proc. of IFM 2004, LNCS Vol. 2999, Springer, 2004, pp. 87-107

[23] S. Schneider, H. Treharne, H. Wehrheim: A CSP Approach to Control in Event-B. In: Proc. of IFM 2010, LNCS Vol. 6396, Springer 2010, pp. 260-274

[24] J. Vaščák, L. Madarász: Adaptation of Fuzzy Cognitive Maps – a Comparison Study, Acta Polytechnica Hungarica, Vol. 7, No. 3, 2010, pp. 109-122