Generating code automatically with Petri nets

Juliette Fournis d'Albiat
student in Master 1 Jacques Herbrand
ENS Paris Saclay

August 2020

This is an internship report that has been supervised by Michael Blondin, professor at the Université de Sherbrooke

1 Introduction

The SyPet application [5] is a program synthesis tool for Java libraries that automatically constructs programs by composing APIs. SyPet application is written in Java, it uses Petri net to encode large Java libraries for code synthesis. The core issue of the synthesis procedure is the problem of legal firing sequences generation in the Petri net. Reachability and legal firing sequences generation are well known theoretical problems. However, by now this problem is hard to solve for general Petri nets. Nevertheless, Petri nets that are generated in SyPet have a particular shape. In this work, $glcf$-Petri nets are defined. $glcf$-Petri net captures SyPet Petri net specificity. They are studied theoretically, in particular a characterization of reachability is given. Finally, an algorithm that uses the results is presented and implemented.

1.1 Contribution Overview

The aim of our work was to provide a new algorithm for SyPet application [5]. This paper makes the following contributions:

- In section 2, the main definitions about Petri net are given.
- In section 3, SyPet is presented. SyPet is the framework of our work, since the purpose of our work has been to improve SyPet's performance thanks to a theoretical work on Petri net.
- In section 4, the theoretical work on which our algorithm is based is presented.
- In section 5, the algorithm is presented and explained briefly.

1.2 Related Work

Our goal is to develop an application able to generate code automatically thanks to Petri nets. This problem has been previously explored. We present some works on the subject

- SyPet [5] is implemented in Java. It has been developed to synthesize Java code. It takes as input a set of Java libraries, a query type, and a set of test cases. It returns a Java piece of code that uses functions of the provided libraries. SyPet uses Petri net as the underlying data structure to express the search space.
- Similarly to SyPet, CodeHint [6] takes as input a method signature and test cases. It can also take snippets of code with hols. It performs type inhabitation at runtime, and uses a probabilistic model based on the frequency of function utilization in practice. Then it synthesizes and evaluates code at runtime. The user can add new specifications at runtime to help the code synthesis.
- H+ [11] is implemented in Haskell. It can synthesize Haskell code. Similarly to SyPet [5], it takes as input a set of Haskell libraries and a query type and uses Petri nets. However it does not use test cases. Because Haskell enables the use of higher order type, a main difference between H+ and other approaches is the work on type abstraction.
- OCPet [10] is a program synthesizer for the OCaml language. OCPet is inspired by SyPet, contrary to H+ [11], it does not uses type abstraction and cannot synthesize programs that use higher-order types.

2 Primer on Petri Nets

In this section, Petri net are defined and basic knowledge about them are provided since they will be useful to understand the remainder.

Given a finite alphabet T the symbols T^* and $T^\oplus$ denote the free monoid and the free commutative monoid generated by T respectively. Given a sequence $\sigma \in T^*$, the projection of σ in $T^\oplus$ is defined as the set of transitions that appears in σ. A Petri net is a directed bipartite graph with two types of nodes: places and transitions. Each place in a Petri net contains a number of tokens. Transitions have input ant output edges coming and going into places. These edges (arrows) have a weight which is a natural number. See Figure 1 for a representation of a Petri net.

Definition 2.1 (*Marking*)**.** A marking of a Petri net is a mapping $M\colon P \to \mathbb{N}$ where P is the set of places giving the number of tokens on a place.

Definition 2.2 (*Petri net*)**.** A Petri net $\mathcal{N}$ is a 5-tuple (P, T, E, W, M_0) where P and T are disjoints, P is a set of places, T is a set of transitions, and $E \subseteq (P \times T) \cup (T \times P)$ is the set of edges (arcs), W is a mapping from each edge $e \in E$ to a weight $W \in \mathbb{N}$, and M_0 is the initial marking of $\mathcal{N}$.

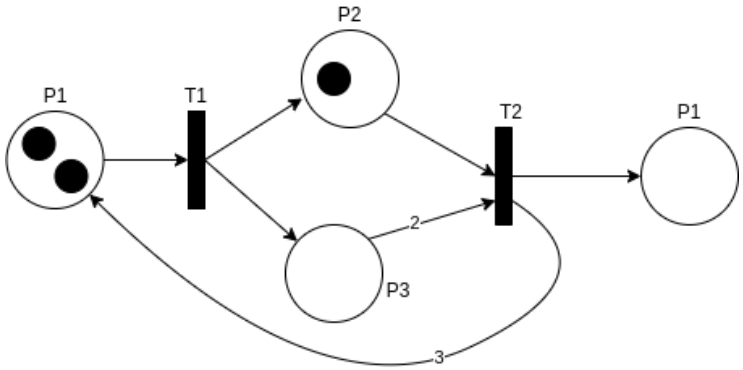

Figure 1: Representation of a Petri net
The places are $\{P1, P2, P3, P4\}$ *(large circles),*
there are two tokens (dark circle) on place $P1$*,*
the transitions are $\{T1, T2\}$ *(dark rectangles), and*
the arcs on which no weight appears have weight one.
The current marking is $[P1 \mapsto 2; P2 \mapsto 0; P3 \mapsto 0; P4 \mapsto 0]$

Definition 2.3 (*Transition System*)**.** A Petri net $\mathcal{N}$ is naturally associated to a transition system K. K is a 4-tuple (S, T, I, Δ) where S is the markings of $\mathcal{N}$, T is the transitions, and $I = \{M_0\}$ the initial marking. Δ is the reachability relation. $\Delta = \{(M, t, M') \mid \forall p \in P : M(p) \geq W(p,t) \wedge M'(p) = M(p) - W(p,t) + W(t,p)\}$ When $(M, t, M') \in \Delta$, we say that t is enabled in M and that its firing produces the successor marking M' which we also write $M \xrightarrow{t} M'$. Given a sequence $\sigma = t_0 \cdots t_f \in T^*$, we write $M \xrightarrow{\sigma} M'$ for $M \xrightarrow{t_0} \cdots \xrightarrow{t_f} M'$ and $M \xrightarrow{X} M'$ if $X \in T^{\oplus}$ and $\exists \sigma \in T^*$ s.t. $\Phi(\sigma) = X$.

Let p be a place and let t be a transition, we say that p is an input (resp output) place of t if $(p,t) \in F$ (resp $(t,p) \in F$). The set ${}^\bullet a = \{a' \mid (a', a) \in F\}$ is called the pre-set of a, and the set $a^\bullet = \{a' \mid (a, a') \in F\}$ is its post-set. We extend the notation to sets A by ${}^\bullet A = \bigcup_{a \in A} {}^\bullet a$ and $A^\bullet = \bigcup_{a \in A} a^\bullet$

Definition 2.4 (*Parikh Vector* $\Phi(\sigma)$)**.** Given a sequence $\sigma \in T^*$, the Parikh vector of σ is defined as the projection $\Phi(\sigma)$ of σ on $T^{\oplus}$.

For a set $X \in T^*$, we say that X is enabled in $\mathcal{N}$ at initial marking M_0 if there exists a sequence fireable in $\mathcal{N}$ at M_0 whose Parikh vector is X. We note it $M_0 \xrightarrow{X} M$ where M is the marking obtain after the firing of such sequence.

Given a marking M, (resp a Parikh vector X), the support of M (resp of X) is defined as $Supp(M) = \{\text{place } p \mid M(p) > 0\}$ (resp $Supp(X) = \{\text{transition } t \mid X(t) > 0\}$).

Definition 2.5 (*Induced Graph*)**.** A Petri net $N = (P, T, E, W, M_0)$ is associated to a graph $G = (E, V)$, with $E = P \cup T$ and $V = \{(p,t) \in P \times T \mid W(p,t) \neq 0\} \cup \{(t,p) \in T \times P \mid W(t,p) \neq 0\}$.

Given a Petri net $N = (P, T, E, W, M_0)$ and a set of transition $\tilde{T}$, the subnet induced by $\tilde{T}$ is defined as $N_{\tilde{T}} = (\tilde{P}, \tilde{T}, \tilde{E}, \tilde{W}, \tilde{M_0})$ where $\tilde{P} = {}^\bullet\tilde{T} \cup \tilde{T}^\bullet$ and $\tilde{E}, \tilde{W}, \tilde{M_0}$ are the restriction of E, W, M_0 to $\tilde{T}$ and $\tilde{P}$.

Well-known problems on Petri net are

- The Reachability Problem: Given a Petri net $\mathcal{N}$, is there a sequence of valid execution steps that reaches a given final marking.
- RecLFS (Recognize Legal Firing Sequence) : Given a Petri net $\mathcal{N}$ and a vector $X \in \mathbb{N}^T$, is X enabled in $\mathcal{N}$? A well-known necessary condition for the RecLFS problem is the marking equation, for all place p, $M_0(p) + \Sigma_{t \in T}(W(p,t) - W(t,p)) \cdot X(t) = M(p)$.

3 The Program Synthesis Approach of SyPet

The goal of this work, is to improve SyPet's performance thanks to a theoretical work on Petri nets. This section aims at giving a quick overview of SyPet approach to automatically generate code using Petri nets.

3.1 Global overview

The synthesis procedure of [5] takes a method signature S, a set of components Λ, and test cases E. Its output is either $\perp$, meaning that the specification cannot be synthesised using components Λ, or a well-typed program, loop-free and without conditionals, i.e. the program consists only on sequential function calls, that passes all test cases E. The procedure constructs a Petri net $\mathcal{N}$ where each transition is a component $f \in \Lambda$ and each place corresponds to a type. There is an arc in the Petri net from τ to f with weight w, component f takes w arguments of type τ. The method signature defines the initial and final markings, arguments types correspond to the initial marking and the return type corresponds to the final marking. The functions of the code synthesised by SyPet are given by a firing sequences from M_0 to M in the constructed Petri net.

3.2 An Example of a Task to Synthesise

Consider the small library and the function `write` represented in tabular 1 and Listing 1. Test cases are represented in Listing 2.

Listing 1: code to synthesise of the function `write`

```
void write(Document doc, char a, int n) {
    doc.move(n);
    doc.write(a);
}
```

Table 1: Methods and Constructors of Class Document in the library java.doc

Type	Constructor
`Document`	`Document()`
Type	**Method**
`char`	`read()`
`void`	`append(char c)`
`void`	`move(int n)`
`int`	`size()`

Listing 2: Tests cases on `write`

```
public static boolean test() throws Throwable {
    Document doc = new Document();
    write(doc,0,'a');
    doc.move(0);
    return doc.read() == 'a';
}
```

Suppose that we want to use SYPET in order to synthesise the function `write` and obtain the code shown in Listing 1. Then we should give it the `write`'s method signature

```
void write(Document doc, char a, int n)
```

and the library in tabular 1, SYPET will generate a Petri net whose sketch is represented in Figure 2. Then SYPET will try to find a sequence from the initial marking $M_0 = \{Document: 1, char: 1, int: 1\}$ to the final marking $M = \{void: 1\}$. The Petri net represented in Figure 2 is just a sketch of the Petri net constructed by SYPET. Indeed in Java langage :

- variables can be used several times. A solution is for each place p, to add a transition that increases number of tokens on place providing that there is already tokens. In the following, this transitions are called κ-transition.

  Definition 3.1 (κ-transition)**.** Using the terminology of [5] a κ-transition t is associated to a place p. The place p is the post-set and the pre-set of t. Its incoming arc has weight 1, and outgoing arc, weight 2.

- variables and arguments which are produced are not always used. A solution is to add transitions to each places, which consume one token and produce nothing. For a place p, we name this transition $kill_p$.

- place void is special since it can be produced freely. In order to capture this specificity, place void is always added in initial markings.

Our synthesis problem amounts to the following: *Given the Petri net $\mathcal{N}$ of Figure 2, the initial marking $M_0 = \{Document : 1, char : 1, int : 1\}$ and final*

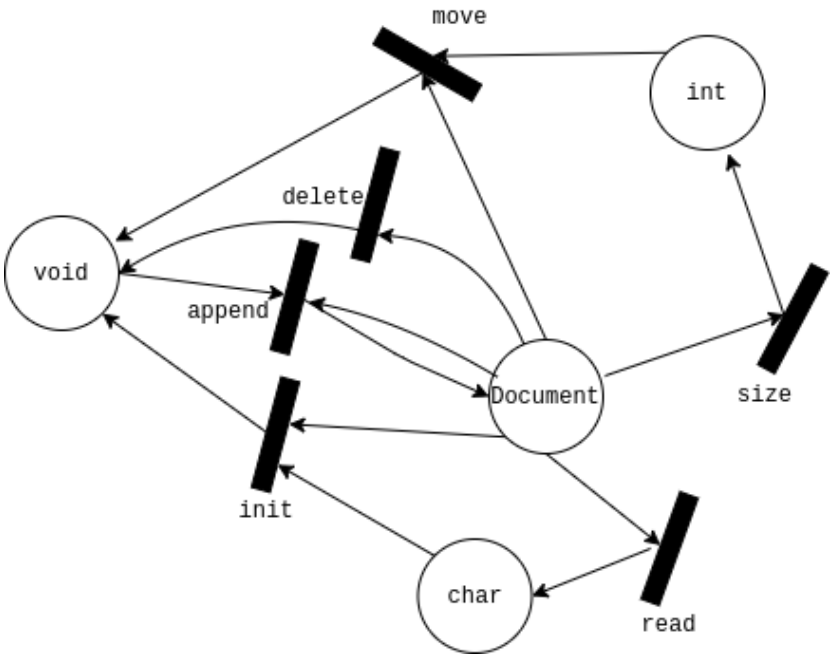

Figure 2: A sketch of the Petri net associated to the java.document library. *The init function correspond to the constructor of Class Document.*

marking $M = \{void : 1\}$*, find* σ *such that* $M_0 \xrightarrow{\sigma} M$ *and the program obtained from* σ *satisfies* E *(shown in Listing 2).* In this case, there are solutions. One of them is $\sigma = \kappa_{Document} \cdot read \cdot move$, where $\kappa_{Document}$ is the κ-transition associated to place *Document*.

Finally, to obtain a well-formed program, SyPet removes special transitions κ and *kill*. Given the functions of solution σ and their signatures, it is possible to generate a sketch of code with holes at variable places. With logical formulas, every possible solution (modulo renaming of the variables) are generated and test cases are compiled at run-time.

4 A Theoretical Background for the Synthesis Procedure

In this Section we give useful definitions and properties in order to develop new efficient algorithms for SyPet's synthesis procedure. In Section 3, it has been seen that a key step to generate code automatically is to give a solution to the following problem:

Problem 4.1. Given a Petri net $\mathcal{N}$, initial and final markings M_0 and M, find σ such that $M_0 \xrightarrow{\sigma} M$.

In general this problem is non-elementary and more precisely TOWER-hard [2] and solvable in Ackermaniann time [9]. However, it can be observed that the Petri net generated by SyPet have a particular shape, that we define in the following.

4.1 cf and $glcf$-Petri nets

Definition 4.1 (Communication-Free Petri Nets or cf-Petri net)**.** A Petri net $\mathcal{N} = (P, T, W)$ is communication-free if $|{}^\bullet t| = 1$ for every $t \in T$, and $W(p, t) \leq 1$ for every $p \in P$ and every $t \in T$.

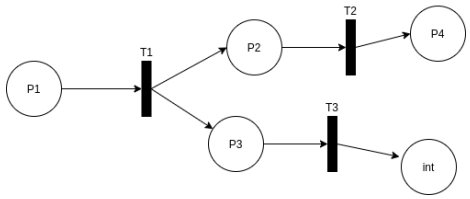

Figure 3: A Communication-free Petri Net

cf-Petri net are well-known in the literature. They correspond to Petri net generated by SyPet if we remove the κ-transitions and inverse every arrow direction.

Definition 4.2 (Guarded-Lossy-Communication-Free Petri net or $glcf$-Petri net)**.** A net $\mathcal{N}' = (P, T', W')$ is *guarded-lossy-communication-free* if $\mathcal{N} = (P, T, W)$ is a communication-free Petri Net where

- $T' = T \cup K$ where $K = \{\kappa_s | s \in S\}$ is a set of transitions called κ-transitions that removed one token on a given place, provided there is already one, i.e.: $W'(s, \kappa_s) = 2$, $W'(\kappa_s, s) = 1$ and for each place s' distinct of s, $W'(s', \kappa_s) = 0 = W'(\kappa_s, s')$.
- mappings W' and W coincides on W's domain.

As far as we know, $glcf$-Petri nets have not already been studied in the literature. They exactly correspond to Petri net generated by SyPet application if we inverse every arc direction. In the following, it will be seen that they are a subclass of ω-Petri net (define in [7]). Moreover, we present some useful properties for solving problem 4.

4.2 $glcf$-Petri net and ω-Petri net are equivalents

Definition 4.3 (ω-Petri net)**.** A Petri net with ω-arcs (ω-Petri net) is a tuple $\mathcal{N} = (P, T, W)$, where P is a finite set of places, T a finite set of transitions and each transition $t \in T$ incoming arcs and outgoing arcs weight can take arbitrary

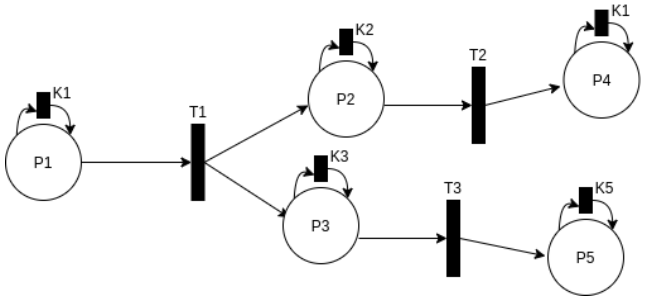

Figure 4: A Guarded-Lossy-Communication-Free Petri Net

value in $\mathbb{N}$, where ω can take an arbitrary strictly positive value among natural numbers.

Proposition 4.1 (The ω-Petri net are equivalents to $glcf$-Petri net)**.** Given a ω-Petri net $\mathcal{N}$ and a $glcf$-Petri net $\mathcal{N}'$, we say that $\mathcal{N}$ is equivalent to $\mathcal{N}'$ if for every pair of marking (M_0, M), there exists σ such that $M_0 \xrightarrow{\sigma} M$ in $\mathcal{N}'$ iff there exists σ' such that $M_0 \xrightarrow{\sigma'} M$ in $\mathcal{N}'$. If $\mathcal{N}$ is a $glcf$-Petri net then there exists a ω-Petri net $\mathcal{N}'$ such that $\mathcal{N}$ is equivalent to $\mathcal{N}'$ and reciprocally.

4.3 Reachability characterization for $glcf$-Petri nets

In [4], Esparza proposes a characterization of the reachability relation of communication-free Petri nets, which consists in the following equivalence.

Theorem 4.1 (cf-Petri Net Reachability)**.** *Let ($\mathcal{N}$,M_0) be a communication-free Petri net with a set T of transitions, and let $X \in T^{\oplus}$. There exists a sequence $\sigma \in T^*$ such that $M_0 \xrightarrow{\sigma} M$ and $\mathcal{P}(\sigma) = X$ iff*

(a) $M_0(p) + \sum_{t \in T}(W(t,p) - W(p,t)) \cdot X(t) = M(p)$ *for every place p of $\mathcal{N}$*

(b) every place of $\mathcal{N}_X$ is markable from M_0 in $\mathcal{N}_X$

We provide a characterization for $glcf$-Petri net reachability.

Theorem 4.2 ($glcf$-Petri net reachability)**.** *Let $\mathcal{N} = (P, T, W)$ be a $glcf$-Petri net, $\mathcal{N}' = (P, T', W')$ be the cf-Petri net obtained by removing κ-transitions from $\mathcal{N}$ and M_0, M be some marking from $\mathbb{N}^P$. Then the following the statements are equivalents*

(a) X is such that $M_0 \xrightarrow{X} M$

(b) there exists a vector X' and a marking M' greater or equal than M (as a function of $\mathbb{N}^P$) and having the same support than M such that $M_0 \xrightarrow{X'} M'$. Moreover we can also ensure that X and X' only differs on κ-transitions.

In the following we prove Theorem 4.2.

Definition 4.4 (The rewriting mapping $\xrightarrow{p}$)**.** Given a place p, let us define the mapping $\xrightarrow{p}$ as : $\begin{array}{ccc} K(^\bullet p) & \rightarrow & TK^* \\ k_p t_p & \xrightarrow{p} & t_p^2 k_1^{\alpha_1} \cdot \cdots \cdot k_n^{\alpha_n} \end{array}$

where

- K is the set of κ-transitions
- T is the set of transitions
- for every $i \in \{1, ..., n\}$, k_i is the κ-transition associated to p_i and $\alpha_i = W(t_p, p_i)$
- and $\{p_1, ..., p_n\} = t_p^\bullet$.

The following shema illustrates the mapping $\xrightarrow{p}$.

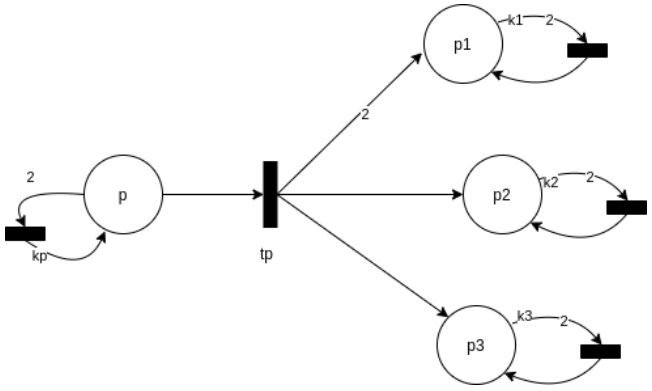

Figure 5: Illustration of $\xrightarrow{p}$
In the figure above $k_p t_p \xrightarrow{p} t_p^2 k_1^2 k_2 k_3$

Proposition 4.2 (effect preservation)**.** If $k_p t_p \in K(^\bullet p)$ is such that $k_p t_p \xrightarrow{p} u$ then $\Delta(k_p t_p) = \Delta(u)$.
Indeed,

- for p :

  – $\begin{aligned} \Delta(k_p t_p)(p) &= -2W(t_p, p) \\ &= -2 \\ &= -W(t_p, p) + (W(p, k_p) - W(k_p, p)) \\ &= \Delta(u)(p) \end{aligned}$ if $p \notin p_1, ..., p_n$

  – $\begin{aligned} \Delta(k_p t_p)(p) &= 2(-W(t_p, p_i) + W(t_p, p_i)) - \alpha_i(W(p_i, k_i) - W(k_i, p_i)) \\ &= -2 + 2W(t_p, p_i) - \alpha_i \\ &= -2 + W(t_p, p_i) \\ &= \Delta(u)(p) \end{aligned}$
  if $p = p_i \in p_1, ..., p_n$

- for $p_1, ..., p_n$:
  $\begin{aligned} \Delta(k_p t_p)(p_i) &= 2W(t_p, p_i) - \alpha_i(W(p_i, k_i) - W(k_i, p_i)) \\ &= 2W(t_p, p_i) - \alpha_i \\ &= W(t_p, p_i) \\ &= \Delta(u)(p_i) \end{aligned}$ if $p_i \neq p$

Proposition 4.3 ($\xrightarrow{p}$ preserves fireability)**.** Given a Petri net (M_0, N), and a marking M, let $k_p t_p \in K(^\bullet p)$ be such that $k_p t_p \xrightarrow{p} u$, if $k_p t_p$ is fireable in (M, N), then u is fireable in (M, N)

Definition 4.5 (The rewriting mapping $\xrightarrow{t}$)**.** Given a transition t such that t has one input place p, define the mapping $\xrightarrow{t}$ as :
for a word $u \in T^*$, $ut \xrightarrow{t} u' k_p^{\beta_p} t$ where T' is the set of transitions of T which are not κ-transition. u' is the projection of u on T' of and β_p is the number of transitions we removed from u to obtain u'.
In other words, $\xrightarrow{t}$ removes the transition of K_p in u and rewrites them just before t.

Proposition 4.4 ($\xrightarrow{t}$ preserves fireability and Δ)**.** The fact that $\xrightarrow{t}$ preserves Δ is clear.
If ut is fireable in (M, N) then

- u' is fireable in (M, N) since u is fireable because the action of removing κ-transition preserves fireability.

- $\begin{aligned} (M + \Delta(u'))(p) &= (M + \Delta(ut))(p) + \beta_p + 1 \\ &\geq \beta_p + 1. \end{aligned}$

- t is fireable in $M + \Delta(u' k_p^{\beta_p}) = M + \Delta(u)$

With this properties, we are able to give a proof of Theorem 4.2.

Proof.

- $(a) \Rightarrow (b)$ Suppose we are given a reachable marking M. Let σ be such that $M_0 \xrightarrow{\sigma} M$ in N and σ' the projection of σ on T'. Then for a sequence u, write M_u for the marking $M_0 + \Delta(u)$.
  Now let $\Theta_\sigma = \{p \mid p \in Supp(M_{\sigma'}) \setminus Supp(M)\}$.
  - If $\Theta_\sigma = \emptyset$, $M' = M_0 + \Delta(\sigma')$ satisfies (b) and σ' is fireable, so we are done.
  - Otherwise consider p in $\Theta_{\sigma'}$ and t_p the last transition of σ such that $t_p \in {}^\bullet p \setminus p^\bullet$.
    First this transition exists :
    $M(p) = 0$ and $M_{\sigma'}(p) \neq 0$, then $\sigma \cap K_p \neq \emptyset$ and ${}^\bullet p \cap p^\bullet$ is not empty. So we can write $\sigma = u\tilde{t}v$ with $u \in T^*$, $\tilde{t} \in {}^\bullet p \cap p^\bullet$ and $v \in (T \setminus ({}^\bullet p \cap p^\bullet))^*$ and t_p exists.
    Since $M(p) = 0$ and $(M_0 + \Delta(u\tilde{t}))(p) \geq 1$, $\Delta(v)(p) = (M - (M_0 + \Delta(u\tilde{t})))(p) < 0$ thus $\exists t \in v$ such that $\Delta(t)(p) < 0$ and $t \in {}^\bullet p \setminus p^\bullet$.
    Then rewrite σ as $\sigma = ut_pv$ and define σ'' as $\sigma'' = u't_pwv$ with $ut_p \xrightarrow{t_p} u'k_p^\beta t_p \xrightarrow{p}{}^* u't_pw \in (T \setminus K_p)t_p(T \setminus K_p)$. With the propositions above, we know that σ'' is fireable iff σ is fireable and $\Delta(\sigma'') = \Delta(\sigma)$. Thus we can iterate by replacing σ by σ'' while $\Theta_{\sigma'} \neq \emptyset$.
    To ensure that the rewriting will terminate, let's define $\Pi_{\sigma'} = \{p \mid {}^\bullet p \setminus p^\bullet \cap \Phi(\sigma') \neq \emptyset\}$. We can order $\Pi_{\sigma'}$ by $p \prec p'$ iff t_p appears before $t_{p'}$ in σ'. (For $p \in P$, t_p is the last transition of p in $({}^\bullet p \setminus p^\bullet) \cap \sigma'$). And always choosing for p the place which maximizes $\Pi_{\sigma'}$ among the ones which satisfy the properties above. This ensures that the rewriting will terminates in at most $|\{p \mid {}^\bullet p \setminus p^\bullet \neq \emptyset\}|$ steps.
    At the end of the iterations, we obtain a path $\tilde{\sigma}$ such that $\Theta_{\tilde{\sigma}} = \emptyset$, $\tilde{\sigma}$ is fireable in (M_0, N) and $\Phi(\tilde{\sigma}) \setminus \Phi(\sigma) \subset K$ thus $\Delta(\tilde{\sigma}) < \Delta(\sigma)$ and $M_{\tilde{\sigma}}$ satisfies (b).

- $(b) \Rightarrow (a)$ Let M' be a marking satifying (b), and consider a sequence $\sigma' \in T'^*$ such that $M_0 \xrightarrow{\sigma} M'$. Let $p_1, ..., p_n$ be the places such that $\alpha_i = (M' - M)(p_i)$ is positive. Now consider $\sigma = \sigma' \cdot k_1^{\alpha_1} \cdots k_n^{\alpha_n}$.
  - first we have
    $$\begin{aligned} M_0 + \Delta(\sigma) &= M' + \Delta(k_1^{\alpha_1} \cdots k_n^{\alpha_n}) \\ &= M' - (M' - M) \\ &= M \end{aligned}$$
  - also we know that σ is fireable in (M_0, N) since σ' is fireable in (M_0, N) and
    for every $i \in \{1, ..., n\}$
    $$\begin{aligned} (M_0 + \Delta(\sigma' \cdot k_1^{\alpha_1} \cdots k_i^{\alpha_i}))(p_i) &\geq (M_0 + \Delta(\sigma' \cdot k_1^{\alpha_1} \cdots k_n^{\alpha_n}))(p_i) \\ &= M(p_i) \geq 1 \text{ since } p_i \text{ belongs to } Supp(M) \\ &= (M_0 + \Delta(\sigma' \cdot k_1^{\alpha_1} \cdots k_{i-1}^{\alpha_{i-1}}))(p_i) \geq \alpha_i + 1 \end{aligned}$$
    thus $k_i^{\alpha_i}$ is fireable in $(M_0 + \Delta(\sigma' \cdot k_1^{\alpha_1} \cdots k_{i-1}^{\alpha_{i-1}}), N)$.

In conclusion $M' = M + \Delta(\sigma')$ satisfies (a).

□

4.4 $glcf$-Petri nets and pc-grammars

In the literature, it is well-known that cf-Petri nets and commutative context-free grammars are equivalent (see for example [4] and [8]). Similarly, we found an relation between $glcf$-Petri nets and partially commutative grammar (defined in [1]).

Definition 4.6 (Partially Commutative Grammar or pc-commutive grammar)**.** A pc-grammar is a Greibach context-free grammar equipped with a symmetric and irreflexive relation $I \subseteq V \times V$ called the independence relation and in which two kinds of derivation steps are enabled :

- production step: $X\beta \to \alpha\beta$, for a production $X \to \alpha$
- swap step: $\alpha XY\beta \to \alpha YX\beta$, where X and Y are independent

(In the definition aboves X, Y designate non-terminals and α, β designate terminals).

Proposition 4.5 ($glcf$-Petri net and partially commutative grammar)**.** Given a $glcf$-Petri net $\mathcal{N}$, an initial marking M_0, the set of reachable markings M corresponds to a language that can be generated by a pc-grammar.

5 Description of the algorithms

In this Section, we present the algorithm that we have developed in order to design an efficient synthesis procedure. Our algorithm has been implemented in Java. It requires an SMT solver. We have used the SMT prover Z3 [3]. Z3 has a Java interface. We have tried to test our algorithm on real benchmark such as the ones used by SyPet's developers. However, because of lack of time, results will not be presented in this document. In a first time, the synthesis procedure is presented in a general manner. Then we explain how we have implemented the procedure based on characterization of theorem 4.2.

5.1 General Synthesis Procedure Overview

The synthesis procedure takes as input a Petri net $\mathcal{N}$, initial marking M_0 and final marking M and looks for a sequence σ such that $M_0 \xrightarrow{\sigma} M$ and which leads to a program which satisfies test cases E. First, the synthesis procedure guesses a vector X of T^* satisfying $M_0 \xrightarrow{X} M$. Then from X, it deduces all firing sequences σ such that $\sigma = \Phi(X)$. It maintains a set of vectors Σ which corresponds to vectors that have been already seen. Σ will contain the vectors X that satisfy $M_0 \xrightarrow{X} M$ but which fail on test cases E. The vector X is guessed among the ones that are not in Σ. The costly operation of this procedure is to

guess X not in Σ and satisfying $M_0 \xrightarrow{X} M$. We call the procedure associated to this operation $ObjLFS$.

Problem 5.1 (Find Legal Firing Sequence with Blocking Set)**.** Given a Petri net $\mathcal{N}$, a set of vectors of $T^{\oplus}$ Σ, initial and final markings M_0 and M, find X such that $M_0 \xrightarrow{X} M$ and $X \notin \Sigma$.

The general procedure is defined as algorithm 1.

Algorithm 1 Find σ such that $M_0 \xrightarrow{\sigma} M$ and the code associated to σ satisfies test cases E.

```
1: Σ ← ∅
2: X ← ∅
3: while X ≠ ⊥ do
4:    X ← ObjLFS(N, M_0, M, Σ)
5:    for all σ such that M_0 -σ-> M and Φ(σ) = X do
6:       // all σ satisfying Φ(σ) = X can be found easily with an SMT solver
7:       synthesize the code associated to σ and test it on E
8:    end for
9:    if test succeed then
10:      return σ
11:   else
12:      add X to Σ
13:   end if
14: end while
```

5.2 How to implement $ObjLFS$ efficiently?

5.2.1 SyPet's Approach

The authors of SyPet use logical formulas to encode the conditions $M_0 \xrightarrow{X} M$ and $X \notin \Sigma$ in a set of formulas that are solved with an SMT solver. They use integer variables X_t that represent the value of $X(t)$. More precisely :

- they search for vectors of a given length k and increment k iteratively.
- the condition $X \notin \Sigma$ is expressed as a conjunction of clauses $c_1 \wedge c_2 \wedge ... \wedge c_n$. For each i, between 1 and n, the clause c_i blocks a vector X_{fail}, $c_i = \bigvee_{t \in T}(X_t \neq X_{fail}(t))$.
- the condition $M_0 \xrightarrow{X} M$ is expressed by encoding the states of the transition system (of $\mathcal{N}$). States $M_0, ..., M_k$ are encoded in Presburger arithmetic. Variables of these formulas are boolean variables $x_t^{t_i}$ and $tok_p^{t_i}$ which give the transition t fired at time t_i and the number of tokens in place p at time t_i with $t_i \in \{0, \cdots, k\}$.

5.2.2 Limits of the approach of SyPet

The limits of the approach [5] are

- the procedure looks for paths of increasing length then it prioritizes path of shortest length, however a better criteria than length can be used.
- the number of boolean variables can be very large. It grows linearly with the length of the solution k. There are $((k+1) \times |P|)$ boolean variables $tok_p^{t_i}$, $(k \times |T|)$ variables $x_t^{t_i}$ and $|T|$ variables X_t.
- the clauses c_i representing vector X_{sol} have length $|T|$, and the length of $|T|$ is in the order of 1000. The size of Σ can reach $|\{X \mid |X| = k \wedge M_0 \xrightarrow{X} M\}|$ and this set grows quickly with k.
- the approach does not take advantage of the shape of the underlying Petri nets. Although the reverse of Petri net are $glcf$-Petri net, the procedure uses a general approach. Thus an approach adapted to $glcf$-Petri net may be more efficient.

Besides we find some error in SyPet's author prunning strategy.

5.2.3 A Task That SyPet Failed to Synthesize

SyPet uses a pruning strategy to encode the transition system states. Every sequence of states that assign more than $k+1$ tokens to a place p, where k denotes the maximum weight of any outgoing edge of p are pruned (see [5] for more details). This pruning strategy is based on a wrong statement and we have constructed a counterexample that SyPet fails to synthesize due to this mistake. We briefly present our counter example.

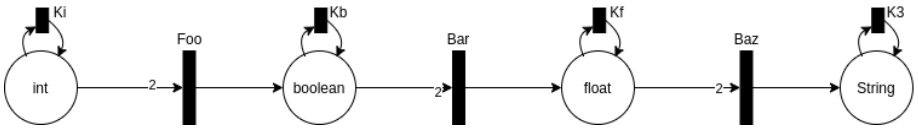

Figure 6: The Petri net associated to the library Foo.

Although the path foo^4bar^2baz is a valid path, SyPet will fail to synthesize it because according to the pruning strategy, the maximum number of tokens which can be obtained in a place which was empty in M_0 is 3.

We have compiled the small library containing the three functions of Listing 3, and we have created the test cases of Listing 5. The test produces the expected result, SyPet does not manage to produce a function which satisfies the test cases.

Listing 3: Foo library

```
public class Foo {
```

```
    private static int fail = 0;
    private static boolean success = false;

    public static float foo(int x,int y) {
        fail++;
        return 0;
    }

    public static boolean bar(float x,float y) {
        if (fail == 4) { success = true; }
        fail = 0;
        return true;
    }

    public static String baz(boolean x, boolean y) {
        if (success) { return "success"; }
        else { return "failure"; }
    }
}
```

Listing 4: Function FooBarBaz that should be synthetised

```
void FooBarBaz(int x0,int x1,int x2,int x3,int x4,int
   x5,int x6,int x7) {
    float y0 = foo(x0,x1);
    float y1 = foo(x2,x3);
    float y2 = foo(x4,x5);
    float y3 = foo(x6,x7);
    boolean z0 = bar(y0,y1);
    boolean z1 = bar(y2,y3);
    String res = baz(z0,z1);
    return res;
}
```

Listing 5: test cases for FooBarBaz

```
public static boolean test() throws Throwable {
    if (FooBarBaz(1,1,1,1,1,1,1,1) == "success") {
       return true; }
    else { return false; }
}
```

5.2.4 A procedure to find sequence in cf-Petri net

Define the procedure $cfObjLFS$ (resp $glcfObjLFS$) as the procedure $ObjLFS$ with the difference that it takes as input a cf-Petri net (resp a $glcf$-Petri net)

$\mathcal{N}$. We can observe that $ObjLFS$ is not needed to write the synthesis procedure, indeed $ObjLFS$ is more general than $glcfObjLFS$ and because the Petri nets $\mathcal{N}$ that the synthesis procedure takes as input are $glcf$-Petri net solving $glcfObjLFS$ is sufficient. The characterization given by theorem 4.2 shows that $glcfObjLFS$ and $cfObjLFS$ are closely related. Because $cfObjLFS$ is easier that $glcfObjLFS$ we first study $cfObjLFS$.

We used characterization of Theorem 4.1 to implement $cfObjLFS$.

Algorithm 2 Procedure $cfObjLFS(\mathcal{N}, M_0, M, \Sigma, \psi)$

1: Rewrite equation (a) and condition (b) of characterization given by Theorem 4.1 in logical formulas as well as ψ, and use an SMT solver to solve them.
2: Deduce σ from the SMT solver model

In order to rewrite equation (a) and condition (b) in Presburger arithmetic, we declared positive integer variables X_t and $Time_t$, for each transition t, X_t represents the value of $X(t)$ in the marking equation and $Time_t$ gives the position of t in the sequence σ such that $\sigma = \Phi(X)$ and $M_0 \xrightarrow{\sigma} M$. Then condition (b) can be expressed as

- $(X_t = 0) \Leftrightarrow (Time_t = 0)$, the transitions which are associated to a positive value $X(t)$ are the transitions which are fired and thus which are associated to a (strictly) positive time.
- $\bigvee_{t \in T \setminus T_0} (Time_t \neq 0 \implies \bigvee_{t' \to t} Time_t = Time_{t'} + 1)$, where $T_0 = T \cap {}^\bullet P_0$ and $t' \to t$ means that there exists $p \in t^\bullet \cap {}^\bullet t'$. This condition implies that there is no unmarked place, since given a transition t such that $X_t > 0$, it is possible to find transitions $t_0, ..., t_n = t$ such that $Time_{t_i} = Time_{t_{i-1}} + 1$, $t_{i-1} \to t_i$ and $t_0 \in T_0$.

The characterization 4.1 shows that if X is defined by $X(t) = Mod(X_t)$ where

- Mod is a model of the logical formulas encoding conditions (a) and (b),
- $Mod(X_t)$ designates the value of X_t under Mod,

then X satisfies the condition $M_0 \xrightarrow{X} M$. In practice we also prioritize smallest solutions by minimizing the sum on t of the values of X_t.

5.2.5 A procedure to find sequences in $glcf$-Petri net

Then we can use Characterization 4.2 to write $glcfObjLFS$ using $cfObjLFS$. (See algorithm 3).

At first sight, it could be strange to call the procedure $ObjLFS$, in order to implement $glcfObjLFS$, but since $ObjLFS$ is called on instance $(\tilde{\mathcal{N}}, \tilde{M}_0, \tilde{M}, \Sigma)$ of size significantly smaller than instance of $glcfObjLFS$, $(\mathcal{N}, M_0, M, \Sigma)$, thus it is possible to use the implementation seen in section 5.2.1 and to obtain a procedure more efficient than the procedure $ObjLFS$ seen in section 5.2.1.

Algorithm 3 Procedure $glcfObjLFS(\mathcal{N}, M_0, M, \Sigma)$

1: **for** M' such that $M' \geq M \wedge Supp(M') = Supp(M)$ **do**
2: $X' \leftarrow cfObjLFS(\mathcal{N}, M_0, M', \Sigma)$
3: **if** $X' \neq \perp$ **then**
4: $W \leftarrow Supp(M) \cup {}^\bullet X$
5: $\tilde{N} \leftarrow \mathcal{N}_W$
6: $\tilde{M_0} \leftarrow$ the restriction of M_0 to places of $\tilde{\mathcal{N}}$
7: $\tilde{M} \leftarrow$ the restriction of M' to places of $\tilde{\mathcal{N}}$
8: $X \leftarrow ObjLFS(\tilde{\mathcal{N}}, \tilde{M_0}, \tilde{M}, \Sigma)$
9: **if** $X \neq \perp$ **then**
10: **return** X
11: **end if**
12: **end if**
13: **end for**

Proposition 5.1 (Soundness and Completness of the Procedure)**.** The procedure defined in algorithm 3 is sound and complete.

- To prove soundness, we need to ensure that if the procedure returns a vector X, then X satisfies $M_0 \xrightarrow{X} M$ in $\mathcal{N}$ and $X \notin \Sigma$. This follows from the soundness of $cfObjLFS$, which holds by assumption.
- To prove completeness, we need to ensure that if there exists X such that $M_0 \xrightarrow{X} M$ then the procedure will return such a X. Indeed, suppose there exists X such that $M_0 \xrightarrow{X} M$, then theorem 4.2 ensures that there exists X' which differs from X only on κ-transition, and such that $M_0 \xrightarrow{X'} M'$ where $M' \geq M$ and $Supp(M) = Supp(M')$. Because X and X' differ only on κ-transitions, we have ${}^\bullet X' \subset Supp(M_0) \cup {}^\bullet X$, thus the search can be limited to $\mathcal{N}_W$ where $W = Supp(M_0) \cup {}^\bullet X$.

6 Future work

Despite hard work has already been done, there remains a lot of work before to harness code generation.

- In this study, to generate a procedure of a given type, we prioritize shortest solutions (we minimize the number of line that the procedure contains). We can also use a more sophisticated objective function to guide the search. This solution have been previously explored by Joel Galenson et al. in [6]. They developed a probabilistic model for the plugin *eclipse* of Java. This work can be extended for more libraries and the model can be enlarged and improved.
- Programs that are synthesized are loop-free and without conditionals. A main challenge in is to add loops and conditionals.

- Users, willing to use automatic generation tools have to give some specification of the code they want to generate. A clever implementation for code generation, needs to study carefully the better way to specify extern information. The easiest the specifications can be expressed and more the application will be useful.

7 Conclusion

Automatic code generation is a thrilling idea. It seems to require a great computational power but with the developement of technologies and solid theoretical tools such as Petri net, useful tools for programmers that generate code have already been developed. Nevertheless, for the moment, these tools are limited. They failed to synthesise complex functions with many lines of code and they can't manage every subtilities that language such as Java proposes, like loops or conditionals. A first step for harness code generation, is to improve efficiency of existing application for code generation. That why, in this paper we have presented an algorithm used by a state-of-the-art application (see [5]), which is able to generate code automatically. The algorithm is presented in a general form, in order to emphase possible customization. We have also studied theoretically tools that are used for code generation and finally we have modified algorithm of [5]. The modifications are based on our theoretical results. They aim at improving the speed of code generation.

References

[1] Wojciech Czerwinski and Slawomir Lasota. Partially-commutative context-free languages. In Bas Luttik and Michel A. Reniers, editors, *Proceedings Combined 19th International Workshop on Expressiveness in Concurrency and 9th Workshop on Structured Operational Semantics, EXPRESS/SOS 2012, Newcastle upon Tyne, UK, September 3, 2012*, volume 89 of *EPTCS*, pages 35–48, 2012.

[2] Wojciech Czerwinski, Slawomir Lasota, Ranko Lazic, Jérôme Leroux, and Filip Mazowiecki. The reachability problem for petri nets is not elementary. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 24–33. ACM, 2019.

[3] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[4] Javier Esparza. Petri nets, commutative context-free grammars, and basic parallel processes. In Horst Reichel, editor, *Fundamentals of Computation Theory, 10th International Symposium, FCT '95, Dresden, Germany, August 22-25, 1995, Proceedings*, volume 965 of *Lecture Notes in Computer Science*, pages 221–232. Springer, 1995.

[5] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. Component-based synthesis for complex APIs. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 599–612. ACM, 2017.

[6] Joel Galenson, Philip Reames, Rastislav Bodík, Björn Hartmann, and Koushik Sen. Codehint: dynamic and interactive synthesis of code snippets. In Pankaj Jalote, Lionel C. Briand, and André van der Hoek, editors, *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 653–663. ACM, 2014.

[7] Gilles Geeraerts, Alexander Heußner, M. Praveen, and Jean-François Raskin. ω-petri nets: Algorithms and complexity. *Fundam. Inform.*, 137(1):29–60, 2015.

[8] Dung T. Huynh. Commutative grammars: The complexity of uniform word problems. *Inf. Control.*, 57(1):21–39, 1983.

[9] Jérôme Leroux and Sylvain Schmitz. Reachability in vector addition systems is primitive-recursive in fixed dimension. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*, pages 1–13. IEEE, 2019.

[10] Zhanpeng Liang and Yu Feng and. Component-based program synthesis in ocaml. 2017.

[11] David Justo Jiaxiao Zhou Ziteng Wang Ranjit Jhala Zheng Guo, Michael James and Nadia Polikarpova. Program synthesis by type-guided abstraction refinement. *ACM SIGPLAN*, 2019.