This document presumes the definition of Term algebras and parameterized actions from the pNet theory definition, that we will not repeat here.

1 Translations from pNets to SMTLib

In order to submit satisfiability problems to Z3 (for the predicates in open transitions), we need to generate SMTlib programs, from the pNet Algebra presentation and predicates. More precisely, we need to translate to SMTlib:

- for a given pNet: the presentation of the action algebra (sorts, operators, constants),
- for each open transition: the declaration of variables, and the predicate (including action expressions).

During this translation, in order to guarantee that the generated code will cause no runtime errors during parsing and execution, we need to ensure that all objects used in the SMTlib code are properly declared, and that they are correctly typed.

2 Type system

The type system in pNets is quite simple, the expressions are built from variables, constants, and operators of a many-sorted algebra, and we only need simple first-order polymorphic types. Additionaly, for convenience, we want to introduce some "generic" operators in the algebra, with an overloading mechanism; as these do not exist in SMTlib, we will have a specific translation mechanism.

Note that in principle, such an algebra correspond to a given high-level language (e.g. a process algebra), and that the algebra presentation will be defined once and for all in the framework of the pNet semantics of each specific language.

2.1 Algebra Presentations

We have a minimal, predefined algebra presentation for all pNets, including three basic sorts *Bool*, *Action* and *Int* and their operators. Table 1 defines these elements.

Table 1. Algebra Presentation: predefined Sorts and Operators

Sort	Operator
Bool	$\wedge$, $\vee$, `true`, `false`
Action	*no predefined operator*
Int	$-$`(unary)`, $+$, $-$`(binary)`, $\times$, $\div$, $0, 1, ... \in$ `Nat`
for any sort	$=$, $\neq$

For a given language, or for a given use-case, the designer can declare more sorts and operators, using our pNet API. As an example, the CCS action algebra include:

2.2 Type Rule

The environment is a set of variables with their own types like $x_1 : A_1, ..., x_n : A_n$. And use dom($\Gamma$) dedicate the collection of $x_1, ..., x_n$. Let $\mathcal{P}$ to be the presentation of the pNet.

Table 2. Judgments for Open pNets

$\Gamma \vdash \diamond$	Γ is a well-formed environment
$\Gamma \vdash A$	A is a well-formed type in Γ
$\Gamma \vdash M : A$	M is a well-formed term of type A in Γ

2.3 Map to SMT-LIB language

The pNet semantics can be full translated into SMT-LIB language, though some difference on defining functions exist.

3 Submission to Z3

3.1 Z3 Notations

Our semantic has already got enough informations for Z3 checking. Z3 has a number of notations for various usages. We only list the notations we needed in our algorithm as follows: **[TODO:Finish the description.]**

1. `declare-datatypes`
2. `declare-function`
3. `declare-const`
4. `assert`

3.2 Submission of Algebra

Before the submission of the open transitions to Z3, the user-inputed algebra of the pNet should be known at first. It gives Z3 informations of the sorts of the expression and the operators, corresponding to the `declare-datatypes` and `declare-function`. We declare the algebra in Z3 logic through Z3 API methods having the same effect as the notations. At the same time, we have several internal lists (actually hash maps) `exprs`, `funcDecls`, `sortDatatypes` to store the generated Z3 objects with their names for later proofs.

Table 3. Type Rules for Open pNets

(Env $\varnothing$)

$$\frac{}{\varnothing \vdash \diamond}$$

(Type Bool)
$$\frac{\mathcal{P} \vdash \mathit{Bool} \quad \Gamma \vdash \diamond}{\mathcal{P}, \Gamma \vdash \mathit{Bool}}$$

(Type Action)
$$\frac{\mathcal{P} \vdash \mathit{Action} \quad \Gamma \vdash \diamond}{\mathcal{P}, \Gamma \vdash \mathit{Action}}$$

(Type Int)
$$\frac{\mathcal{P} \vdash \mathit{Int} \quad \Gamma \vdash \diamond}{\mathcal{P}, \Gamma \vdash \mathit{Int}}$$

(Var x)
$$\frac{\mathcal{P} \vdash A \quad \Gamma \vdash x : A}{\mathcal{P}, \Gamma \vdash x : A}$$

(Binary operators, e.g.: $\wedge, \vee$ for booleans, $+, -, \times, \div, \leq, \geq$ for integers, etc.)

$$\frac{\mathcal{P} \vdash \mathit{BinOp} :: ty1, ty1 \to ty2 \quad \Gamma \vdash x_1 : ty1 \quad \Gamma \vdash x_2 : ty1}{\mathcal{P}, \Gamma \vdash x_1 \ \mathit{BinOp}\ x_2 : ty2}$$

(Unary operators, e.g. $\neg$ for booleans, - for integers)

$$\frac{\mathcal{P} \vdash \mathit{UnOp} :: ty1 \to ty2 \quad \Gamma \vdash x : ty1}{\mathcal{P}, \Gamma \vdash \mathit{Unop}\ x : ty2}$$

(Polymorphic EQ and NEQ)
(Note: I removed the $\mathcal{P} \vdash \neq :: A, A \to \mathit{Bool}$)

$$\frac{\mathcal{P} \vdash A \quad \Gamma \vdash x_1 : A \quad \Gamma \vdash x_2 : A}{\mathcal{P}, \Gamma \vdash x_1 = x_2 : \mathit{Bool}} \qquad \frac{\mathcal{P} \vdash A \quad \Gamma \vdash x_1 : A \quad \Gamma \vdash x_2 : A}{\mathcal{P}, \Gamma \vdash x_1 \neq x_2 : \mathit{Bool}}$$

(Overloaded FUN)

$$\frac{\mathcal{P} \vdash \texttt{FUN} :: A_1, ..., A_n \to A \quad \mathcal{P} \vdash A_1 \quad ... \quad \mathcal{P} \vdash A_n \quad \Gamma \vdash x_1 : A_1 \quad ... \quad \Gamma \vdash x_n : A_n}{\mathcal{P}, \Gamma \vdash \texttt{FUN}(x_1, ..., x_n) : A}$$

Table 4. Mapping

(Presentation)	
	$Sort \hookrightarrow$ declare-datatypes
	$Operator \hookrightarrow$ declare-function
(Checking)	
	$dom(\Gamma) \hookrightarrow$ declare-const
	$Pred \hookrightarrow$ assert
(Expressions)	
	$\texttt{FUN}(x_1, ..., x_n)$
	$x_1 - x_n \quad \hookrightarrow$ declare-function
	...

3.3 Submission of Open Transitions

Each time we submit each open transition to Z3 module, we translate its predicate into Z3 language format and send it for satisfiability checking. Every term of the predicate is declared as an `assert` in Z3. A constant action or a parameterized expression is easy to get from the internal list storing the objects while all the variables are not declared at the beginning. So we declare them before the submission of a predicate term with the API method conducting `declare-const`.

3.4 Other works

Quantifier

Filter the State without Precursor