ABCD: A user-friendly language for formal modelling and analysis

Franck Pommereau

ABCD: a User-Friendly Language for Formal Modelling and Analysis

Franck Pommereau

IBISC, University of Évry, 23 boulevard de France, 91037 Évry Cedex, France
franck.pommereau@ibisc.univ-evry.fr

Abstract. This paper presents an algebra of coloured Petri nets called the *Asynchronous Box Calculus with Data*, or ABCD for short. ABCD allows to model complex systems using a user-friendly and high-level syntax. In particular, parts of the model can be directly programmed in Python [21], which allows to embed complex computation and data values within a model. A compiler for ABCD is shipped with the toolkit SNAKES [16, 18] and ABCD has been used for years, which is quickly surveyed. This paper is the first complete and formal presentation of the language and its semantics. It also presents uses cases of ABCD for the modelling and analysis of various systems.

Keywords: Formal modelling, high-level models, Petri nets semantics.

1 Introduction

The *Asynchronous Box Calculus with Data*, or ABCD for short, is an algebra of Petri net, *i.e.*, a process algebra with a Petri net semantics [17]. With respect to the other algebras of the family like the *Petri Box Calculus* [2,3], ABCD is a high-level modelling language: its semantics is based on high-level Python-coloured Petri nets, that can be composed and transformed using various operations like the terms of a process algebra. With respect to other members of the family, ABCD is asynchronous, *i.e.*, does not have the transition synchronisation operation *à la* CCS; however, it could be added easily if needed. An important aspect of ABCD is that it uses Python as a concrete programming language to provide data, expressions and computation. The syntax of ABCD is inspired from that of Python but separated even if it actually embeds the fragment for expressions (Python expressions may be used in ABCD). A compiler is distributed within the SNAKES toolkit [16, 18] and allows to build a Petri net from ABCD source code.

This paper provides the first complete and formal definition of ABCD, *i.e.*, its syntax and semantics. Only informal introductions though examples had been published so far, the most complete of which being [17, pages 30–33]. Other contributions of this paper are: a definition of Python-coloured Petri nets with read/fill/flush-arcs, a presentation of the ABCD compiler and simulator, and a short survey of ABCD use cases for research and teaching to showcase its usability.

The next section presents the syntax of ABCD together with its intuitive semantics. The formal semantics is defined in section 3 as a translation of ABCD

terms into Petri nets. Section 5 describes the compiler and the interactive simulator. Section 4 presents existing applications of ABCD. Finally, the paper ends on a conclusion with perspectives. This paper assumes no particular knowledge about Python, however, a good Python tutorial is available online [20].

2 Modelling with ABCD

Figure 1 shows an ABCD model of the dining philosophers problem with four philosophers. We use it to introduce the main concepts of ABCD. Line 1, a *buffer* called forks is *declared*, it is a typed container that will initially hold the four integers 0 to 3 (whose type is int in Python), each of which models a fork identifier. Line 3–4, a *sub-process* called philo is declared. It is parametrised by the two values of the left and right fork that a particular philosopher has to use. Line 4 is the *process expression* for a philosopher that consists of three *atomic actions* enclosed between square brackets and connected with *control-flow operators*. The first atomic action is "[forks−(left), forks−(right)]" and specifies that value left is consumed (thus the "−") from buffer forks and, at the same time, value right is also consumed in the same buffer. This atomic action is composed sequentially (operator ";") with another atomic action that produces (thus the "+") the two same values into the same buffer. Then, the sequence itself is composed in a loop (operator "∗") with atomic action "[False]", which means that the sequence can be arbitrarily repeated until "[False]" is executed to finish the loop (and here the whole process). However, "[False]" is an atomic actions that can never be executed so we actually have an infinite loop. Finally, Line 6 defines the *main process* of the model that composes in parallel (operator "|") four *instances* of sub-process philo with parameters chosen to arrange the philosophers on a circle.

More generally, an ABCD model consists of a *process* description comprising optional *declarations* (in particular sub-processes and resources) and a main *process expression*. Sub-process can themselves include declarations that are local to them, *i.e.*, cannot be used from outside the sub-process. The full grammar of ABCD is given in Figure 2 and is commented in the rest of the section.

```
1  buffer forks : int = 0, 1, 2, 3
2
3  net philo (left, right):
4      ([forks−(left), forks−(right)] ; [forks+(left), forks+(right)]) ∗ [False]
5
6  philo(0, 1) | philo(1, 2) | philo(2, 3) | philo(3, 0)
```

Fig. 1. A model of four dining philosophers where a generic philosopher is specified as a parametrised sub-process.

⟨*spec*⟩	::=	⟨*global*⟩ ↲ ⟨*spec*⟩	global declaration
	|	⟨*local*⟩ ↲ ⟨*spec*⟩	local declaration
	|	⟨*proc*⟩	process
⟨*global*⟩	::=	import_stmt	Python import statement
	|	**symbol** name { "," name}	symbols declaration
	|	**typedef** name ":" ⟨*type*⟩	type declaration
	|	**const** name "=" expr	constant declaration
⟨*local*⟩	::=	**buffer** name ":" ⟨*type*⟩ "=" expr	buffer declaration
	|	**net** name "(" ⟨*params*⟩? "):" ↳ ⟨*sub*⟩ ↰	sub-process declaration
⟨*sub*⟩	::=	⟨*local*⟩ ↲ ⟨*sub*⟩	local declarations
	|	⟨*proc*⟩	behaviour specification
⟨*type*⟩	::=	name	native Python type
	|	⟨*type*⟩ "&" ⟨*type*⟩	intersection
	|	⟨*type*⟩ "|" ⟨*type*⟩	union
	|	⟨*type*⟩ "*" ⟨*type*⟩	cross-product
	|	**enum** "(" expr { "," expr} ")"	enumerated type
	|	⟨*cont*⟩ "(" ⟨*type*⟩ { "," ⟨*type*⟩} ")"	container type
⟨*cont*⟩	::=	name "(" ⟨*type*⟩ ")"	collection type
	|	"dict(" ⟨*type*⟩ "," ⟨*type*⟩ ")"	dictionary type
⟨*params*⟩	::=	name { "," ⟨*params*⟩}	value parameter
	|	name ":" **buffer** { "," ⟨*params*⟩}	buffer parameter
⟨*proc*⟩	::=	⟨*proc*⟩ "|" ⟨*proc*⟩	parallel composition
	|	⟨*proc*⟩ ";" ⟨*proc*⟩	sequential composition
	|	⟨*proc*⟩ "+" ⟨*proc*⟩	choice composition
	|	⟨*proc*⟩ "*" ⟨*proc*⟩	iteration
	|	"(" ⟨*proc*⟩ ")"	nested process
	|	name "(" ⟨*args*⟩? ")"	anonymous net instance
	|	name "::" name "(" ⟨*args*⟩? ")"	named net instance
	|	"[True]"	always possible action
	|	"[False]"	always blocking action
	|	"[" ⟨*access*⟩ { "," ⟨*access*⟩} "]"	unconditional action
	|	"[" ⟨*access*⟩ { "," ⟨*access*⟩} **if** expr "]"	conditional action
⟨*access*⟩	::=	name "+(" expr ")"	production
	|	name "-(" expr ")"	consumption
	|	name "?(" expr ")"	test
	|	name "<>(" expr "=" expr ")"	swap
	|	name ">>(" name ")"	flush
	|	name "<<(" expr ")"	fill
⟨*args*⟩	::=	expr { "," ⟨*args*⟩}	arguments

Fig. 2. The syntax of ABCD, where: ⟨···⟩ denotes non-terminals, ↲ denotes a newline; import_stmt is a Python import statement; **bold** face denotes keywords; name is an arbitrary Python name (*i.e.*, an identifier); expr is an arbitrary Python expression; "..." denotes literals; {···} denotes parts that can be repeated zero or more times; ...? denotes parts that can be omitted; ↳ denotes a newline followed by an indented block; ↰ denotes a newline at the end of an indented block.

2.1 Global Declarations

These are declarations that are only allowed at the top level of the specification, *i.e.*, not within a sub-process.

A Python *import statement* [20, sec. 6] allows to make visible names defined in an external Python module. For instance, using "**import** math" allows to use function "math.factorial" in expressions; statement "**from** math **import** ∗" gives a direct access to function "factorial" and others, without the prefix "math.".

Defining *symbols* is a way to create unique named values in a model. For instance, "**symbol** OPEN, CLOSE" defines names "OPEN" and "CLOSE" that have opaque values, distinct from every other existing values.

It is also possible to define *new types*, *i.e.*, give a name to a type, using the "**typedef**" declaration. Types in ABCD are sets of values and can be specified using a rich type algebra, see ⟨*type*⟩ in the grammar. Basically, a type is a Python class, for instance "int" or "str", and corresponds to the set of all the objects of this class. Two classes are worth mentioning: "object" is Python's universal class, *i.e.*, any value an "object" instance; "BlackToken" is a class with a single value "dot" that implements the Petri nets black token. Building more complex types is possible using union, intersection and cross-product of types; two more constructions deserve explanation. Enumerated types are defined as sets of values, for instance "**enum**(1, 2, `"hello"`)" defines a type with only the three enumerated values. Container types are Python collections whose content is constrained, for instance "tuple(int)" denotes the set of tuples of integers; "list" and "set" are the two other supported simple containers. Finally, Python dictionaries (*i.e.*, mappings) are also supported containers, for instance "dict(int, str)" denotes the set of dictionaries whose keys are integers and values are strings.

Other global declarations are *constants*. For instance, "**const** foo = 42" defines the name "foo" whose value is 42. Contrasting with symbols, constants have known values that can be exploited in the model, for instance, "foo+2" is a correct expression when "foo" has been declared as above.

2.2 Buffers

Resources in ABCD are stored in *buffers*, *i.e.*, unbounded and unordered data containers that can be accessed from the process that declares the buffer as well as from any of its sub-processes. In the semantics, they correspond to (coloured) places. A buffer is declared using keyword **buffer** and is given:

- a *name* that is an identifier that will be used in process expressions to access the buffer;
- a *type* that restricts the values allowed in the buffer;
- an *initial content* given as an expression that is interpreted as a series of values initially stored in by the buffer.

The following ABCD code shows the declarations of two buffers:

```
1 buffer foo : int = ()
2 buffer bar : str = "hello", "world"
```

The first line declares a buffer named "foo", whose type is "int" the set of integers and whose initial content is empty since "()" denotes the empty tuple in Python. The second line declares a buffer named "bar" whose type is "str" the set of strings and that initially contains the two strings `"hello"` and `"world"`.

2.3 Sub-Processes

An important feature of ABCD is to provide *parametrised sub-process* that can be instantiated later on. A sub-process is declared using keyword "**net**" followed by a name and a list of parameters. For instance "**net** sub (a, b):↓" introduces a sub-process called sub that is parametrised by two values "a" and "b". If a parameter needs to be a buffer, this must be explicit, like in "**net** sub (a, b: **buffer**):↓" where "b" is now a buffer parameter. The specification of a sub-process is given in an indented block after the first line, it comprises local declarations and a process expression that specifies the behaviour. We will see later on how sub-processes are instantiated within a process expression.

2.4 Process Expressions

These specify the *behaviour of a (sub-)process*. The most basic behaviours are *atomic actions* and are enclosed in square brackets "[···]". In the semantics, atomic actions correspond to Petri nets transitions. The simplest ones are "[True]" that can always be executed and has no effect on buffers, and "[False]" that can never be executed and is always blocking. We have seen and example using "[False]" above and will see one with "[True]" later on.

More complex actions are formed as lists of *buffer accesses* and an *optional execution condition*. Each buffer access is given as a buffer name, a symbol to specify the access type and an expression to specify the data accessed. For instance, "buf+(2∗n)" specifies that the atomic action, when executed, creates in buffer "buf" a value that is the result of evaluation expression "2∗n". "buf−(x)" allows to consume from the buffer a value that is bound to variable "x". An actual value may be used instead of variable "x" to consume a known value. It is also possible to use patterns, like in "buf−(x,y,0)" that consumes a triple whose first and second elements are bound to x and y respectively and whose third element must be 0. Currently, patterns may only be nested tuples, allowing to decompose the consumed values. Note that it is not possible to specify an arbitrary expression to be consumed because this would require to solve an arbitrary equation, which is not possible in general. Two other access types are: test "?" that behaves like consumption except that is does not actually consume the value; and swap that is a shorthand for consumption plus production, for instance, "buf<>(x=x+1)" can be replaced by "buf−(x), buf+(x+1)".

Then come two accesses handling multiple tokens. First, the flush "buf>>(v)" empties buffer "buf" and binds the multiset of its content to variable "v". Note that this is possible even if the buffer is empty in which case "v" is bound to the empty multiset, this gives a possible implementation of a test for zero using "[..., buf>>(v) **if** not v]" where "v" used as a Boolean expression is true if and

only if it is not empty. Next, the fill operation produces values into a buffer: using "buf<<(expr)", expression "expr" is evaluated and iterated over as a collection so that each of its value is added to "buf". (Note that this does not overwrite existing values in the buffer, it just adds new content.) For instance, one may increment all the values within a buffer using a single atomic action: "[buf>>(v), buf<<(x+1 **for** x **in** v)]" where expression "x+1 **for** x **in** v" is a Python *comprehension* [20, sec. 5.1.4].

A *guard* may be specified at the end of an atomic action, using keyword "**if**" followed by an expression. This allows the execution only if the expression evaluates to true. The scope of the variables used within an action is limited to this action. Variables are bound thanks to buffer accesses that consume or test values in buffers (*i.e.*, "−", "?", ">>" and the left-hand side of "<>"), free variables in other accesses and in the guard are forbidden.

Actions can be composed using four *control-flow operators* and parentheses: ";" is the sequential composition allowing to execute first its left-hand side process and then its right-hand side process; "|" is the parallel composition allowing to execute two processes concurrently; "+" is the choice composition allowing to execute only one of the two processes it composes; "∗" is the iteration allowing to execute repeatedly its left-hand side process (including zero times) followed by exactly one execution of its right-hand side process. Action "[False]" is often used at the right-hand side of an iteration to create an infinite loop, like in "[buf−(x), buf−(y), buf+(x) **if** y % x == 0] ∗ [False]" which implements a sieve of Eratosthenes.

Finally, a process may also include *instances* of previously declared sub-processes (*i.e.*, nets). A term composed of the net name followed by a list of effective arguments is replaced by the whole sub-process in which all the parameters have been substituted by the arguments. Such an instance may be named to simplify the access to its places in the Petri net semantics. Imagine for example a buffer "mybuf" declared inside a sub-process "mynet" parametrised by three values, when building instance "mynet(1,2,3)", the resulting copy of "mybuf" is normally called "mynet(1,2,3).mybuff". By using a named instance, one can simplify this, for example, instance "foo::mynet(1,2,3)" gives rise to buffer "foo.mybuff" and a place with the same name in the Petri net semantics.

Note that within a process expression, spaces and newlines are not significant, only indentation must be respected. And within a process nested in parentheses, even indentation is not significant anymore. This allows to choose clearer presentation for process expressions, as for instance in the example of Section 4.1.

3 Petri Net Semantics of ABCD

We define now the Petri nets semantics of ABCD, and first the variant of Petri nets we use: an algebra of Python-coloured Petri nets extended with read/fill/flush-arcs, and supporting control-flow compositions. This class of Petri nets corresponds to *Petri nets with control-flow* as defined in [17, sec. 2.1 to 2.3] in

which the originally abstract colour domain has been concretized as the Python language, and with an extension to support read/fill/flush-arcs.

3.1 Python-Coloured Petri Nets

A Python-coloured Petri net (PCPN) involves values, variables and expressions. These objects are defined by the Python programming language and, because we do not want to defined them here, abstracted away as follows:

- $\mathbb{D}$ is the set of *data* values, *i.e.*, all the possible Python objects, including "dot" that implements "$\bullet$";
- $\mathbb{D}_\perp \stackrel{\text{df}}{=} \mathbb{D} \uplus \{\perp\}$ is the set of data enriched with a special "undefined" value;
- $\mathbb{V}$ is the set of *variables*, *i.e.*, all the possible Python identifiers;
- $\mathbb{E}$ is the set of *expressions*, involving values, variables and appropriate operators according the syntax of Python. Let $e \in \mathbb{E}$, we denote by $\mathit{vars}(e)$ the set of variables from $\mathbb{V}$ involved in e. Moreover, variables or values are valid (simple) expressions, *i.e.*, we have $\mathbb{D} \cup \mathbb{V} \subset \mathbb{E}$.

We make no assumption about the typing or syntactical correctness of values or expressions. Instead, we assume that any expression can be evaluated, possibly to $\perp$ (undefined). More precisely, a *binding* is a partial function $\beta : \mathbb{V} \to \mathbb{D}$. Let $e \in \mathbb{E}$ and β be a binding, we denote by $\beta(e)$ the evaluation of e under β. For instance, if $\beta_1 \stackrel{\text{df}}{=} \{x \mapsto 1, y \mapsto 2\}$, we have $\beta_1(x + y) = 3$. With $\beta_2 \stackrel{\text{df}}{=} \{x \mapsto 1, y \mapsto \texttt{"2"}\}$, Python raises an exception upon evaluation, which corresponds in our setting to $\beta_2(x + y) = \perp$; similarly, if the domain of β does not include $\mathit{vars}(e)$ then $\beta(e) \stackrel{\text{df}}{=} \perp$. The application of a binding to evaluate an expression is naturally extended to sets and multisets of expressions.

In the following, given a set X, we denote by $X^\star$ the set of multisets over X. We use the standard notations for mutisets, $+$ for sum, $-$ for difference, $\leq$ for inclusion, etc., as well as an extended set notation $\{\cdots\}$.

Definition 1 (Python-Coloured Petri Nets). *A* Python-coloured Petri net *(*PCPN*) is a tuple* (S, T, ℓ, α) *where:*

- S *is the finite set of* places*;*
- T*, disjoint from* S*, is the finite set of* transitions*;*
- ℓ *is a* labelling *function such that:*
  - *for all* $s \in S$*,* $\ell(s) \subseteq \mathbb{D}$ *is the* type *of* s*,* i.e.*, the set of values that* s *is allowed to carry,*
  - *for all* $t \in T$*,* $\ell(t) \in \mathbb{E}$ *is the* guard *of* t*,* i.e.*, a condition for its execution,*
  - *for all* $(x, y) \in (S \times T) \cup (T \times S)$*,* $\ell(x, y) \in \mathbb{E}^\star$ *is the* arc *from* x *to* y*. Arcs from* $S \times T$ *(resp.* $T \times S$*) are called* input arcs *(resp.* output arcs*);*
- α *is the* arc type *function that associates to each arc in* $(S \times T) \cup (T \times S)$ *a function* $\mathbb{D}^\star \times \mathbb{D}^\star \to {\mathbb{D}_\perp}^\star$ *that takes the marking of a place plus the evaluation of an arc annotation and returns the actual multiset of consumed or produced tokens. In particular we shall use four functions:*
  - $\alpha_= \stackrel{\text{df}}{=} (m, a \mapsto a)$ *for a regular arc;*

- $\alpha_? \stackrel{\text{df}}{=} (m, a \mapsto \emptyset \text{ if } a \leq m \text{ else } \{\bot\})$ *for a read arc;*
- $\alpha_{>>} \stackrel{\text{df}}{=} (m, a \mapsto a \text{ if } a = m \text{ else } \{\bot\})$ *for a flush arc;*
- $\alpha_{<<} \stackrel{\text{df}}{=} (m, a \mapsto \sum_{x \in a} \mathit{iter}(x))$ *for a fill arc, where iter is a function that builds a multiset from the elements in collection x (set, list, ...).* □

As usual, Petri nets are depicted as graphs in which places are round nodes, transitions are square nodes, and arcs are directed edges with arrow tips depending on the arc types: $\to$ for regular, — for read, $\twoheadrightarrow$ for flush (input arcs) or fill (output arc). Empty arcs, *i.e.*, arcs such that $\ell(x, y) = \emptyset$, are not depicted.

Definition 2 (Markings and Firing). *Let* $N \stackrel{\text{df}}{=} (S, T, \ell, \alpha)$ *be a* PCPN. *A* marking M *of* N *is a function on* S *that maps each place* s *to a finite multiset over* $\ell(s)$ *representing the tokens in* s. *A transition* $t \in T$ *is* enabled *at a marking* M *and a binding* β, *which is denoted by* $M[t, \beta\rangle$, *iff the following conditions hold:*

- M *has enough tokens,* i.e., *for all* $s \in S$, $\alpha(s, t)(M(s), \beta(\ell(s, t))) \leq M(s)$;
- *the guard is satisfied,* i.e., $\beta(\ell(t)) = \mathsf{True}$;
- *place types are respected,* i.e., *for all* $s \in S$, $\alpha(t, s)(M(s), \beta(\ell(t, s)))$ *is a multiset over* $\ell(s)$.

If $t \in T$ *is enabled at marking* M *and binding* β, *then* t *may* fire *and yield a marking* M' *defined for all* $s \in S$ *as* $M'(s) \stackrel{\text{df}}{=} M(s) - \alpha(s, t)(M(s), \beta(\ell(s, t))) + \alpha(t, s)(M(s), \beta(\ell(t, s)))$. *This is denoted by* $M[t, \beta\rangle M'$. □

We may observe how the various arc types are implemented:

- $\alpha_=$ always returns the evaluation of the arc $\beta(\ell(s, t))$ or $\beta(\ell(t, s))$ so we fall back to the definition given in [17, def. 1];
- on an input arc, $\alpha_?$ requires that the tokens from $\beta(\ell(s, t))$ are actually in the marking of s but then it returns $\emptyset$ so no token is consumed. But if some tokens are not in the place, returning $\{\bot\}$ forbids the firing because $\bot \notin \ell(s)$ by definition. Returning $\emptyset$ unconditionally would not work because it would be like removing the arc;
- on an input arc also, $\alpha_{>>}$ forces to find β such that $\beta(\ell(s, t)) = M(s)$ so that all the marking is consumed;
- finally, on an output arcs, $\alpha_{<<}$ transforms a collection of multisets (or other collection types) into the sum of these multisets which is how fill arcs are expected to behave.

3.2 Petri Nets Compositions

To implement control-flow operations, PCPN are equipped with control-flow operations adapted from the Petri Box Calculus [2] and Petri Net Algebra [3]. We refer to [17, sec 2.3] for a complete definition and give here a summary.

First, places of a PCPN are separated into *control-flow* and *data* places. To do so, PCPN are equipped with an additional labelling function σ that returns for each place its *status* which may be for control-flow places: `e` for an entry place that is marked when the PCPN starts its execution; `x` for an exit place

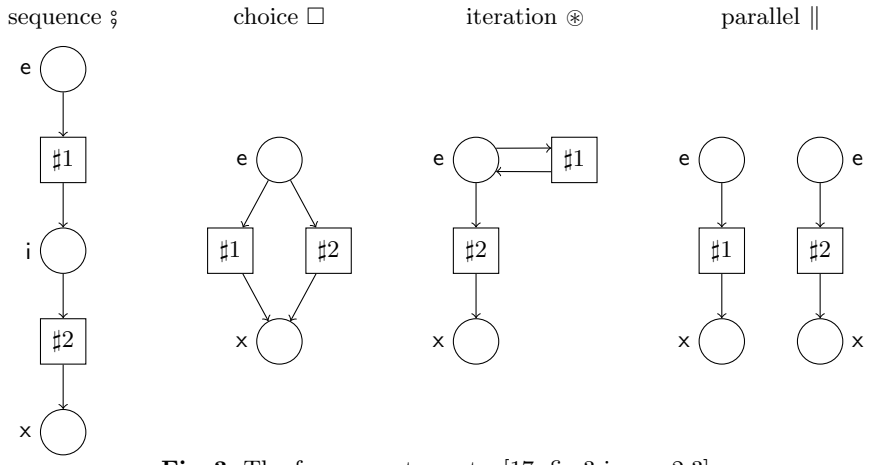

Fig. 3. The four operator nets. [17, fig 3 in sec 2.3]

that is marked when the PCPN has finished its execution; i for an internal place when the PCPN is in an intermediary state. For data places, status may be ε for an anonymous data place or an arbitrary label $\underline{name} \notin \{\mathsf{e}, \mathsf{i}, \mathsf{x}, \varepsilon\}$ for a named data place. Control-flow places must have type $\{\bullet\}$ and data places may have arbitrary types.

Then, control-flow operations are defined from the *operator nets* shown in Figure 3. Intuitively, each transition $\sharp i$ in an operator net is to be replaced with the i-th operand net of the specified operation. To do so, we consider in turn each place in the operator net and use its arcs to collect in the operand net the places to be combined. For instance, take the internal place of the sequence operator net to compute $N_1 \fatsemi N_2$: it is an output of $\sharp 1$ so we collect the exit places of N_1; it is also an input place of $\sharp 2$ so we collect the entry places of N_2. The sets of collected places are then composed using a cross-product and become internal places in the resulting net because we considered an internal place in the operator net. The same principle is applied for every place of the operator net which results in a composition of N_1 and N_2 whose control-flow places have been combined to implement the required control-flow. To finish the control-flow operation, we have to glue together (adding the markings) all the named data places that have the same name (usually one such place comes from each operand name), so that each named place is present only once. Anonymous data places are left untouched because they are considered local to each operand net.

One more operation is needed for our purpose, this is *name hiding* $N/\underline{name}$ that replaces the status of every place named $\underline{name}$ with ε, making it anonymous and no more mergeable upon control-flow compositions.

3.3 From ABCD to Petri Nets

The translation of ABCD to PCPN is defined through a recursive function *net* that takes two arguments: an environment that is used to collect information

symbol definition
$\mathsf{net}(env, \text{"}\textbf{symbol}\ name_1, \ldots, name_k \hookleftarrow tail\text{"})$
$\stackrel{\text{df}}{=} \mathsf{net}(env + \{name_1 \mapsto [\mathsf{symbol}], \ldots, name_k \mapsto [\mathsf{symbol}]\}, \text{"}tail\text{"})$
type definition
$\mathsf{net}(env, \text{"}\textbf{typedef}\ name\ :\ type \hookleftarrow tail\text{"})$
$\stackrel{\text{df}}{=} \mathsf{net}(env + \{name \mapsto [\mathsf{type}, \mathsf{type}(env, type)]\}, \text{"}tail\text{"})$
buffer definition
$\mathsf{net}(env, \text{"}\textbf{buffer}\ name : type = expr \hookleftarrow tail\text{"})$
$\stackrel{\text{df}}{=} N_{buffer} \parallel \mathsf{net}(env + \{name \mapsto [\mathsf{buffer}, \mathsf{type}(type), expr]\}, \text{"}tail\text{"})$
net definition
$\mathsf{net}(env, \text{"}\textbf{net}\ name(par_1, \ldots, par_k) : \downarrow\!\!\rightarrow sub \leftarrow\!\!\downarrow\ tail\text{"})$
$\stackrel{\text{df}}{=} \mathsf{net}(env + \{name \mapsto [\mathsf{net}, sub, par_1, \ldots, par_k]\}, \text{"}tail\text{"})$
constant definition
$\mathsf{net}(env, \text{"}\textbf{const}\ name = expr \hookleftarrow tail\text{"})$
$\stackrel{\text{df}}{=} \mathsf{net}(env + \{name \mapsto [\mathsf{const}, expr]\}, \text{"}tail\text{"})$
parallel composition
$\mathsf{net}(env, \text{"}proc_1 \mid proc_2\text{"}) \stackrel{\text{df}}{=} \mathsf{net}(env, \text{"}proc_1\text{"}) \parallel \mathsf{net}(env, \text{"}proc_2\text{"})$
sequential composition
$\mathsf{net}(env, \text{"}proc_1\ ;\ proc_2\text{"}) \stackrel{\text{df}}{=} \mathsf{net}(env, \text{"}proc_1\text{"}) \fatsemi \mathsf{net}(env, \text{"}proc_2\text{"})$
choice composition
$\mathsf{net}(env, \text{"}proc_1 + proc_2\text{"}) \stackrel{\text{df}}{=} \mathsf{net}(env, \text{"}proc_1\text{"}) \square\ \mathsf{net}(env, \text{"}proc_2\text{"})$
iteration
$\mathsf{net}(env, \text{"}proc_1 * proc_2\text{"}) \stackrel{\text{df}}{=} \mathsf{net}(env, \text{"}proc_1\text{"}) \circledast \mathsf{net}(env, \text{"}proc_2\text{"})$
nested process
$\mathsf{net}(env, \text{"}(proc)\text{"}) \stackrel{\text{df}}{=} \mathsf{net}(env, proc)$
always possible action
$\mathsf{net}(env, \text{"}[\mathsf{True}]\text{"}) \stackrel{\text{df}}{=} N_{\mathsf{True}}$
always blocking action
$\mathsf{net}(env, \text{"}[\mathsf{False}]\text{"}) \stackrel{\text{df}}{=} N_{\mathsf{False}}$
unconditional action
$\mathsf{net}(env, \text{"}[access_1, \ldots access_k]\text{"}) \stackrel{\text{df}}{=} \mathsf{net}(env, \text{"}[access_1, \ldots access_k\ \textbf{if}\ \mathsf{True}]\text{"})$
conditional action
$\mathsf{net}(env, \text{"}[access_1, \ldots access_k\ \textbf{if}\ expr]\text{"}) \stackrel{\text{df}}{=} N_{action}$
anonymous net instance
$\mathsf{net}(env, \text{"}name(arg_1, \ldots, arg_k)\text{"})$
$\stackrel{\text{df}}{=} \mathsf{net}(env, \text{"}[\![sub \mid par_1 \leftarrow arg_1, \ldots, par_k \leftarrow arg_k]\!]\text{"})/\underline{\mathit{buf1}}/\cdots/\underline{\mathit{bufn}}$
where $env(name) = [\mathsf{net}, sub, par_1, \ldots, par_k]$
and $\underline{\mathit{buf1}}, \ldots, \underline{\mathit{bufn}}$ are the names of the buffers declared inside net $name$
named net instance
$\mathsf{net}(env, \text{"}alias :: name(arg_1, \ldots, arg_k)\text{"})$
$\stackrel{\text{df}}{=} \langle\!\langle \mathsf{net}(env, \text{"}name(arg_1, \ldots, arg_k)\text{"}) \mid name(arg_1, \ldots, arg_k) \leftarrow alias \rangle\!\rangle$

Fig. 4. Translation rules of the PCPN semantics of ABCD.

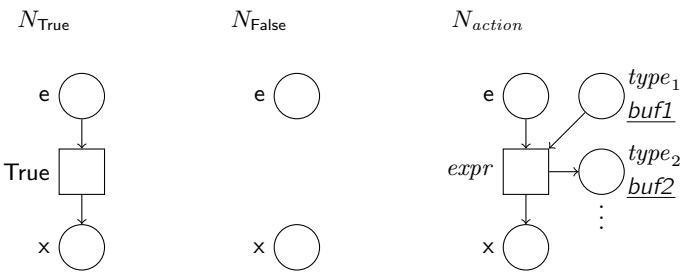

Fig. 5. Base nets for atomic actions.

about ABCD declarations encountered so far, and a fragment of ABCD source code that remains to be compiled. The environment is a mapping from declared names to various information about the corresponding declarations. Look for instance at the first rule in Figure 4: when a symbol declaration is found, this is simply recorded in the environment and the translation proceeds with the rest of the code. Type definition is treated similarly but the value of the type is recorded also, it is computed using an auxiliary function *type*. We will not detail it because it is both straightforward and an implementation detail (interested readers may look at module snakes.typing that is used for this purpose [16]). Buffer definition starts to build parts of the resulting Petri net: the net returned by translating the rest of the code is composed in parallel with a net N_{buffer} that consists of a single data place, whose type, status and marking is exactly the type, name and initial content of the buffer respectively. When applying the composition, this will result in merging this place with another empty copy (see the translations of actions below) in order to initialise the marking. The next six rules are straightforward.

Then come two rules to build simple actions: "[True]" (resp. "[False]") is translated to a net N_{True} (resp N_{False}) depicted in Figure 5. Then, an unconditional action is just a conditional action with a true guard. Conditional action itself is implemented by a simple Petri net N_{action} as sketched in Figure 5:

- it has one entry place and one exit place, connected by a single transition whose guard is exactly the guard of the action;
- for each buffer access, there is a data place named and typed as the buffer (which is known from the environment) to implement the buffer, and an arc with the appropriate type:
  - $\alpha_=$ on an input arc for a consumption ("-"),
  - $\alpha_=$ on an output arc for production ("+"),
  - $\alpha_?$ on an input arc for a test ("?"),
  - $\alpha_{>>}$ on an input arc for a flush (">>"),
  - $\alpha_{<<}$ on an output arc for a fill ("<<"),
  - a swap ("<>") is decomposed into the corresponding pair of consumption and production;
- multiple regular arcs to (resp. from) the same buffer are merged, *i.e.*, there annotations is summed. Multiple read arcs and multiple fill arcs are merged

as well. Multiple arcs with the same direction but of one of the other types or of mixed types are forbidden. For instance we cannot flush twice a place nor we can flush it and at the same time consume one token from it.

Finally, net instances are built as follows:

- function *net* is called recursively on the content *sub* of the net declaration in which we substitute each parameter par_i with the effective argument arg_i, which is denoted by $[\![sub \mid par_1 \leftarrow arg_1, \ldots, par_k \leftarrow arg_k]\!]$;
- the data places for all the buffer declared inside the net are hidden so that they become private to the resulting Petri net and no more mergeable;
- if the net instance is named, its nodes are renamed by replacing the prefix $name(arg_1, \ldots, arg_k)$ with the name *alias* of the instance, which is denoted by $\langle\langle \cdots \mid name(arg_1, \ldots, arg_k) \leftarrow alias \rangle\rangle$.

Definition 3 (Semantics of ABCD). *Given function net as defined above, the complete semantics of an* ABCD *specification "spec" is the* PCPN *returned by* $net(\{\}, \text{"spec"})$ *in which all the entry places are marked with a single "•".* □

Note that we did not include import statement in the semantics because it does not change the net itself. It is rather an implementation detail and just consists in making the imported names available in the execution environment of the Petri net. This is made also for constants and symbols.

4 Applications and Use Cases

This section presents uses of ABCD for scientific applications and for teaching. The first one is detailed to provide a complete example of an actual ABCD model. The next examples will be partly presented due to space limitation, so chosen fragments will be showed to illustrate particular points we would like to discuss.

4.1 Critical Systems

A model of a railroad crossing system with multiple tracks has been developed in ABCD in order to generate Petri nets for the model-checking contest (MCC) 2012 [11,12]. The purpose of the MCC is to compare model-checkers on a variety of models with a scaling parameter, *i.e.*, a parameter allowing to tune the size of the model and of its state space. For the railroad model, the idea was to develop a model that is generic with respect to the number of tracks, each track being a net instance. However, as one may have remarked with the dining philosophers model of Figure 1, ABCD has no mechanism to instantiate a number of processes that depends on a parameter: in the case of the philosophers, we have to explicitly instantiate a statistically chosen number of nets. In the case of the railroad model, this was solved using a template engine that allowed to generate ABCD source code for any number of tracks, each such ABCD file was then converted to a Petri net. Below, we show the model for two tracks, it is easy to figure out how this can be parametrised (actually, only the first and last lines need to be changed).

The model is based on three nets to model respectively: a pair of gates; generic tracks with a green/red light to control the progression of trains; and a controller to count the trains and operate the gates. The model starts with various declarations: number of tracks, values and type for the state of the gates, then all the global buffers. Note line 5 how a comment is inserted just like in Python: "`#this comment extends until the end of the line↲`". Note also that in source code below, the omitted lines are all blank. Buffer "green" stores the number of each track that has a green light, so that a red light is modelled by the absence of the corresponding token. Other buffers are dedicated to the communication between the components: "enter" receives the track numbers on which a trains approaches the gates; "leave" receives the track numbers on which a train leaves the gates; "down" allows to ask the gates to go down; "up" allows to ask the gates to go up; "done" is used by the gates to notify the controller that they have finished a command.

```
1  const NUM = 2
2  symbol OPEN, MOVING, CLOSED
3  typedef gatestate : enum(OPEN, MOVING, CLOSED)
4
5  # green lights on tracks
6  buffer green : int = range(NUM)
7  # tracks -> controller
8  buffer enter : int = ()
9  buffer leave : int = ()
10 # controller -> gates
11 buffer down : BlackToken = ()
12 buffer up : BlackToken = ()
13 # gates -> controller
14 buffer done : gatestate = ()
```

The gates are modelled with a net that has a private buffer "state" reflecting the current position of the gates. The associated process is a repeated sequence of four actions: wait for the request to go down and start proceed it; arrive down and notify the controller; wait for the request to go up and start proceed it; arrive up and notify the controller.

```
16 net gates () :
17     buffer state : gatestate = OPEN
18     ([down-(dot), state<>(OPEN=MOVING)]
19      ; [state<>(MOVING=CLOSED), done+(CLOSED)]
20      ; [up-(dot), state<>(CLOSED=MOVING)]
21      ; [state<>(MOVING=OPEN), done+(OPEN)])
22     * [False]
```

The model for the tracks has the same structure. A local buffer "crossing" is marked when a train is crossing the road and a series of actions executed repeatedly corresponds to the successive steps of the progression of a train: approach the gates and switch the green light to red; start crossing the road only on a green light and switch it back to red; leave the crossing zone.

```
24 net track (this) :
25     buffer crossing : BlackToken = ()
26     ([enter+(this), green-(this)]
27      ; [green-(this), crossing+(dot)]
28      ; [crossing-(dot), leave+(this)])
29     * [False]
```

Then, the model for the controller is composed of one buffer "count" to count the trains present in the supervised zone, one buffer "waiting" to record on which track a train is waiting for the green light, and one process that can repeatedly execute one of four behaviours: detect the first train approaching ("c == 0" line 34) and ask the gates to go down, then wait until they arrive down and give the green light to the train that triggered this behaviour; count another train approaching (line 36) and give it the green light; count a train leaving the zone while there are still other trains in the zone ("c > 1" line 37); detect that the last train leaves the zone ("c == 1" line 37), ask the gates to go up and wait until this order is executed. When a train leaves, the corresponding light is turned green again so another train is allowed to approach on this track. Without this mechanism, we could have an accumulation of tokens in buffer "leave".

```
31 net controller () :
32     buffer count : int = 0
33     buffer waiting : int = ()
34     (([enter-(num), count<>(c=c+1), down+(dot), waiting+(num) if c == 0]
35       ; [done-(CLOSED), waiting-(num), green+(num)])
36      + [enter-(num), count<>(c=c+1), green+(num) if c > 0]
37      + [leave-(num), green+(num), count<>(c=c−1) if c > 1]
38      + ([leave-(num), green+(num), count<>(c=c−1), up+(dot) if c == 1]
39         ; [done-(OPEN)]))
40     * [False]
```

The complete system is just a parallel composition of instances of these nets: one pair of gates, one controller and several tracks.

```
42 # all components in parallel
43 gates() | controller() | track(0) | track(1)
```

On the Petri net obtained from this model, safety (1) and liveness (2) LTL properties may be verified using Neco-SPOT model-checker [9]:

$$\forall\, 0 \leq i \leq 1 :\ G\big(\mathsf{track}(i).\mathsf{crossing} \neq \emptyset \Rightarrow \mathsf{gates}().\mathsf{state} = \{\mathsf{CLOSED}\}\big) \quad (1)$$

$$G\big(\mathsf{gates}().\mathsf{state} = \{\mathsf{CLOSED}\} \Rightarrow F(\mathsf{gates}().\mathsf{state} = \{\mathsf{OPEN}\})\big) \quad (2)$$

where G and F are respectively the *globally* and *eventually* modalities. Note that Neco actually requires slight changes to the model presented above because it does not support dot or parentheses in place names (so buffers "crossing" and "state" need to be replaced with global buffers), symbols (to be replaced with constants) nor enumerated types (to be replaced with "int"), see [15].

4.2 Security Protocols

A massively parallel CTL* model-checker for models of security protocols has been developed in [10]; a side product of this work has been the actual modelling of a bunch of security protocols, which has been made using ABCD. We show here two excerpts of a model of the Needham-Schroeder public key protocol [13], it is not needed to present it to understand our purpose, it is enough to know that some agents are exchanging encrypted messages in the presence of an attacker.

```
# implementation of a Dolev-Yao attacker
from dolev_yao import *
```

First, a Python module called "dolev_yao" is imported, it contains the definition of various symbols ("CRYPT", "PUB", "PRIV" and "NONCE") used to make a symbolic treatment of cryptography (*i.e.*, replace actual computation of cryptographic operations with terms that express it, which is a classical treatment of cryptography when modelling protocols) as well as a class "Spy" that implements a Dolev-Yao attacker [8]. Such an attacker has an infinite memory and computational power, however, it cannot break the cryptography that is assumed perfect. So it can capture messages exchanged by the other agents, gain knowledge (*i.e.*, learn) by decomposing messages, decrypt messages when it has the key to do so, recompose or encrypt messages, and inject new messages on the network. Recomposition leads in practice to infinite computation because just one object may be assembled into sequences of arbitrary sizes. The classical solution to bound the computation is to restrict compositions of objects to patterns that actually appear in the protocol: indeed, other sequences are useless to produce because no other agent in the system would ever use them as a valid message.

Modelling such an attacker is actually not difficult, but it requires to implement the learning actions discussed above, which immediately leads to state space explosion because we expose in the model all the intermediate steps of a fixed point computation (*i.e.*, the attacker applies each learning action until it cannot learn anything new). Instead of this, using ABCD, we *implement* (*i.e.*, program) the Dolev-Yao attacker directly in Python and have an efficient execution of this learning phase on a single transition. So, in the case of the Needham-Schroeder protocol, we have a simple model of the attacker as follows:

```
net Mallory (this, agents) :
    buffer knowledge : object = ([this, (NONCE, this), (PRIV, this)]
                                 + [(PUB, a) for a in agents]
                                 + agents)
    # Dolev-Yao engine, bound by the protocol signature
    buffer spy : object = Spy((CRYPT, (PUB, int), int, (NONCE, int)),
                              (CRYPT, (PUB, int), (NONCE, int), (NONCE, int)),
                              (CRYPT, (PUB, int), (NONCE, int)))
    # capture on message and learn from it
    ([spy?(s), nw-(m), knowledge>>(k), knowledge<<(s.learn(m, k))]
     # loose message or inject another one (may be the same)
     ; ([True] + [spy?(s), knowledge?(x), nw+(x) if s.message(x)]))
    * [False]
```

Buffer "knowledge" stores all the information learnt by the attacker; initially, this is information about itself, plus the public keys and identities of the other agents. Buffer "spy" stores an instance of class "Spy" that implements the Dolev-Yao learning mechanism. This class is instantiated with the signature of the protocol, that is, the types of all the possible messages, here also presented in a symbolic way (*i.e.*, as terms). Then, the process executed by the attacker is always the same for any protocol, it repeatedly execute a sequence of two behaviours:

- line 34, capture a message on the network "nw−(m)", learn from it and the previous knowledge by calling method "Spy.learn", and enrich the knowledge with decomposed and recomposed messages;
- line 36, immediately loop with "[True]" which causes a message loss, or inject a new message on the network using method "Spy.message" to check that a syntactically correct message is actually injected (anything else would yield additional states for nothing).

This example illustrates well how in ABCD programming and modelling can nicely complement each other. Not only this is simpler for the modeller, but also it leads to more efficient verification because it reduces state spaces a lot. Of course, the programmed part has to be correct but in this case, it is only 100 lines of simple Python that can be carefully written and scrutinised as well as thoroughly tested. In particular, we have verified that known attacks are detected, which shows that our Dolev-Yao attacker is at least as good as that implemented by specialised tools like Avispa [1]. On the other hand, when functions of the system are programmed and model-checking is applied, we can consider that this code has a good level of certification because it has been intensely exercised without triggering a bug nor producing an invalid run from the model-checking point of view. In other word, this code is part of the model and is verified just like the ABCD part.

ABCD has been initially developed for the purpose of modelling an industrial peer-to-peer storage system whose security needed to be assessed [5,19]. The kind of models we obtain for such a use case is similar to models of security protocols, the main difference is that we model identical peers instead of distinct partners with distinct roles. However, additionally to model-checking in the Dolev-Yao perspective, we have used statistical analysis of large sets of random traces to assess quantitative properties; in particular, we obtained the number of file loss with respect to the number of malicious peers connected to the system (a typical case where the yes/no answer of a model-checker is not enough).

4.3 Teaching Formal Modelling and Verification

The most recurring use of ABCD is for teaching: it is used for years at the university of Évry to teach formal modelling to master students in computer science. They are presented models like those discussed above and they must produce such models themselves.

This experience has shown that ABCD is not easier nor harder to understand by such students than coloured Petri nets. However, when it comes to actually produce models, students are much more successful and efficient using ABCD. In particular, sub-processes are naturally adopted and the models produced tend to be clearly structured, contrasting with Petri net models that quickly become random-looking and completely wrong. Clearly, the similarity of the syntax of ABCD with that of programming languages helps a lot to this respect. The interactive simulator is also very much appreciated because it allows an immediate feedback during the process of modelling.

5 Implementation, Compilation and Simulation

ABCD is implemented in SNAKES [16, 18] as a compiler that takes ABCD source code as its input and has various possible outcomes: pictures, SNAKES' variant of PNML,[1] or an interactive simulator that allows to execute a model in a user-friendly graphical user-interface. A naive reachability model-checker is also implemented but we will not describe it, and actually we did not describe the related parts in the syntax, because it is intended to be replaced with something more general, robust and efficient. The compiler can be invoked from a command line interface or from a Python program in which case the constructed Petri net object is returned as an instance of SNAKES' PetriNet class.

The compiler is a rather straightforward implementation of the rules presented in Figure 4, extended with syntactic and semantic constraints checks.

The interactive simulator is an important feature of the ABCD compiler, it is often the main tool invoked by users during the design of a model, just like a programmer invokes the compiler and make dry runs to exercise programs. The simulator has been completely reworked recently and is now displayed as a responsive Web user interface: when simulation is asked from the compiler, a Web page opens in which all the interaction takes place. Figure 6 show the main parts of this simulator (hiding a menu with a few auxiliary features):

- at the top is a player that allows to automatically run a chosen number of actions randomly selected, with a controlled speed;
- in the left column, under the ABCD label, the ABCD source code is displayed and enabled actions are highlighted. This is static information that will not evolve with the execution (apart from the highlighting);
- in the right column, a dynamic tree view of the model allows the user to observe and control the execution. Because a sub-process may have many instances, it is necessary to display separately each instance so that it can be controlled separately and its state (*i.e.*, the content of its buffers) can be displayed separately. In the example of Figure 6 that shows a simulation of the specification from Figure 1 (restricted to two philosophers for readability), we can see the instances of "**net** philo", each with its actions in various enabling states. For instance, among the two actions of the first instance,

[1] Which is not valid PNML in the case of the coloured models obtained from ABCD [18].

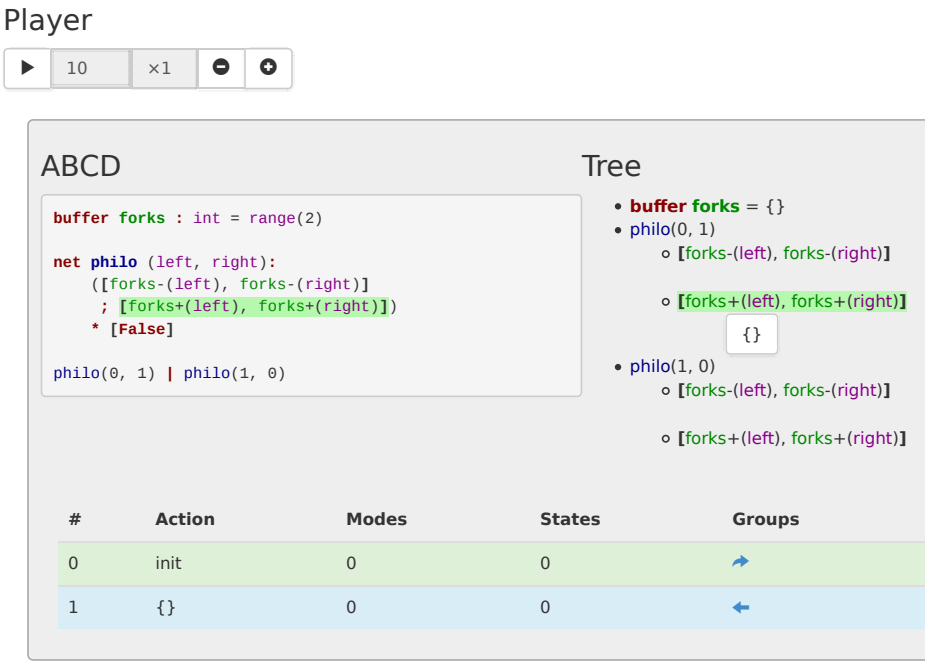

Fig. 6. A screenshot of (part of) the Web user interface.

only the second one is enabled for only one possible mode that is {} here because there is no variables;
– finally, at the bottom is the trace executed so far. Using the arrow in the right-most column, it is possible to navigate into the trace in order to update the tree view to the state it had just before the corresponding action has been executed; it is also possible to restart a new trace from this point.

The architecture of the simulator is modular and flexible and it is possible to adapt it to simulate Petri nets based formalisms others than ABCD. For instance, we have developed a similar simulator for models of biological regulatory networks [6, 7] where the state is a plot of concentration levels of the regulated products (on the y axe) with respect to time (on the x axe). To do so, it is necessary to provide some HTML code that provides the presentation of the model and its state, and some JavaScript code to translate the interactions with them into appropriate calls to the simulation engine, as well as to implement the updates requested by the engine.

6 Conclusion

We have presented ABCD, a modelling language that is mixing the Python programming language and a process algebras. We have defined the Petri nets semantics of ABCD, targeting a Python-coloured variant of Petri nets extended

with read-, flush- and fill-arcs. All this is implemented and freely available with the toolkit SNAKES. Use cases of ABCD have showed its suitability to qualitative analysis through model-checking as well as to quantitative analysis through statistics on large sets of traces. ABCD is also suitable to teach formal modelling to master students and is regularly used for this purpose.

Among the numerous languages or notations with a Petri net semantics, apart from the PBC and PNA family of which ABCD is a member, essentially one can be directly related to ABCD: the *Basic Petri Net Programming Notation*, B(PN)^2 [4], is also a process algebra and its semantics is expressed in terms of coloured Petri nets. Data in B(PN)^2 is stored into variables which makes necessary to distinguish the value of a variable before and after the execution of an atomic action. So, a variable x is actually used as $'x$ and x' within the Boolean expressions that form the atomic actions. After having taught both B(PN)^2 and ABCD, it appears that ABCD is easier to understand than B(PN)^2 because it is more explicit with respect to data storage through buffer accesses, the latter being clearly distinguished from the guard when B(PN)^2 unifies both aspects. Finally, B(PN)^2 is implemented only in the PEP toolkit [14] that is not maintained anymore.

Future work about ABCD will aim at implementing various extensions and improving its connection with analysis tools.

Considered extensions include: *buffer capacities* to block actions that would add (resp. remove) too much tokens to (resp. from) a buffer; *arrays of buffers* to declare k identical buffers at the same time, where k is a constant; *parametric composition* to compose identical processes depending on a parameter, for instance to compose the k tracks of the railroad example of Section 4.1 where k is a constant; *dynamic threads of executions* like suggested in [17, sec. 4.3] allowing to create dynamic instances of sub-processes with abort/suspend/resume capabilities and to emulate function calls (including recursive calls); *syntax for raw Petri nets* allowing to include arbitrary Petri nets within an ABCD model, which is sometimes useful when control-flow is over-constraining; *inhibitor access* to leverage the inhibitor arcs already implemented in SNAKES. These extensions will be included in a demand driven fashion: they are identified to be potentially useful but the actual need for them has not been too crucial so far to trigger the effort of implementing them.

Two ways are envisaged to improve the usability of ABCD for analysis. First, we would like to ease the invocation of Neco-SPOT [9] to support direct model-checking of LTL formulas on ABCD models, this will require some work on Neco itself that currently has a few blocking limitations as explained above. Then, we are already working on a fast multi-core simulation engine coupled with automatic execution-related data collection for statistical analysis. Currently, this is a manual process with an inefficient execution of the traces, and so, building even simple statistics is quite a tedious process.

Finally, we will continue to model systems using ABCD in the context of research projects as well as for teaching because it has proved to be a good tool for these purposes.

References

1. Armando, A., *et al.*: The AVISPA tool for the automated validation of Internet security protocols and applications. In: Proc. of CAV'05. LNCS, vol. 3576. Springer (2005)
2. Best, E., Devillers, R., Hall, J.G.: The box calculus: a new causal algebra with multi-label communication. In: Advances in Petri Nets 1992, The DEMON Project. Springer (1992)
3. Best, E., Devillers, R., Koutny, M.: Petri net algebra. Springer (2001)
4. Best, E., Hopkins, R.P.: B(PN)^2 – a basic Petri net programming notation. In: Proc. of PARLE'93. Springer (1993)
5. Chaou, S., Utard, G., Pommereau, F.: Evaluating a peer-to-peer storage system in presence of malicious peers. In: Proc. of HPCS'11. IEEE Computer Society (2011)
6. Delaplace, F., Di Giusto, C., Giavitto, J.L., Klaudel, H., Spicher, A.: Activity networks with delays application to toxicity analysis. Tech. Rep. hal-01152719, I3S, univ. Nice Sophia Antipolis (2015)
7. Di Guisto, C., Klaudel, H., Delaplace, F.: Systemic approach for toxicity analysis. In: Proc. of BioPPN'14. Workshop Proceedings, vol. 1159. CEUR (2014)
8. Dolev, D., Yao, A.C.: On the security of public key protocols. IEEE Transactions on Information Theory 29(2) (1983)
9. Fronc, Ł., Duret-Lutz, A.: LTL model checking with Neco. In: Proc. of ATVA'13. LNCS, vol. 8172. Springer (2013)
10. Gava, F., Pommereau, F., Guedj, M.: A BSP algorithm for on-the-fly checking CTL* formulas on security protocols. The Journal of Supercomputing (2014)
11. Kordon, F., Buchs, D., Garavel, H., Hillah, L.: MCC'2016 - models. `http://mcc.lip6.fr/models.php`
12. Kordon, F., Linard, A., Buchs, D., Colange, M., Evangelista, S., Fronc, Ł., Hillah, L.M., Lohmann, N., Paviot-Adet, E., Pommereau, F., Rohr, C., Thierry-Mieg, Y., Wimmel, H., Wolf, K. (eds.): Raw report on the Model-Checking Contest at Petri nets 2012. No. arXiv:1209.2382, CoRR (September 2012)
13. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Communication of the ACM 21(12) (1978)
14. Parallel Systems Group, University of Oldenburg: The PEP tool. `http://peptool.sourceforge.net`
15. Pommereau, F.: Model-checking with ABCD and Neco. `https://goo.gl/FLVItx`
16. Pommereau, F.: The SNAKES toolkit. `http://www.ibisc.univ-evry.fr/~fpommereau/SNAKES/` with sources at `http://github.com/fpom/snakes`
17. Pommereau, F.: Algebras of coloured Petri nets. Lambert Academic Publishing (2010)
18. Pommereau, F.: SNAKES: A flexible high-level Petri nets library. In: Proc. of PETRI NETS'15. LNCS, vol. 9115. Springer (2015)
19. Sanjabi, S., Pommereau, F.: Modelling, verification, and formal analysis of security properties in a P2P system. In: Proc. of COLSEC'10. IEEE Digital Library, IEEE Computer Society (2010)
20. The Python Software Foundation: The Python tutorial. `http://docs.python.org/tutorial`
21. The Python Software Foundation: Python web site. `http://www.python.org`