Towards a certified Petri net model-checker

Lukasz Fronc, Franck Pommereau

Towards a certified Petri net model-checker

Lukasz Fronc and Franck Pommereau

IBISC, University of Évry, Tour Évry 2
523 place des terrasses de l'Agora, 91000 Évry, France
{fronc,pommereau}@ibisc.univ-evry.fr

Abstract. *Petri nets* are widely used in the domain of automated verification through *model-checking*. In this approach, a Petri Net model of the system of interest is produced and its reachable states are computed, searching for erroneous executions. *Model compilation* can accelerate this analysis by generating code to explore the reachable states. This avoids the use of a fixed exploration tool involving an "interpretation" of the Petri net structure. In this paper, we show how to compile Petri nets targeting the *LLVM language* (a high-level assembly language) and formally prove the *correctness* of the produced code. To this aim, we define a *structural operational semantics* for the fragment of LLVM we use.

Keywords: explicit model-checking, model compilation, LLVM, SOS

1 Introduction

Verification through *model-checking* [1] consists in defining a formal model of the system to be analysed and then using automated tools to check whether the expected properties are met or not. We consider here more particularly the widely adopted setting in which models are expressed using *coloured Petri nets* [9] and there states are explored using *explicit model-checking* that enumerates them all (contrasting with symbolic model-checking that works with sets of states). *Model compilation* is one of the numerous techniques to speedup explicit model-checking, it relies on generating source code (then compiled into machine code) to produce a high-performance implementation of the state space exploration primitives. For instance, this approach is successfully used in the well known SPIN tool [7], or in Helena coloured Petri net model-checker [14, 3].

In this paper, we propose a way to prove the correctness of such an approach. More precisely, we focus on the produced code and prove that the object computed by its execution is an actual representation of the state space of the compiled model. We consider the *Low-Level Virtual Machine* (*LLVM*) language as our target language for compilation, which reconciles two otherwise contradictory objectives: on the one hand, this is a typed language with reasonably high-level operations allowing to express algorithms quite naturally; on the other hand, it is a low-level language that can be equipped with a formal semantics allowing to formally prove the programs correctness. To do so, we define a structural operational semantics of the fragment of LLVM we need and use it to establish the properties of the programs generated by our compiler.

To the best of our knowledge, this is the first attempt to provide a formal semantics for LLVM. Moreover, if model-checkers are widely used tools, there exists surprisingly few attempts to prove them at the implementation level [19], contrasting with the domain of proof assistants [2, 15] for which "proving the prover" is a common expectation.

The rest of the paper is organised as follows. We first recall the main notions about coloured Petri nets. Then, we present the LLVM framework, in particular the syntax of the language and its intuitive semantics, and how it can be embedded LLVM into a Petri net as a concrete colour domain. In section 4, we present algorithms and data structures for state space exploration. We then formally define an operational semantics for LLVM, including an explicit memory model. Finally we present our correctness results. Due to the limited number of pages, many definitions and intermediary results have been omitted, as well as the detailed proofs. This material can be found in [5, 4]. Notice also that our compilation approach is evaluated from a performance point of view in [6].

2 Coloured Petri nets

A (coloured) Petri net involves objects defined by a *colour domain* that provides data values, variables, operators, a syntax for expressions, possibly typing rules, etc. Usually, elaborated colour domains are used to ease modelling; in particular, one may consider a functional programming language [9, 17] or the functional fragment (expressions) of an imperative programming language [14, 16]. In this paper we will consider LLVM as a concrete colour domain.

All these can be seen as implementations of a more general *abstract colour domain* providing $\mathbb{D}$ the set of *data values*, $\mathbb{V}$ the set of *variables* and $\mathbb{E}$ the set of *expressions*. 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 may be considered as (simple) expressions, *i.e.*, we assume $\mathbb{D} \cup \mathbb{V} \subseteq \mathbb{E}$.

At this abstract level, we do not make any assumption about the typing or syntactical correctness of expressions; instead, we assume that any expression can be evaluated, possibly to $\bot \notin \mathbb{D}$ (undefined value) in case of any error. More precisely, a *binding* is a partial function $\beta : \mathbb{V} \to \mathbb{D} \cup \{\bot\}$. Then, let $e \in \mathbb{E}$ and β be a binding, we denote by $\beta(e)$ the evaluation of e under β; if the domain of β does not include $\mathit{vars}(e)$ then $\beta(e) \stackrel{\text{df}}{=} \bot$. The application of a binding to evaluate an expression is naturally extended to sets and multisets of expressions.

Definition 1 (Petri nets). *A* Petri net *is a tuple* (S, T, ℓ) *where* S *is the finite set of* places*,* T*, disjoint from* S*, is the finite set of* transitions*, and* 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 values that* s *may contain;*
- *for all* $t \in T$*,* $\ell(t) \in \mathbb{E}$ *is the* guard *of* t*,* i.e.*, a condition for its execution;*
- *for all* $(x, y) \in (S \times T) \cup (T \times S)$*,* $\ell(x, y)$ *is a multiset over* $\mathbb{E}$ *and defines the* arc *from* x *toward* y*.*

A marking *of a Petri net is a map that associates to each place $s \in S$ a multiset of values from $\ell(s)$. From a marking M, a transition t can be* fired *using a binding β and yielding a new marking M', which is denoted by $M[t, \beta\rangle M'$, iff:*

- *there are enough tokens: for all $s \in S$, $M(s) \geq \beta(\ell(s, t))$;*
- *the guard is validated: $\beta(\ell(t)) = true$;*
- *place types are respected: for all $s \in S$, $\beta(\ell(t, s))$ is a multiset over $\ell(s)$;*
- *M' is M with tokens consumed and produced according to the arcs: for all $s \in S$, $M'(s) = M(s) - \beta(\ell(s, t)) + \beta(\ell(t, s))$.*

Such a binding β is called a mode *of t at marking M.*

For a Petri net node $x \in S \cup T$, we define ${}^\bullet x \overset{\text{df}}{=} \{y \in S \cup T \mid \ell(y, x) \neq \emptyset\}$ and $x^\bullet \overset{\text{df}}{=} \{y \in S \cup T \mid \ell(x, y) \neq \emptyset\}$ where $\emptyset$ is the empty multiset. Finally, we extend the notation vars to a transition by taking the union of the variable sets in its guard and connected arcs.

In this paper, we assume that the considered Petri nets are such that, for all place $s \in S$ and all transition $t \in T$, $\ell(s, t)$ is either $\emptyset$ or contains a single variable $x \in \mathbb{V}$. We also assume that $vars(t) = \bigcup_{s \in S} vars(\ell(s, t))$, *i.e.*, all the variables involved in a transition can be bound using the input arcs. The second assumption is a classical one that allows to simplify the discovery of modes. The first assumption is made without loss of generality to simplify the presentation.

3 LLVM

The *LLVM* project (*Low Level Virtual Machine*) [11] is a modern and modular toolkit for compiler development used by a wide variety of commercial and open source projects as well as academic researches [13, 12]. The *LLVM-IR* (*LLVM Intermediate Representation*) [10] is a part of the LLVM project and is a low-level, platform-independent, intermediate language. Every program written in this language can be run in a virtual machine or compiled to native code on all the platforms supported by the LLVM project. Importantly, the LLVM compiler runs a variety of optimisation passes on the LLVM-IR, which allows us to produce simple source code knowing it will be optimised by LLVM.

3.1 Syntax and intuitive semantics

A LLVM program is composed of a set of *blocks* (*i.e.*, sequences of instructions) identified by *labels*. Entering or leaving a block is always explicit through branching instructions (jumps), subprograms calls or return instructions.

To define the syntax, we consider the following pairwise disjoint sets:

- $\mathbb{P}$ is the set of *pointers*;
- $\mathbb{T}$ is the set of *types*, defined inductively as the smallest set containing the *primitive types* in $\mathbb{T}_0 \overset{\text{df}}{=} \{int, bool, \dots\}$ (integers, Boolean values and other types defined by LLVM but not needed here) and such that if $t_0, \dots, t_n \in \mathbb{T}$ then $struct(t_0, \dots, t_n) \in \mathbb{T}$, which represents a data structure with $n + 1$ fields of types t_0 to t_n;

- $\mathbb{L}$ is the set of *labels*, it contains arbitrary names as well as some specific labels like $f\llparenthesis a_1, \dots, a_n \rrparenthesis$, where $a_i \in \mathbb{V}$ for $1 \leq i \leq n$, that correspond to subprograms entry points (including the formal parameters). We define a set $\mathbb{L}_\perp \overset{\text{df}}{=} \mathbb{L} \cup \{\perp\}$ where $\perp \notin \mathbb{L}$ stands for an undefined label.

A program is represented as a partial function P from $\mathbb{L}$ to the set of blocks, *i.e.*, that associates each label in its domain to a sequence of instructions.

For our purpose, we need to consider a fragment of LLVM that is formed by three main syntactic classes: sequences in *seq*, commands in *cmd* (*i.e.*, instructions) and expressions in *expr*. A sequence is a list of commands which may end with an expression, in which case it is considered as an expression itself (which is not reflected on the grammar in figure 1 to keep it simpler).

We assume that programs are syntaxically correct and well typed, so that we can simplify the syntax by forgetting all types in LLVM source code. The resulting syntax is presented in figure 1. To allow for writing one-line sequences, we introduce the sequencing operator ";" that corresponds to the line endings. We also introduce the *skip* command that denotes the empty sequence. It may be noted that *pcall* (procedure call) and *fcall* (function call) do not exist in LLVM but are different instances of the *call* instruction. This distinction can be easily made in LLVM because the instruction contains the return type of the subprogram (function or procedure). Instruction *store* (resp. *load*) is the action of storing (resp. loading) data into (resp. from) the memory through a pointer. Instruction *icmp* compares two integers. Instruction *phi* is used to access variables assigned in previously executed blocks. Instruction *gep* corresponds to pointer arithmetic, we freeze the second argument to 0, which is enough to access fields in structures by their indexes.

3.2 LLVM-labelled Petri Nets

To compile Petri nets as defined previously into LLVM programs, we need to consider a variant where the colour domain explicitly refers to a LLVM program.

Definition 2 (LLVM labelled Petri nets). *A LLVM labelled Petri net is a tuple $N \overset{\text{df}}{=} (S, T, \ell, P)$, where P is a LLVM program, and such that (S, T, ℓ) is a coloured Petri net with the following changes:*

- *for all place $s \in S$, $\ell(s)$ is a LLVM type in $\mathbb{T}$, interpreted as a subset of $\mathbb{D}$;*
- *for all transition $t \in T$, $\ell(t)$ is a call to a Boolean function in P whose parameters are the elements of $vars(t)$;*
- *for all $s \in t^\bullet$, $\ell(t, s)$ is a singleton multiset whose unique element is a call to a $\ell(s)$-typed function in P whose parameters are the elements of $vars(t)$.*

We assume that all the functions involved in the annotations terminate.

With respect to the previous definition, we have concretized the place types and each expression is now implemented as a LLVM function called from the corresponding annotation. To simplify the presentation, we have also restricted the output arcs to be singleton multisets, but this can be easily generalised.

$$\begin{array}{rcll}
\mathit{seq} & ::= & \mathit{cmd} & \text{(statement)} \\
 & | & \mathit{expr} & \text{(expression)} \\
 & | & \mathit{cmd};\ \mathit{seq} & \text{(sequence of instructions)} \\
\mathit{cmd} & ::= & \mathit{br}\ \mathit{label} & \text{(unconditional branching)} \\
 & | & \mathit{br}\ \mathit{rvalue},\ \mathit{label},\ \mathit{label} & \text{(conditional branching)} \\
 & | & \mathit{pcall}\ \mathit{label}(\mathit{rvalue}, \dots, \mathit{rvalue}) & \text{(procedure call)} \\
 & | & \mathit{ret} & \text{(return from a procedure)} \\
 & | & \mathit{var} = \mathit{expr} & \text{(variable assignment)} \\
 & | & \mathit{store}\ \mathit{rvalue},\ \mathit{rvalue} & \text{(assignment through a pointer)} \\
 & | & \mathit{skip} & \text{(empty sequence)} \\
\mathit{expr} & ::= & \mathit{add}\ \mathit{rvalue},\ \mathit{rvalue} & \text{(addition)} \\
 & | & \mathit{load}\ \mathit{rvalue} & \text{(read a value through a pointer)} \\
 & | & \mathit{gep}\ \mathit{rvalue},\ 0,\ \mathit{nat} & \text{(get a pointer to a structure field)} \\
 & | & \mathit{icmp}\ \mathit{op},\ \mathit{rvalue}, \mathit{rvalue}, & \text{(integers comparison)} \\
 & | & \mathit{phi}\ (\mathit{rvalue}, \mathit{label}), \dots, (\mathit{rvalue}, \mathit{label}) & \text{(get a value after branching)} \\
 & | & \mathit{fcall}\ \mathit{label}(\mathit{rvalue}, \dots, \mathit{rvalue}) & \text{(function call)} \\
 & | & \mathit{alloc}\ \mathit{type} & \text{(memory allocation)} \\
 & | & \mathit{ret}\ \mathit{rvalue} & \text{(return a value from a function)} \\
 & | & \mathit{rvalue} & \text{(variable or value)}
\end{array}$$

Fig. 1. Our fragment of the LLVM syntax, where $\mathit{label} \in \mathbb{L}$, $\mathit{rvalue} \in \mathbb{D} \cup \mathbb{P} \cup \mathbb{V}$, $\mathit{var} \in \mathbb{V}$, $\mathit{type} \in \mathbb{T}$, $\mathit{nat} \in \mathbb{N}$ and $\mathit{op} \in \{<, \leq, =, \neq, \geq, >\}$.

Moreover, the definitions of binding and modes are extended to LLVM. A *LLVM binding* is a partial function $\beta : \mathbb{V} \to \mathbb{D} \cup \mathbb{P}$ that maps each variable from its domain to a pointer or a value, and that is widened to $\mathbb{D}$ by the identity function. $\mathbb{B}$ is the set of all LLVM bindings. A *LLVM mode* is thus a LLVM binding enabling a transition in a LLVM labelled Petri net.

4 Implementing state space exploration

Given an *initial marking* M_0, the state space we want to compute in this paper is the set R of *reachable marking*, *i.e.*, the smallest set such that $M_0 \in R$ and, if $M \in R$ and $M[t, \beta\rangle M'$ then $M' \in R$ also. The correctness and termination of the implementation presented in this section are addressed in section 6.

Our algorithms are implemented on the basis of data structures (multisets, places, markings, and sets) that must respect some interfaces. An interface is presented as a set of procedures or functions that manipulates a data structure through a pointer (C-like interfaces). Moreover, each such subprogram has a formal specification of its behaviour that relies on an explicit interpretation of the data structure before and after the subprogram call. Precise examples are given in section 5.3, after the definition of LLVM formal semantics.

A multiset structure to store values from a type d is assumed and we call the set of values from d having non-zero occurrences in the multiset its *domain*. The multiset interface contains in particular: two procedures $\mathit{add}_{mset}(\!|\, p_{mset}, \mathit{elt} \,|\!)$ and $\mathit{rem}_{mset}(\!|\, p_{mset}, \mathit{elt} \,|\!)$ to respectively add or remove an element elt in p_{mset}; a function $size_{mset}\llparenthesis p_{mset} \rrparenthesis$ to return the domain size; a function $nth_{mset}\llparenthesis p_{mset}, n \rrparenthesis$ to return the n^{th} element from the domain (for an arbitrary fixed order).

As a container of tokens, a place can be basically implemented as a multiset of tokens. So the place interface is exactly the multiset interface but annotated by the place name, for instance add_s is like add_{mset} but for place s.

The markings interface contains for each place s a function $get_s\llparenthesis p_{mrk} \rrparenthesis$ that returns a pointer to the corresponding place structure, as well as a function $copy_{mrk}\llparenthesis p_{mrk} \rrparenthesis$ that returns a copy of the marking structure.

Finally, the set interface contains a function $cons_{set}\llparenthesis\, \rrparenthesis$ that builds a new empty set and a procedure $add_{set}\llparenthesis p_{set}, elt \rrparenthesis$ that adds an element elt to p_{set}.

Transitions firing. Let $t \in T$ be a transition such that ${}^\bullet t = \{s_1, \dots, s_n\}$ and $t^\bullet = \{s'_1, \dots, s'_m\}$. Then, function $fire_t$, that computes the marking M' reachable from M by firing t given a valuation of its variables, can be expressed as shown on the left of figure 2. This function simply creates a copy M' of M, removes from it the consumed tokens and adds the produced tokens before to return M'. One could remark that it avoids a loop over the Petri net places but instead it executes a sequence of statements. This is generally more efficient (no branching penalties, no loop overhead, no lookup of functions $f_{t,s'_j}, \dots$) and the code is simpler to generate. Let now x_{mrq} be a pointer to a marking structure implementing M. The firing algorithm can be implemented as shown on the right of figure 2.

Successors computation. To discover all the possible modes for transition t, function $succ_t$ enumerates all the combinations of tokens from the input places. If a combination corresponds to a mode then the suitable transition firing function is called to produce a new marking. This algorithm is shown in figure 3. Note the nesting of loops that avoids an iteration over ${}^\bullet t$, which saves from querying the Petri net structure and avoids the explicit construction of a binding. Moreover,

```
$fire_t : M,\ x_1,\ \dots,\ x_n \to M'$
    $M' \leftarrow copy(M)$
    $M'(s_1) \leftarrow M(s_1) - \{x_1\}$
    ...
    $M'(s_n) \leftarrow M(s_n) - \{x_n\}$

    $M'(s'_1) \leftarrow M(s'_1) + f_{t,s'_1}(x_1, \dots, x_n)$
    ...
    $M'(s'_m) \leftarrow M(s'_m) + f_{t,s'_m}(x_1, \dots, x_n)$

    **return** $M'$
```

$$P(fire_t\llparenthesis x_{mrq}, x_1, \dots, x_n \rrparenthesis) \stackrel{\text{df}}{=} \left\{ \begin{array}{l} \quad \textsf{// copy the structure marking} \\ x'_{mrq} = fcall\ copy_{mrq}(x_{mrq}) \\ \quad \textsf{// consume tokens} \\ \left.\begin{array}{l} x_{s_i} = fcall\ get_{s_i}(x'_{mrq}) \\ pcall\ rem_{s_i}(x_{s_i}, x_i) \end{array}\right\}_{1 \le i \le n} \\ \quad \textsf{// produce tokens} \\ \left.\begin{array}{l} x_{s'_j} = fcall\ get_{s'_j}(x'_{mrq}) \\ o_{s'_j} = fcall\ f_{t,s'_j}(x_1, \dots, x_n) \\ pcall\ add_{s'_j}(x_{s'_j}, o_{s'_j}) \end{array}\right\}_{1 \le j \le m} \\ \quad \textsf{// return the new marking} \\ ret\ x'_{mrq} \end{array}\right.$$

Fig. 2. On the left, the transition firing algorithm, where x_i is the variable in $\ell(s_i, t)$ for all $1 \le i \le n$, and f_{t,s'_j} is the function called in $\ell(t, s'_j)$ for all $1 \le j \le m$. On the right, its LLVM implementation.

since g_t is written in the target language, we avoid an interpretation of the corresponding expression. For the LLVM version, let x_{mrq} be a pointer to a marking structure and x_{next} be a pointer to a marking set structure. Then, the algorithm from figure 3 can be implemented as shown in figure 4. Each iteration over x_k is implemented as a set of blocks subscribed by t, k (for $n \geq k \geq 1$); blocks subscribed by $t, 0$ corresponds to the code inside the innermost loop. Note the *phi* instruction to update the value of index i_{s_i} (used to enumerate the tokens in place s_i): when the program enters block $loop_{t,i}$ for the first time, it comes from block $header_{t,i}$, so we initialise the value of i_{s_i} to the last index in the domain of s_i; later, the program comes back to block $loop_{t,i}$ from block $footer_{t,i}$, so it assigns i'_{s_i} to i_{s_i} that is the value of $i_{s_i} - 1$ (*i.e.*, the previous index).

A function *succ* is also defined to compute the set of all the successors of a marking, which is made by calling all the transition specific successor functions and accumulating the discovered markings into the same set. This algorithm and its translation in LLVM are shown in figure 5.

5 A formal semantics of LLVM

5.1 Memory model

To start with, we define a memory model for LLVM, including *heaps* to store dynamically allocated pointers as well as *stacks* to store local variables and arguments for subprograms calls.

A *heap* is a partial function $H : \mathbb{P} \to \mathbb{T} \times (\mathbb{D} \cup \mathbb{P} \cup \{\bot\})^*$ with a finite domain. Each heap maps every pointer in its domain to a type and a tuple of values or pointers. The set of all heaps is $\mathbb{H}$. A heap is well formed if every pair in its image is type-consistent, for instance if $H(p) = (int, d)$ then d is indeed an integer or is $\bot$ (uninitialised).

The set of all the pointers accessible starting from a pointer p in a heap H is denoted by $p{\downarrow_H}$ and is defined for all p in $\mathbb{P}$ as:

$$\begin{array}{ll} p{\downarrow_H} \stackrel{\text{df}}{=} \{\} & \text{if } p \notin dom(H) \\ p{\downarrow_H} \stackrel{\text{df}}{=} \{p\} & \text{if } H(p) = (t, v) \text{ and } t \in \mathbb{T}_0 \\ p{\downarrow_H} \stackrel{\text{df}}{=} \{p\} \cup p_0{\downarrow_H} \cup \cdots \cup p_n{\downarrow_H} & \text{if } H(p) = (struct(t_0, \ldots, t_n), (p_0, \ldots, p_n)) \end{array}$$

In can be shown that if a heap H is well formed then $p{\downarrow_H} \subseteq dom(H)$ for every $p \in dom(H)$, and more generally that $dom(H) = \bigcup_{p \in \mathbb{P}} p{\downarrow_H}$.

Then, we need to access and update the data stored onto a heap. For each heap H, we define a data structure traversal function $\cdot[\cdot]_H : \mathbb{P} \times \mathbb{N} \to \mathbb{P} \cup \mathbb{D}$ as:

$$p[i]_H \stackrel{\text{df}}{=} \begin{cases} p_i & \text{if } H(p) = (struct(t_0, \ldots, t_n), (p_0, \ldots, p_n)) \text{ and } 0 \leq i \leq n \\ \text{undefined otherwise} \end{cases}$$

The overwriting function $\oplus : \mathbb{H} \times \mathbb{H} \to \mathbb{H}$ represents the writing into memory and is defined for each $p \in \mathbb{P}$ as:

$$(H \oplus H')(p) \stackrel{\text{df}}{=} \begin{cases} H'(p) & \text{if } p \in dom(H') \\ H(p) & \text{if } p \notin dom(H') \wedge p \in dom(H) \\ \text{undefined otherwise} \end{cases}$$

```
$succ_t : M,\ next \to \bot$

for $x_n$ in $M(s_n)$ do
  ...
  for $x_1$ in $M(s_1)$ do
    if $g_t(x_1, \ldots, x_n)$ then
      $next \leftarrow next \cup \{fire_t(M, x_1, \ldots, x_n)\}$
    endif
  endfor
  ...
endfor
```

Fig. 3. Transition specific successors computation algorithm, where g_t is the function that evaluates the guard $\ell(t)$.

$$P(succ_t \llparenthesis x_{mrq}, x_{next} \rrparenthesis) \stackrel{\text{df}}{=} \begin{cases} br\ header_{t,n} \end{cases}$$

$$P(header_{t,k}) \stackrel{\text{df}}{=} \begin{cases} x_{s_k} = fcall\ get_{s_k}(x_{mrq}) \\ s_{s_k} = fcall\ size_{s_k}(x_{s_k}) \\ br\ loop_{t,k} \end{cases}$$

$$P(loop_{t,k}) \stackrel{\text{df}}{=} \begin{cases} i_{s_k} = phi\ (s_{s_k}, header_{t,k}), (i'_{s_k}, footer_{t,k}) \\ c_{s_k} = icmp\ >,\ i_{s_k},\ 0 \\ br\ c_{s_k},\ body_{t,k},\ footer_{t,k+1} \end{cases}$$

$$P(body_{t,k}) \stackrel{\text{df}}{=} \begin{cases} x_k = fcall\ nth_{s_k}(x_{s_k}, i_{s_k}) \\ br\ header_{t,k-1} \end{cases}$$

$$P(footer_{t,k}) \stackrel{\text{df}}{=} \begin{cases} i'_{s_k} = add\ i_{s_k},\ -1 \\ br\ loop_{t,k} \end{cases}$$

$$P(header_{t,0}) \stackrel{\text{df}}{=} \begin{cases} c_g = fcall\ g_t(x_1, \ldots, x_n) \\ br\ c_g,\ body_{t,0},\ footer_{t,1} \end{cases}$$

$$P(body_{t,0}) \stackrel{\text{df}}{=} \begin{cases} x'_{mrq} = fcall\ fire_t(x_{mrq}, x_1, \ldots, x_n) \\ pcall\ add_{set}(x_{next}, x'_{mrq}) \\ br\ footer_{t,1} \end{cases}$$

$$P(footer_{t,n+1}) \stackrel{\text{df}}{=} \begin{cases} ret \end{cases}$$

Fig. 4. LLVM transition specific successor function, for $1 \leq k \leq n$.

```
$succ : M \to next$

  $next \leftarrow \emptyset$
  $succ_{t_1}(M, next)$
  $succ_{t_2}(M, next)$
  ...
  $succ_{t_n}(M, next)$
  return $next$
```

$$P(succ \llparenthesis x_{mrq} \rrparenthesis) \stackrel{\text{df}}{=} \begin{cases} x_{next} = fcall\ cons_{set}() \\ pcall\ succ_{t_1}(x_{mrq}, x_{next}) \\ pcall\ succ_{t_2}(x_{mrq}, x_{next}) \\ \ldots \\ pcall\ succ_{t_n}(x_{mrq}, x_{next}) \\ ret\ x_{next} \end{cases}$$

Fig. 5. Computation of all successors (left) and its LLVM implementation (right), where x_{mrq} is a pointer to a structure marking.

In order to compare heaps, a notion of structural equivalence needs to be defined. This relation ensures that two heaps contain the same data, accessible from distinct sets of pointers but with the same layout. More precisely we consider two heaps H, H' and two pointers p, p' and write $(H,p) =_{st} (H',p')$ whenever $H(p)$ and $H'(p')$ are structurally the same values.

We also need to define an operation $\mathit{new} : \mathbb{H} \times \mathbb{T} \to \mathbb{H} \times \mathbb{P}$ to build new heaps, which corresponds to a pointer allocation, using a helper function $\mathit{alloc} : 2^{\mathbb{P}} \times \mathbb{P} \times \mathbb{T} \to \mathbb{H}$ as follows:

$$\begin{aligned}
\mathit{new}(H,t) &\overset{\text{df}}{=} (\mathit{alloc}(\mathit{dom}(H) \cup \{p\}, p, t), p) \\
&\qquad\qquad \text{for } p \notin \mathit{dom}(H) \text{ a "fresh" pointer} \\
\mathit{alloc}(d,p,t) &\overset{\text{df}}{=} \{p \mapsto (t, \bot)\} \qquad\qquad \text{for } t \in \mathbb{T}_0 \\
\mathit{alloc}(d,p,\mathit{struct}(t_0,\ldots,t_n)) &\overset{\text{df}}{=} \{p \mapsto (\mathit{struct}(t_0,\ldots,t_n),(p_0,\ldots,p_n))\} \\
&\quad \oplus \mathit{alloc}(d \cup \{p_0,\ldots,p_n\}, p_0, t_0) \\
&\quad \oplus \cdots \\
&\quad \oplus \mathit{alloc}(d \cup \{p_0,\ldots,p_n\}, p_n, t_n) \\
&\qquad\qquad \text{for } p_0,\ldots,p_n \notin d \text{ "fresh" pointers}
\end{aligned}$$

It can be shown that *new* always returns a well formed heap, and that calling *new* using equivalent heaps always returns equivalent heaps.

To define subprogram calls, our memory model also defines *stacks* that contain *frames* implicitly pushed onto the stack by the inference rules in the semantics. A *frame* is a tuple $F \in \mathbb{F} \overset{\text{df}}{=} \mathbb{L}_\bot \times \mathbb{L} \times \mathbb{B}$ whose elements are denoted by $(l_{p,F},\ l_{c,F},\ \beta_F)$, where $l_{p,F}$ is the label the block the program comes from (or undefined), $l_{c,F}$ is the label of the block currently executed, and β_F is a LLVM binding representing the current evaluation context. We widen the binding functional notation to the frames, so we denote by $F(x)$ the binding $\beta_F(x)$ of x by β_F.

Like for heaps we need operations to update frames. The same operator $\oplus$ is used because the operations are very similar, but on distinct objects. The binding overwriting operation $\oplus : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ and the frame binding overwriting operation $\oplus : \mathbb{F} \times \mathbb{B} \to \mathbb{F}$ are defined as:

$$(\beta \oplus \beta')(p) \overset{\text{df}}{=} \begin{cases} \beta'(p) & \text{if} \quad p \in \mathit{dom}(\beta') \\ \beta(p) & \text{if} \quad p \notin \mathit{dom}(\beta') \land p \in \mathit{dom}(\beta) \\ \text{undefined otherwise} \end{cases}$$

$$(l,\ l',\ \beta) \oplus \beta' \overset{\text{df}}{=} (l,\ l',\ \beta \oplus \beta')$$

The structural equivalence can be widened to pairs of heaps and frames and is denoted by $(H,F) =_{st} (H',F')$ for two heaps H, H' and two frames F, F'. Intuitively, it checks that all data accessible from the frame bindings are structurally equivalent. This holds also for values stored directly in the bindings (*i.e.*, without pointers) since the heap equivalence reduces to the equality on $\mathbb{D}$.

5.2 Inference rules

The operational semantics is defined for a fixed and immutable program P, which means that no function nor block can be created nor modified during the execution. We denote the result of a computation by $\overline{\;\cdot\;}$, for example $\overline{2+3}$ is 5. The main objects handled by our inference rules are *configurations* that represent a state of the program during its execution. A configuration is a tuple (seq, H, F), denoted by $\big(\, seq \,\big)_{H,F}$, where *seq* is a sequence of instructions, H is a heap and F is a frame.

The inference rules for expressions are shown in figure 6; expressions evaluate to values in the context of a frame. The inference rules for sequence and commands are shown in figure 7; sequences and commands evaluate to other sequences or commands in the context of a heap and a frame. One can remark how a frame is pushed onto the stack in *pcall* and *fcall* rules, a new frame F_0 is actually replacing the current frame F in the subsumption of these rules and used to execute the body of the called subprogram. This semantics mixes up small-step and big-step reductions. Indeed, most of the rules are small-step except for *pcall* and *fcall* rules in which we link the computation to its result by making a sequence of reductions in the rule subsumption.

5.3 Data structures interpretation

The link between Petri nets and their LLVM implementation is formalised with a family of interpretation functions for all data structures. This allows to formalise the behavioural requirements on the interfaces presented in section 4.

An interpretation is a partial function which maps a pair formed by a heap and a pointer to a Petri net object: a marking, a set of markings, a multiset of tokens or a single token, depending on the interpreted object. Interpretations are denoted by $[\![H, p]\!]^\star$, where $H \in \mathbb{H}$, $p \in \mathbb{P} \cup \mathbb{D}$ and $\star$ is an annotation describing the interpreted object (for instance we use $mset(t)$ instead of $\star$ to interpret a multiset over a type t). Whenever p is a pointer, we assume that the interpretation depends only on data that is accessible from p, *i.e.*, $p{\downarrow_H}$. Moreover every interpretation function has to respect the following consistency requirement.

Requirement 1 (Consistency) *Let H, H' be two heaps, $[\![\cdot,\cdot]\!]^\star$ an interpretation function, and p, p' two pointers or values. If $(H,p) =_{st} (H',p')$ then $[\![H,p]\!]^\star = [\![H',p']\!]^\star$.*

As presented in section 4, we use data structures and functions as basic blocks for constructing our algorithms, they are either predefined or produced by the compilation process. Each of these functions and data structures is specified (actually, axiomatized) by a formal interface. In particular, this helps to ensure independence and modularity between components both in a programmatic and formal way. Specifying an interface leads to define a set of primitives that respect given derivations and interpretations. For example, let H be a heap and F a

$$\frac{F(x) = p \quad H(p) = (t,v)}{\big(\, \mathit{load}\ x \,\big)_{H,F} \rightsquigarrow \big(\, v \,\big)_{H}}\ \mathit{load} \qquad \frac{F(x) = p \quad p[i]_H \text{ is defined}}{\big(\, \mathit{gep}\ x,\ 0,\ i \,\big)_{H,F} \rightsquigarrow \big(\, p[i]_H \,\big)_{H}}\ \mathit{gep}_0$$

$$\frac{\overline{F(x_1) + F(x_2)} = v}{\big(\, \mathit{add}\ x_1,\ x_2 \,\big)_{H,F} \rightsquigarrow \big(\, v \,\big)_{H}}\ \mathit{add} \qquad \frac{(H', p) = \mathit{new}(H, t)}{\big(\, \mathit{alloc}\ t \,\big)_{H,F} \rightsquigarrow \big(\, p \,\big)_{H \oplus H'}}\ \mathit{alloc}$$

$$\frac{\overline{F(x_1)\ \mathit{op}\ F(x_2)} = v \quad \mathit{op} \in \{<, \leq, =, \neq, \geq, >\}}{\big(\, \mathit{icmp}\ \mathit{op},\ x_1,\ x_2 \,\big)_{H,F} \rightsquigarrow \big(\, v \,\big)_{H}}\ \mathit{icmp}$$

$$\frac{1 \leq i \leq n \quad l_i \neq \bot}{\big(\, \mathit{phi}\ (x_1, l_1), \ldots, (x_n, l_n) \,\big)_{H,(l_i,\ l_c,\ \beta)} \rightsquigarrow \big(\, \beta(x_i) \,\big)_{H}}\ \mathit{phi}$$

$$\frac{\begin{array}{c} f \llparenthesis a_1, \ldots, a_n \rrparenthesis \in \mathit{dom}(P) \\ F_0 = (\bot,\ f \llparenthesis a_1, \ldots, a_n \rrparenthesis,\ \{a_1 \mapsto F(r_1), \ldots, a_n \mapsto F(r_n)\}) \\ \big(\, P(f \llparenthesis a_1, \ldots, a_n \rrparenthesis) \,\big)_{H,F_0} \rightsquigarrow^{*} \big(\, v \,\big)_{H'} \end{array}}{\big(\, \mathit{fcall}\ f(r_1, \ldots, r_n) \,\big)_{H,F} \rightsquigarrow \big(\, v \,\big)_{H'}}\ \mathit{fcall}$$

$$\frac{}{\big(\, \mathit{ret}\ r \,\big)_{H,F} \rightsquigarrow \big(\, F(r) \,\big)_{H}}\ \mathit{ret}$$

Fig. 6. Rules for expressions.

$$\frac{\big(\, \mathit{cmd} \,\big)_{H,F} \rightsquigarrow \big(\, \mathit{seq}' \,\big)_{H',F'}}{\big(\, \mathit{cmd};\ \mathit{seq} \,\big)_{H,F} \rightsquigarrow \big(\, \mathit{seq}';\ \mathit{seq} \,\big)_{H',F'}}\ \mathit{seq}$$

$$\frac{}{\big(\, \mathit{skip};\ \mathit{seq} \,\big)_{H,F} \rightsquigarrow \big(\, \mathit{seq} \,\big)_{H,F}}\ \mathit{skip}$$

$$\frac{}{\big(\, \mathit{br}\ l \,\big)_{H,(l_p,\ l_c,\ \beta)} \rightsquigarrow \big(\, P(l) \,\big)_{H,(l_c,\ l,\ \beta)}}\ \mathit{branch}_1$$

$$\frac{(\beta(r) = \mathit{true} \wedge l = l_1) \vee (\beta(r) = \mathit{false} \wedge l = l_2)}{\big(\, \mathit{br}\ r,\ l_1,\ l_2 \,\big)_{H,(l_p,\ l_c,\ \beta)} \rightsquigarrow \big(\, P(l) \,\big)_{H,(l_c,\ l,\ \beta)}}\ \mathit{branch}_2$$

$$\frac{\begin{array}{c} f \llparenthesis a_1, \ldots, a_n \rrparenthesis \in \mathit{dom}(P) \\ F_0 = (\bot,\ f \llparenthesis a_1, \ldots, a_n \rrparenthesis,\ \{a_1 \mapsto F(r_1), \ldots, a_n \mapsto F(r_n)\}) \\ \big(\, P(f \llparenthesis a_1, \ldots, a_n \rrparenthesis) \,\big)_{H,F_0} \rightsquigarrow^{*} \big(\, \mathit{ret} \,\big)_{H',F'} \end{array}}{\big(\, \mathit{pcall}\ f(r_1, \ldots, r_n) \,\big)_{H,F} \rightsquigarrow \big(\, \mathit{skip} \,\big)_{H',F}}\ \mathit{pcall}$$

$$\frac{\big(\, \mathit{expr} \,\big)_{H,F} \rightsquigarrow \big(\, v \,\big)_{H'}}{\big(\, x = \mathit{expr} \,\big)_{H,F} \rightsquigarrow \big(\, \mathit{skip} \,\big)_{H',F \oplus \{x\ \mapsto\ v\}}}\ \mathit{assign}$$

$$\frac{F(r_p) = p \quad H(p) = (t, d) \quad H' = \{p \mapsto (t, F(r_{\mathit{new}}))\}}{\big(\, \mathit{store}\ r_{\mathit{new}},\ r_p \,\big)_{H,F} \rightsquigarrow \big(\, \mathit{skip} \,\big)_{H \oplus H',F}}\ \mathit{store}$$

Fig. 7. Rules for sequences and commands.

frame such that, $F(x_{mset}) = p_{mset}$ is a pointer on a multiset structure storing elements of type t. Under these conditions, procedure add_{mset} is specified by:

$$\big(\, pcall\ add_{mset}(x_{mset}, x)\,\big)_{H,F} \leadsto \big(\, skip\,\big)_{H \oplus H', F} \tag{1}$$

$$dom(H) \cap dom(H') \subseteq p_{mset}{\downarrow_H} \tag{2}$$

$$[\![H \oplus H', p_{mset}]\!]^{mset(t)} = [\![H, p_{mset}]\!]^{mset(t)} + \{[\![H, F(x)]\!]^{t}\} \tag{3}$$

Condition (1) describes by a reduction the result of the call, condition (2) restrict the updates to be localised in the heap and condition (3) interprets the computation in terms of Petri nets objects. Similarly, any implementation of the marking structures has to respect the two following requirements.

Requirement 2 (Soundness) *Let H, H' be two heaps, F, F' two frames, $p_{mrq} \in dom(H)$ a pointer to a marking structure, and p_s a pointer to a place nested in p_{mrq} (*i.e.*, $p_s \in p_{mrq}{\downarrow_H}$). If $[\![H, p_{mrq}]\!]^{mrq}(s) = [\![H, p_s]\!]^{s}$, $\big(\, seq\,\big)_{H,F} \leadsto \big(\, seq'\,\big)_{H \oplus H', F'}$ and $p_{mrq} \notin dom(H')$ then*

$$[\![H \oplus H', p_{mrq}]\!]^{mrq}(s) = [\![H \oplus H', p_s]\!]^{s}$$

Requirement 3 (Separation) *Let p_{mrq} be a pointer on a structure marking in a heap H. If p_s and $p_{s'}$ are pointers to distinct places in this structure then we have $p_s{\downarrow_H} \cap p_{s'}{\downarrow_H} = \emptyset$.*

The *soundness* property ensures that any update of a place through a pointer returned by get_s is actually made on the marking (not on a copy). The *separation* property ensures that places do not share memory so that updating a place does not have side effects on other places.

6 Correctness and termination results

We present now the two main results proving the correctness of functions $fire_t$ (theorem 1) and $succ_t$ (theorem 2). Both these results are shown in a minimal context, *i.e.*, a heap that just contains the required pointers. An auxiliary theorem (not presented here) allows to generalise both results to any context that includes the minimal one.

Theorem 1. *Let M be a marking, H a heap and p_{mrq} a pointer on a marking structure such that $dom(H) = p_{mrq}{\downarrow_H}$ and $[\![H, p_{mrq}]\!]^{mrq} = M$. Let $\beta \overset{\text{df}}{=} \{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\}$ be a LLVM mode for transition t, which implies that each v_i is a value or a pointer encoding a token in place s_i: $[\![H, v_i]\!]^{\ell(s_i)} \in M(s_i)$. Let F be a frame such that $\beta_F \overset{\text{df}}{=} \beta \oplus \{x_{mrq} \mapsto p_{mrq}\}$. If*

$$M[t, \beta\rangle M' \quad \text{and} \quad \big(\, fcall\ fire_t(x_{mrq}, x_1, \dots, x_n)\,\big)_{H,F} \leadsto \big(\, p'_{mrq}\,\big)_{H \oplus H'}$$

then

$$[\![H \oplus H', p'_{mrq}]\!]^{mrq} = M' \quad \text{and} \quad dom(H) \cap dom(H') = \emptyset$$

Proof (sketch). This theorem is the direct application of two lemmas showing the correctness of tokens consumption and production respectively. Both are proved by induction on the number of places in the marking structure. □

Corollary 1. *Under the same hypothesis, the call to* fire_t *terminates.* □

Theorem 2. *Let* F *be a frame and* p_{mrq} *a pointer in a heap* H *such that* $p_{mrq}\!\downarrow_H = dom(H)$, $[\![H, p_{mrq}]\!]^{mrq} = M$, $\beta_F(x_{mrq}) = p_{mrq}$ *and* $\beta_F(x_{next}) = p_{next}$*. If*

$$\big(\mathit{fcall}\ \mathit{succ}_t(x_{mrq}, x_{next})\big)_{H,F} \rightsquigarrow^* \big(p_{next}\big)_{H\oplus H'} \quad \text{and} \quad [\![H, p_{next}]\!]^{set} = E$$

where E *si a set of markings, then*

$$dom(H) \cap dom(H') = \emptyset$$

$$[\![H \oplus H', p_{next}]\!]^{set} = E \cup \{M' \mid \exists\beta,\ M[t,\beta\rangle M'\}$$

Proof (sketch). The first result is a consequence of the formal interfaces of the called functions. The second result is proved as two inclusions:

$\supseteq$. This is the consequence of two lemmas:
  - by applying reduction rules, we show that if execution goes through a block $\mathit{header}_{t,k}$ then it will necessarily reach a block $\mathit{footer}_{t,k+1}$, and every block annotated by an index greater than k must be executed also;
  - consequently, all combinations of tokens for the input places are actually enumerated, which implies that all potential modes of t are considered.

$\subseteq$. To prove that only actual successor markings are added, we first remark that if a marking is added into the set, then this happens in block $\mathit{body}_{t,0}$. So it is enough to prove that this block is executed only if the binding is actually a mode for t, which can be proved using the reduction rules backward to show that the guard necessarily evaluated to *true*. □

Corollary 2. *Under the same hypothesis, the call to* succ_t *terminates.* □

7 Conclusion

We have shown how a Petri net can be compiled targeting a fragment of the LLVM language. This compilation produces code that provides the primitives to compute the state space of the compiled Petri net model. Then we have defined a formal semantics for the fragment of the LLVM language we use. To produce a readable and usable system of inference rules, we have defined a memory model based on explicit heaps and stacks. Finally we have proved the correctness of the code generated by our compiler. The full proofs provided in [5, 4] are quite long because they are very much detailed to improve our confidence into their correctness and to ease there later validation using a proof assistant. But notice also that they are at the same time quite easy to follow. It is worth noting also that our proofs are modular thanks to clearly defined interfaces with appropriate axiomatisation. As a consequence, we should avoid issues when parts of the generated code are replaced, for instance, to use a more efficient data structure, or alternative state space exploration approaches (like in [8]).

The fragment of LLVM we have considered is rather limited with respect to the number of instructions. However, it is at the same time quite representative of the full language. Indeed, it includes the necessary to handle the stack and the heap which are conceptually the most complicated parts of the language. Extending our fragment to include all the LLVM computational instructions (like arithmetic) would be an easy but tedious work. Adding the instructions to manipulate the stack (like *unwind* for exception handling) looks quite straightforward. The most complicated is probably adding full support for pointers, which would required to refine our heap model (in particular, pointer arithmetic would have to be defined).

In this paper, we have considered an "optimistic" approach in that we assume that the LLVM code provided in the model annotations is correct and terminates. Moreover, we did not make any assumption about the finiteness of the state space or the boundedness of integer values that are assumed not to overflow. In practice, this are however important issues. Approaches based on abstract interpretation of assembly code like [18] may be helpful to prove such properties on the compiled model before to start the state space exploration, ensuring that it will run safely (and allowing to avoid implementing checks in the generated code).

Future works will address a generalisation of the presented approach to compile a wider variety of coloured Petri nets, in particular nets embedding annotation languages easier to use for the modeller than LLVM. Moreover, we would like to refine requirement 3 to allow for a logical separation instead of a physical separation as it is currently defined. This would enable us for implementing memory sharing and thus saving a lot of memory during a state space exploration. We are also interested in particular in exploiting remarkable structures of Petri net models that allow to optimise the code generated by the compiler. Such optimisations also need to be formally proved and preliminary results about this can be found in [4]. A longer term goal is to prove the whole compilation chain to obtain the core (*i.e.*, state space exploration) of a certified explicit model-checker for coloured Petri nets. A complementary aspect is to evaluate the performance of the state space generation, which is of course another important motivation when working on a model-checker. As shown in [6], the current implementation is efficient and can outperform state-of-the-art tools. So, certification is not an objective that contradicts efficiency.

References

1. E. Clarke, A. Emerson, and J. Sifakis. Model checking: Algorithmic verification and debugging. *ACM Turing Award*, 2007.

2. ADT Coq/INRIA. The Coq proof assistant. ⟨http://coq.inria.fr⟩.
3. S. Evangelista. *Méthodes et outils de vérification pour les réseaux de Petri de haut niveau*. PhD thesis, CNAM, Paris, France, 2006.
4. L. Fronc. Analyse efficace des réseaux de Petri par des techniques de compilation. Master's thesis, MPRI, university of Paris 7, ⟨http://www.ibisc.fr/~lfronc/pub/LF_2010.pdf⟩, 2010.
5. L. Fronc and F. Pommereau. Proving a Petri net model-checker implementation. Technical report, IBISC, University of Évry, ⟨http://goo.gl/WMzhp⟩, 2010.
6. L. Fronc and F. Pommereau. Optimizing the compilation of Petri nets models. In *Proc. of SUMo'11*, volume 726. CEUR, ⟨http://ceur-ws.org/Vol-726⟩, 2011.
7. G.J. Holzmann and al. Spin, formal verification. ⟨http://spinroot.com⟩.
8. G.J. Holzmann, D. Peled, and M. Yannakakis. On nested depth-first search. In *Proc. of the 2nd Spin Workshop*. AMS, 1996.
9. K. Jensen and L.M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer, 2009, ISBN 978-3-642-00283-0.
10. C. Lattner. LLVM language reference manual. ⟨http://llvm.org/docs/LangRef.html⟩.
11. C. Lattner and al. The LLVM compiler infrastructure. ⟨http://llvm.org⟩.
12. C. Lattner and al. LLVM related publications. ⟨http://llvm.org/pubs⟩.
13. C. Lattner and al. LLVM users. ⟨http://llvm.org/Users.html⟩.
14. C. Pajault and S. Evangelista. Helena: a high level net analyzer. ⟨http://helena.cnam.fr⟩.
15. L. Paulson, T. Nipkow, and M. Wenzel. The Isabelle proof assistant. ⟨http://www.cl.cam.ac.uk/research/hvg/Isabelle⟩.
16. F. Pommereau. Quickly prototyping Petri nets tools with SNAKES. *Petri net newsletter*, 2008.
17. C. Reinke. Haskell-coloured Petri nets. In *IFL'99*, volume 1868 of *LNCS*. Springer, 1999.
18. Xavier Rival. *Traces Abstraction in Static Analysis and Program Transformation Abstraction de Traces en Analyse Statique et Transformations de Programmes*. PhD thesis, Computer Science Department, École Normale Supérieure, 2005.
19. K.N. Verma, J. Goubault-Larrecq, S. Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In *ASIAN'00*, volume 1961 of *LNCS*. Springer, 2000.