ZINC: a compiler for "any language"-coloured Petri nets

Franck Pommereau

IBISC, Univ. Évry, Univ. Paris-Saclay, 91025 Évry, France
franck.pommereau@ibisc.univ-evry.fr

TECHNICAL REPORT

Abstract. We present ZINC, that is the code name for a complete rewrite of SNAKES [8,9], incorporating the compilation approach from Neco [4,5] at its heart. Doing so, ZINC allows to substantially extend SNAKES and Neco in various ways: efficient firing and statespace computation, Petri nets coloured by potentially any programming language, more general class of Petri nets supported. At its current state, ZINC allows to create Petri nets from a Python program, or to load them from a simple unified file format, then to compile nets targeting: Python [10], Go [6], and CoffeeScript [1] (the latter being compiled to JavaScript). This choice of statically and dynamically typed languages demonstrates the feasibility of supporting potentially any language as a colour domain. This paper presents what one can do using ZINC in this state, how this is implemented, and what is planned as future developments.

Keywords: Petri nets, model compilation, library.

1 Introduction

For more than ten years, we have developed SNAKES [8,9] as a general purpose Petri net library, privileging flexibility over the efficiency of executions (transitions firing and statespace exploration). SNAKES is a Python [10] library to create and execute Python-coloured Petri nets. Over the years, the need for efficient executions has led to the development of Neco [4] that uses model compilation to attain efficiency. Using Neco, a Petri net is compiled into a software library that provides all the necessary primitives to fire transitions and explore the statespace.

In this paper, we present ZINC, that is the code name for a complete rewrite of SNAKES, incorporating the compilation approach from Neco at its heart. This allows to bring three major improvements to both SNAKES and Neco. (1) SNAKES is now quite old and a major code cleanup is desirable, in particular, supporting both the 2.x and 3.x series of Python [10] leads to a lot of code only dedicated to ensure compatibility. Thus, ZINC drops compatibility with Python 2.x that will be discontinued by year 2020. (2) Using a compilation approach implies a greater distance to the annotation language. Not only this reduces the need for code dedicated to handle the annotation language, but, more importantly,

we are able now to support potentially any programming language as a colour domain. To demonstrate this, ZINC currently supports Python, Go [6], and CoffeeScript [1] (and thus indirectly JavaScript that is the target of the CoffeeScript compiler). This is a dramatic enhancement with respect to SNAKES that supports only Python. Supporting Python in ZINC was the most natural choice to provide in the future a good compatibility with SNAKES. Then, Go was chosen to demonstrate that a statically typed languages with type declarations can be actually supported. Finally, CoffeeScript has been chosen as a reasonable alternative to JavaScript with the aim to enable for Petri nets simulations within web browsers. (3) Finally, building a Petri net compiler forced us to focus more in depth on the process of firing transitions, which allowed to release several restrictions enforced by SNAKES and inherited by Neco, leading to support a more general class of Petri nets. For instance, ZINC allows to use arbitrary expressions on input arcs while this is forbidden in SNAKES, the only remaining restriction is to forbid free variables.

From this rewrite we thus expect:

- a cleaner, lighter, and more modern library;
- efficient executions out-of-the-box;
- support for potentially any programming language in Petri nets annotations;
- a generalisation of the supported class of Petri nets.

On the other hand, flexibility that has always been a strength of SNAKES is likely to be harder to achieve within the compilation approach. Indeed, adding a new Petri net features may imply to change how nets are compiled with potentially, but not necessarily, an impact for all the supported annotations languages. ZINC will have to provide help to this respect. Moreover, while this difficulty is expected for most changes on the transition rule, this is absolutely not the case for the large majority of extensions that are dedicated to handle Petri nets at a structural level (for instance, supporting algebras of Petri nets like PBC/PNA [2] can be implemented exactly the same way are they are currently implemented in SNAKES).

In the next sections, we present the current features of ZINC and how it can be used for various tasks. Then we discuss its implementation and performances, before to conclude and sketch a development roadmap.

2 Using ZINC

ZINC is currently at a very early stage of its development, but it is already a useful tool. Mainly, it allows to define Petri nets (just as with SNAKES) and to compile them, either to explore their statespace directly, or to use the resulting library from another tool. Below, we will use Python as the annotation language but everything could equally be made using Go or CoffeeScript.

ZINC supports a very general class of Petri nets coloured by potentially any programming language (3 being currently implemented). Tokens are arbitrary values of the annotation language. ZINC supports arcs annotated by values, variables, expressions, or tuples of such annotations (nesting being allowed). Unlike

SNAKES, ZINC allows to use expressions on input arcs, the only restriction is that all the variables used in such expressions must be bound by another arc (typically labelled by a variable). It supports also multiple arcs (*i.e.*, consuming or producing multiple tokens), whole-place arcs (flush and fill), inhibitor arcs, and test arcs (*i.e.*, read arcs), with combinations that SNAKES forbids (*e.g.*, both a test arc and a regular arc between the same place and transition).

At the time we write this, ZINC is barely documented, but SNAKES documentation can be used as a reference because ZINC can be seen as a subset of SNAKES with a mostly identical API.

2.1 Getting ZINC and installing it

ZINC is available at `https://github.com/fpom/zinc`, there is currently no installation procedure, all one needs is to clone the Git repository ("`git clone https://github.com/fpom/zinc.git`") and use its content:

- directory `zinc` is the Python package;
- script `zn` is a command line net compiler;
- directory `libs` provides libraries needed to execute the code generated by ZINC, in particular, marking data-structures for Go and CoffeeScript.

In the following, we assume that we work in the directory created by command "`git clone`" (which should be called "`zinc`"), and that "`.`" is in `PATH` environment variable (so that one can invoke `zn` directly instead of `./zn`).

ZINC has been developed and tested under Linux only but it should work on any Unix, like MacOS-X or "Windows subsystem for Linux" [3]. It depends on:

- Python (version used: 3.5) `http://www.python.org`;
- 竜 TatSu library (version used: 4.2.3) `http://github.com/neogeny/TatSu`;
- the Go compiler (version used: 1.7.4) `http://golang.org`;
- the CoffeeScript compiler (version used: 2.0.2) `http://coffeescript.org`.

ZINC is free software release under the GNU Lesser General Public License (`http://www.gnu.org/licenses/lgpl.html`).

2.2 Creating a Petri net

The first step to use ZINC is to create a Petri net. There are currently two ways to do so. The first one is to use ZINC as a library, like one could do using SNAKES. The API is almost the same with one major difference: annotations in ZINC are provided as source code within strings and not interpreted (in particular, not checked for syntax correctness). This means that errors in the annotations will remain unnoticed until the net is compiled, or even executed. Figure 1 shows two simple models created this way, using ZINC or SNAKES.

The other way to create a Petri net is to edit a "`.zn`" file as shown at the bottom of Figure 1. ZINC introduces this generic file format that makes it easy to specify nets annotated with arbitrary source code. A Petri net in this format can be loaded easily using function zinc.io.zn.load implemented in ZINC.

ZINC

```
from zinc.nets import *
net = PetriNet("Erathostene sieve", lang="python")
net.add_place(Place("p", ["2", "3", "4", "5", "6"], "int"))
net.add_transition(Transition("t", "n % d == 0"))
net.add_input("p", "t", MultiArc(Variable("n"), Variable("d")))
net.add_output("p", "t", Variable("d"))
```

SNAKES

```
from snakes.nets import *
net = PetriNet("Erathostene sieve")
net.add_place(Place("p", [2, 3, 4, 5, 6], tInteger))
net.add_transition(Transition("t", Expression("n % d == 0")))
net.add_input("p", "t", MultiArc([Variable("n"), Variable("d")]))
net.add_output("p", "t", Variable("d"))
```

`.zn` file

```
lang python
net "Erathostene sieve" :
    place p int = 2, 3, 4, 5, 6
    trans t n % d == 0 :
        < p var = n
        < p var = d
        > p var = d
```

Fig. 1. Creating a Petri net using ZINC as a library (top), using SNAKES (middle), or through a "`.zn`" file (bottom). This net is a model of Eratosthenes' sieve borrowed from the *model-checking contest* [7].

2.3 Compiling a Petri net

Once we have a Petri net, we can compile it to a library, implemented in the language we have chosen for the annotations (Python in the example above). If our net is a PetriNet object in a Python program, like at the top of Figure 1, compilation is triggered with a single method call, *e.g.*, net.build(saveto="`sieve.py`") which saves the generated source code into a file. Otherwise, if the net resides in a file "`sieve.zn`", we can invoke compilation from the command line using "`zn -o sieve.py sieve.zn`".

We discuss now how this generated code can be used, first as a standalone program, then as a library.

2.4 Exploring the statespace

File "`sieve.py`" that we have just generated has can be used as a program to explore the statespace in various ways, which is illustrated in Figure 2 where we use options: "`-ms`" to count the number of reachable markings, "`-ds`" to count the number of deadlocks, "`-d`" to list the deadlocks, "`-m`" to list the markings, "`-g`" to list the markings and their successors, and "`-l`" to show the detailed statespace (modes, consumed and produced tokens).

```
$ python3 sieve.py -ms
4 reachable states
$ python3 sieve.py -ds
1 deadlocks
$ python3 sieve.py -d
[3] {'p': [2, 3, 5]}
$ python3 sieve.py -m
[0] {'p': [2, 3, 4, 5, 6]}
[1] {'p': [2, 3, 4, 5]}
[2] {'p': [2, 3, 5, 6]}
[3] {'p': [2, 3, 5]}
$ python3 sieve.py -g
[0] {'p': [2, 3, 4, 5, 6]}
 > [1] {'p': [2, 3, 4, 5]}
 > [2] {'p': [2, 3, 5, 6]}
[1] {'p': [2, 3, 4, 5]}
 > [3] {'p': [2, 3, 5]}
[2] {'p': [2, 3, 5, 6]}
 > [3] {'p': [2, 3, 5]}
[3] {'p': [2, 3, 5]}
$ python3 sieve.py -l
[0] {'p': [2, 3, 4, 5, 6]}
@ t = {'n': 4, 'd': 2}
 - {'p': [4]}
 + {}
 > [1] {'p': [2, 3, 5, 6]}
@ t = {'n': 6, 'd': 2}
 - {'p': [6]}
 + {}
 > [2] {'p': [2, 3, 4, 5]}
@ t = {'n': 6, 'd': 3}
 - {'p': [6]}
 + {}
 > [2] {'p': [2, 3, 4, 5]}
[1] {'p': [2, 3, 5, 6]}
@ t = {'n': 6, 'd': 2}
 - {'p': [6]}
 + {}
 > [3] {'p': [2, 3, 5]}
@ t = {'n': 6, 'd': 3}
 - {'p': [6]}
 + {}
 > [3] {'p': [2, 3, 5]}
[2] {'p': [2, 3, 4, 5]}
@ t = {'n': 4, 'd': 2}
 - {'p': [4]}
 + {}
 > [3] {'p': [2, 3, 5]}
[3] {'p': [2, 3, 5]}
```

Fig. 2. Using the generated code as a program. Note that markings numbering may differ between executions (see, *e.g.*, markings 1 and 2).

2.5 Using the generated library from another tool

The most general use one can make of the code generated by ZINC is to call it from another program or library. Indeed, the result of the compilation is not only a program, it is also a library. For instance, file “`sieve.py`” we have generated above could be used from another Python module with just an “**import** sieve”. All the generated libraries have a consistent API (however, exact names may vary depending on the target language, for instance in Go, they start with an uppercase character):

- a function init() returns the initial marking;
- a function succ(m) returns the successors of a marking m;
- for each transition, a fonction succ_123(m) returns the successors of m for this particular transition. These functions are numbered automatically and a table succfunc maps the transitions names to these functions (transition names are not expected to be valid identifiers and thus cannot be used in the functions names);
- other functions are addsucc(m, s) and addsucc_123(m, s) that add the successors of m to a set s. The set data structure is declared in a library that is imported in the generated module;

– finally, iterators itersucc(m) and itersucc_123(m) are available to access more precisely to the details of firings: instead of returning a set of successors, they yield for each one a structure with the name t of the transitions, the mode b (that is, a binding of the variables used in its annotations), the marking sub to be subtracted from m, and the marking add to be added to m so that m − sub + add is the successor of m through transition t using mode b;
– high-level functions statespace, lts, and main are also provided to implement the command line interface of the generated code.

ZINC is designed with the aim to generate human readable code, so it is easy to read the code it produces, and this is currently the best source of information about it. In particular, functions statespace and lst are realistic yet simple examples of how the API should be called.

3 Implementation

As a Python library, ZINC is organised as follows:

– zinc	the main package	(2,890 LOC)
– zinc.arcs	arcs labels classes	
– zinc.nodes	places and transitions classes	
– zinc.tokens	markings and tokens classes	
– zinc.nets	Petri nets class	
– zinc.data	auxiliary data structures	
– zinc.io	import/export from/to various file formats	
– zinc.io.zn	ZINC's own format	
– zinc.cli	implementation of command line tool "zn"	
– zinc.compil	compilation engine	
– zinc.compil.coffee	CoffeeScript backend	(372 LOC)
– zinc.compil.go	Go backend	(493 LOC)
– zinc.compil.python	Python backend	(481 LOC)

As with SNAKES, the user usually only need to import zinc.nets module that itself imports all the necessary to work with Petri nets. At the time we write these lines, ZINC represents less than 3k lines of Python code (2,890 LOC to be compared with about 82k LOC for SNAKES). This size includes the backends, each representing less than 500 LOC in Python (to be compared with the 13k LOC of Neco), and the definition of multisets, markings, sets of markings, and the statespace explorations algorithms in Python. For the other target languages, these data structures represent: 827 LOC for CoffeeScript, and 1331 LOC for Go (one third of which being just "`}`"). This shows that implementing a new backend requires only a limited effort.

3.1 How ZINC compiles nets

When method PetriNet.build is called, it computes an abstract syntax tree (AST) of a module that defines the various functions we have listed previously. This

AST is not related to any programming language in particular and has high-level constructs directly related to Petri nets, like testing the presence or absence of a specific token in a specific place. Then, the AST is passed to the appropriate backend that is responsible for producing the actual code. To do so, it performs a traversal of the AST and each kind of node triggers a corresponding method to generate a piece of code. This is illustrated in Figure 3 where we show the AST for the addsucc function for the unique transition of the sieve model, together with its translation into Python. Not all AST nodes are as simple to translate, but the difficult ones have been implemented for three languages already, so that it should be easy to adapt the existing solutions.

AST

```
1  DefSuccProc(trans="t", marking="marking", succ="succ")
2      Declare(d="int", n="int")
3      ForeachToken(place="p", variable="n")
4          ForeachToken(place="p", variable="d")
5              If("n␣%␣d␣==␣0")
6                  IfType(var="d", type="int", )
7                      IfEnoughTokens({"p": ["d", "n"]})
8                          AddSucc(sub={"p": ["n"]}, add={})
```

Python code

```
1  def addsucc_001 (marking, succ):
1      "successors␣of␣'t'"
3      for n in marking("p"):
4          for d in marking("p"):
5              if n % d == 0:
6                  if isinstance(d, int):
7                      test = Marking({"p": mset([d, n])})
7                      if test <= marking:
8                          sub = Marking({"p": mset([n])})
8                          succ.add(marking - sub)
```

Fig. 3. Top: the AST generated for one of the functions corresponding to the transition in the sieve model. Bottom: the corresponding Python code, with line numbered by the corresponding node in the AST.

3.2 Benchmarking ZINC vs SNAKES vs Neco

We have exercised the libraries generated by ZINC on two models in order to evaluate how far the current preliminary implementation is from an optimised net compiler like Neco. We have chosen two models from the *model-checking contest* [7]: “Eratosthenes’ sieve” and “Refendum” are two small coloured Petri nets with rapid combinatorial explosion. It is interesting to consider such small nets in order to restrict as much as possible the overhead of querying a Petri net structure that would impair SNAKES’ performance compared to compiled

approaches. In other words, using these models, we focus the comparison on the cost of firing transitions. We have compared five methods to compute statespaces: using SNAKES, using Neco with the Python backend, using ZINC with each of its backends. Unfortunately, Neco with the Cython backend is crashing on a faulty memory access, but as we will see later on, it would probably not have given more information. Figure 4 shows the result we obtained by setting a timeout of 30 seconds (*i.e.*, a computation longer than 30s is interrupted). We do not report compilation times that are negligible and comparable for all tools (note that the Go backend also has a compilation to machine code that is very fast as well, which was one of the reasons to chose Go).

Let us first consider the Python-based runs: Neco is faster than ZINC's Python backend, which itself is faster than SNAKES. The relative performance of these engines are depicted in Figure 5 that shows that Neco is typically 2–3 times faster than ZINC, which itself is 2–8 times faster than SNAKES. An interesting observation in that all these tools have the same profile: improving of rates as statespaces grow until a performance peak after which rates progressively amortise. The first phase corresponds to cases where launching the tool dominates its execution time, while the second phase is due to the increasing cost of managing larger sets of markings.

These relative performance show that there is room for improving ZINC, which is not surprising since we implemented none of the optimisations from Neco. Indeed, we discovered that Neco lacks some checks, which makes it faster, but

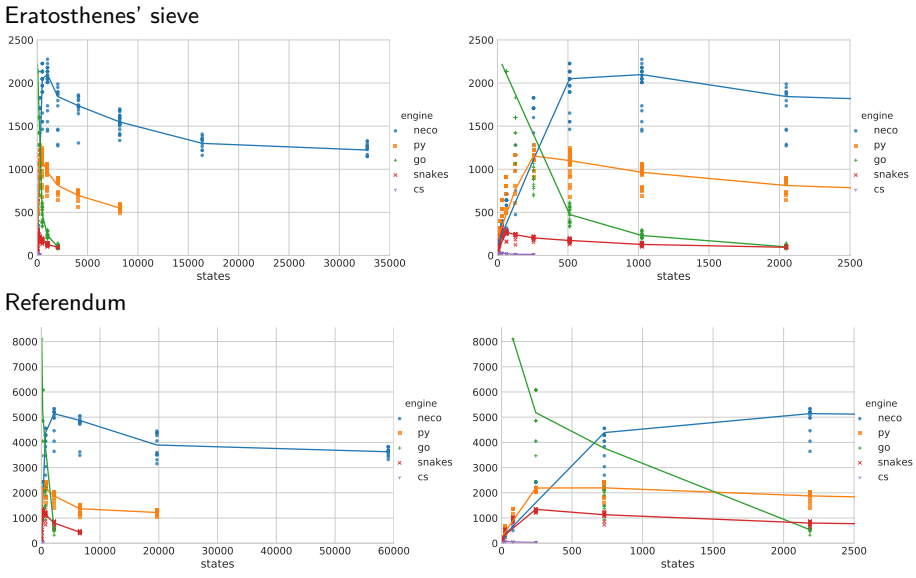

Fig. 4. Firing rates (number of transitions fired by seconds) with respect to the number of states in the models for the two considered models. On the left, the full range of measures we obtained. On the right, a zoomed view for the smallest statespaces.

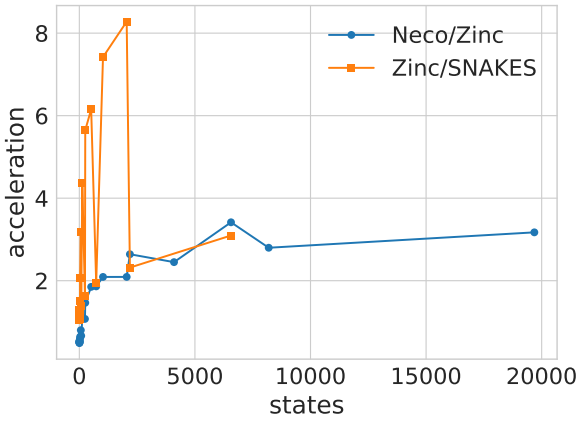

Fig. 5. Relative performances of Neco vs ZINC/Python, and ZINC/Python vs SNAKES, where acceleration is computed as the rate of one tool divided by the rate of the other tool as indicated in the legend.

also make it incorrect in some cases (in particular, Neco does not check the type of produced tokens). So, optimising the generated code like Neco does will require a careful work to keep it correct in every case.

Considering the Go backend, we observe it starts directly with the second phase where rate decreases (more and faster) with the number of states. This shows that the data structure we have used to store statespaces is highly inefficient (probably there are a lot of collisions in the underlying hashtable). So, we first need to fix this and only then it will be interesting to compare ZINC's Go backend to Neco's Cython backend (hoping it can fixed).

Finally, CoffeeScript exhibits very low performances, which can be explained both by the language itself (this is JavaScript in the end), and by the implementation of statespace data structure that mimics that of Go, and thus probably has the same performance issues.

4 Conclusion and roadmap

We have presented ZINC that is the code name for a complete rewrite of SNAKES grounded on models compilation like Neco does. ZINC substantially improves both SNAKES and Neco in various ways, in particular:

- it is a cleaner and more modern tool. The library is much smaller, and even with all SNAKES features ported, it is expected to remain as such because many code in SNAKES is not needed anymore in ZINC, while the new code dedicated to compilation remains quite small;
- it provides efficient executions out-of-the-box. We have shown that while ZINC does not implement the optimisations from Neco, the latter is only

twice as fast. Moreover, we have discovered that some of the optimisations in Neco are not always correct;
– it supports a more general class of Petri nets annotated with potentially any programming language. Nets can be currently annotated and compiled to Python, Go, and CoffeeScript.

ZINC is in its early phases of development but we have shown how it can be used already to define and compile Petri nets, either to explore their statespace directly, or to execute them from another program or library. We have also discussed ZINC's implementation, its architecture, and its approach to model compilation.

Next steps will be to progressively port to ZINC the missing features from SNAKES, in particular the various plugins and the extensive API documentation. We also plan, to introduce a plugin that would automate the compilation of Python-annotated Petri nets and hide the small API changes to provide a compatibility layer with SNAKES (*i.e.*, ZINC with this plugin could be used as a drop-in replacement of SNAKES). We shall also add long-term missing features from SNAKES like in particular the capability of importing/exporting Petri nets and statespaces from commonly used formats. In parallel, we will progressively introduce in ZINC the optimisations from Neco in order to improve execution performances. This must be carried with care because we must ensure that generated code remains correct in every circumstances, which is currently not the case for Neco. Considering performance, we must also improve the Go backend in order to fix its current performance issues, which is more a question of Go programming than a problem related to ZINC itself.

In the process, we hope to attract users (who will be able to report problems, request features, and help to improve the tool), and hopefully contributors that could implement backends suited to their needs.

References

1. Ashkenas, J., et al.: CoffeeScript. `http://coffeescript.org`
2. Best, E., Devillers, R., Koutny, M.: Petri net algebra. Monographs in Theoretical Computer Science. An EATCS Series, Springer (2001)
3. Cooley, S.: Install the Windows subsystem for Linux. `http://msdn.microsoft.com/en-us/commandline/wsl/install-win10`
4. Fronc, L.: Neco net compiler. `http://github.com/Lvyn/neco-net-compiler`
5. Fronc, L., Pommereau, F.: Building Petri nets tools around Neco compiler. In: Proc. of PNSE'13. vol. 989. CEUR-WS (2013)
6. Google inc.: The Go programming language. `http://golang.org`
7. Kordon, F., et al.: the Model Checking Contest at PETRI NETS. `http://mcc.lip6.fr`
8. Pommereau, F.: SNAKES is the net algebra kit for editors and simulators. `http://snakes.ibisc.univ-evry.fr`
9. Pommereau, F.: SNAKES: a flexible high-level Petri nets library. In: Proc. of PETRI NETS'15. LNCS, vol. 9115. Springer (2015)
10. Python Software Foundation: The Python programming language. `http://www.python.org/`