Properties specification language for algebraic Petri nets

MARECHAL MARIN, Alexis Ayar, BUCHS, Didier

Abstract

Model Checking consists in verifying if a model of a given system meets a set of requirements. The model and the properties can be specified with various formalisms. We chose Algebraic Petri Nets (APNs), a powerful formalism used to model concurrent systems, for the model specification. This report proposes a language for defining invariant properties on APNs. The expressiveness of the language proposed is similar to first order logic, with some extensions (deadlocks and cardinalities). These properties can be used to define the requirements for model checking. This language was created for the tool AlPiNA, an APN model checker developed at the University of Geneva.

Reference

Technical report #216 (April 15, 2011)

Properties specification language for algebraic Petri nets
Abstract

Model Checking consists in verifying if a model of a given system meets a set of requirements. The model and the properties can be specified with various formalisms. We chose Algebraic Petri Nets (APNs), a powerful formalism used to model concurrent systems, for the model specification. This report proposes a language for defining invariant properties on Algebraic Petri Nets (APNs). The expressiveness of the language proposed is similar to first order logic, with some extensions (deadlocks and cardinalities). These properties can be used to define the requirements for model checking. This language was created for the tool AlPiNA, an APN model checker developed at the University of Geneva.

**Keywords:** Model checking, Algebraic Petri Nets, Properties, Specification language, first-order logic, AlPiNA.
1 Introduction

AlPiNA [BHMR10a, BHMR10b] is a model checking tool: it allows the creation of models, using
the Algebraic Petri Nets (APNs) formalism, and the validation of invariant properties in the
state space of this model.

This document is meant to describe the language used to define these properties. All the
properties obtained are invariants: the tool checks a property expressed in first order logic on
every state of the state space. Thus, a property is a boolean expression expressed on single states
of the Petri Net state space. If the property does not hold for at least one state in the state
space, we say that the property is invalid, and one counterexample is returned. As of version
1.0 of Alpina, only one counterexample is returned and the tool does not provide a trace of the
transitions fired to reach this counterexample. Please note that reachability properties do not
include properties defined in temporal logics, as CTL or LTL. This work was greatly inspired by
the corresponding language used in Helena [PE07].

Our language is mainly composed of expressions. There are three types of expressions:

- The set of boolean expressions noted $expr_B$
- The set of natural expressions noted $expr_Z$
- The set of term expressions noted $expr_T$

The document is organised as follows: the two first sections (Section 2 and Section 3) explain
all the expressions that compose the properties language: first the basic expressions, which
are rather straightforward, and then the iterated expressions, that give the true power of the
language. Section 4 explains some details about a handy feature on the language: the named
expressions. Section 5 explains how to integrate the previously explained expressions to actually
write properties, and Section 6 gives some examples, from the simplest to some more advanced
ones. Finally, Section 7 gives the complete syntax specification of the language.

2 Basic Expressions

2.1 Basic Boolean Expressions

A boolean expression is an expression that can be evaluated to true or false. For now, we will
describe 4 different types of boolean expressions:

- constants: $TRUE \in expr_B$ and $FALSE \in expr_B$
- negative expressions (the negation of a boolean expression):
  $e \in expr_B \Rightarrow \neg e \in expr_B$
- composed expressions, built with three usual boolean operators:
  $e, e' \in expr_B \Rightarrow e \& e' \in expr_B$ (and operator)
  $e, e' \in expr_B \Rightarrow e \mid e' \in expr_B$ (or operator)
  $e, e' \in expr_B \Rightarrow e \Rightarrow e' \in expr_B$ (implies operator)
2.2 Deadlock

- equals expressions, which evaluate if two expressions are equal or not:
  \[ e, e' \in expr_B \Rightarrow e = e' \in expr_B \]
  \[ e, e' \in expr_Z \Rightarrow e = e' \in expr_Z \]
  \[ e, e' \in expr_T \Rightarrow e = e' \in expr_B \]
  \[ e, e' \in expr_B \Rightarrow e !\neq e' \in expr_B \]
  \[ e, e' \in expr_Z \Rightarrow e !\neq e' \in expr_B \]
  \[ e, e' \in expr_T \Rightarrow e !\neq e' \in expr_B \]

We can check the equality between two expressions of the same type. For two terms, the equality is verified if both terms have the same normal form.

The semantics of all these expressions is trivially described by the corresponding expressions in propositionnal logic. In the tool, composed expressions that have more than 2 elements (e.g. TRUE & FALSE | TRUE) must use parenthesis to indicate the precedence of the operators, otherwise a syntactical error is detected.

2.2 Deadlock

As stated in the introduction, properties in AlPiNA are only checked in individual states, without considering the firing of the transitions. There is one exception to this rule: the deadlock. A deadlock occurs on a state if no transition can be fired from it, i.e. the system is blocked. Thus, a deadlock check is not only performed by looking the state itself, but also the firing of the transitions from this state. This means that there is a strong conceptual difference between the check of a deadlock expression and the check of any other expression presented here.

Nevertheless, syntactically, a deadlock expression is just a boolean expression, expressed with the keyword "Deadlock":

- \[ \text{Deadlock} \in expr_B \]

Deadlocks are boolean expressions, and thus they can be composed with other expressions to perform advanced checks. Take the example of a model where there exists some fault detection mechanism: if the system finds an error, a token is added into some place called ERROR and the whole system stops. To check the correct behavior of this net, we would want to verify that every state where the place ERROR is not empty is a deadlock. The following expression allows such a check:

\[ \text{exists($x$ in ERROR)} \Rightarrow \text{Deadlock} \]

2.3 Basic Integer Expressions

An integer expression is an expression whose evaluation returns an integer. For now, we describe 3 simple types of integer expressions:

- constants: \[ x \in \mathbb{N} \Rightarrow x \in expr_Z \]
- minus expressions: \[ e \in expr_Z \Rightarrow -e \in expr_Z \]

\[ The \ expression \ \text{exists($x$ in ERROR)} \ checks \ that \ the \ place \ ERROR \ is \ not \ empty, \ more \ details \ on \ Section \ 3.3.1 \]
2.4 Term expressions

- composed expressions, built using the usual integer operators:
  \[ e, e' \in expr_Z \Rightarrow e + e' \in expr_Z \]
  \[ e, e' \in expr_Z \Rightarrow e - e' \in expr_Z \]
  \[ e, e' \in expr_Z \Rightarrow e \times e' \in expr_Z \]
  \[ e, e' \in expr_Z \Rightarrow e/e' \in expr_Z \]

Please note that the operations used are integer ones, the division behaves exactly as the Java division. As for the boolean expressions, precedences must be explicited with parenthesis.

2.4 Term expressions

A term expression is simply a term from an ADT, it can eventually take variables.

- \( t \in T_{\Sigma,X} \Rightarrow t \in expr_T \)

These expressions can only be used as members of equals expressions (see section 2.1). The evaluation of a term expression is simply its normal form, eventually after the substitution of the variables (see section 3).

3 Iterated Expressions

Iterated expressions use iterators to perform checks on sets of terms. Two types of sets can be iterated: the content of a place of the APN, or the terms of a given sort\(^2\).

We use three operators to iterate over sets: the two quantifiers from first order logic (i.e. \( \forall \) and \( \exists \)), and an observer to count the terms of a given set, called \( \text{card} \).

3.1 Iterators

An iterator is a couple \(<\text{Variable}, \text{Domain}>\), where the domain can be either the content of a Place or the elements of a given Sort. Iterators use variables to perform iterations. Each variable has a sort, defined in the Algebraic Data Types (ADTs). We note \( X_s \) the set of variables of the sort \( s \). Moreover, in APNs, places must also have a sort. We note \( \sigma(p) \) the sort of a place \( p \).

The set \( It \) of iterators is defined as follows:

- \( p \in P, x \in X_{\sigma(p)} \Rightarrow [x \text{ in } p] \in It \) (iterator over the content of a place)
- \( s \in S, x \in X_s \Rightarrow [x \text{ of } s] \in It \) (iterator over the terms of a sort)\(^2\)

The iteration over sorts can only be performed on unfolded sorts. Algebraic unfolding is an advanced feature of AlPiNA, described in the cited articles. Informally, we can say that in AlPiNA each sort can be unfolded, i.e., the values of this sort can be generated during a preprocessing phase, prior to the state space computation. This unfolding can be total for finite sorts, or partial. Thus, a sort iterator will iterate over the set of values generated by the unfolding for the corresponding sort.

\(^2\)As of version 1.0, AlPiNA does not support the iteration over sorts. This will be implemented in a future version. For now only the content of a place can be iterated.
3.2 Composition of iterators

A single iterated expression can work on more than one iterator at the same time. For this, we use a composition of iterators. These iterators will iterate over the Cartesian product of all the domains. For example, the composition of iterators $x$ in $P_1$, $y$ in $P_2$ will iterate over the cartesian product of the contents of $P_1$ and $P_2$: each pair of elements from these places will be considered. For more details please check the examples in Section 6.

We inductively define the set of iterators composition $It_c$ as follows:

- $it \in It \Rightarrow [it] \in It_c$
- $it \in It, it_c \in It_c \Rightarrow [it, it_c] \in It_c$

3.3 Iterated Expressions

3.3.1 Boolean Iterated Expressions

An iterated expression uses a composition of iterators to explore the values in a given set, and evaluates a boolean expression on each one of these values. We first introduce the expressions that use the two well known operators from first order logic, $\forall$ and $\exists$:

- $it_c \in It_c, e \in expr_B \Rightarrow \forall(it_c : e) \in expr_B$
- $it_c \in It_c, e \in expr_B \Rightarrow \exists(it_c : e) \in expr_B$
- $it_c \in It_c \Rightarrow \exists(it_c) \in expr_B$ (the expression to be checked is implicitly equal to $\text{TRUE}$)

In the tool, there is a special restriction on the used variables: free variables are forbidden, i.e., each variable found in a Term expression must be bound by an iterator.

For example:

$\forall(x \in P : x = \text{zero})$ will be evaluated to 'true' if for every marking of the APN state space, the place $P$ contains only tokens equal to zero, whereas $\exists(x \in P : x = \text{zero})$ will be evaluated to 'true' if, for every marking of the APN state space, there is at least one token in $P$ equal to zero.

Like in first order logic, if $P$ is empty at some point, the first expression will be evaluated to 'true' and the second one to 'false'.

3.3.2 Integer Iterated Expressions

In AlPiNA, we add a new operators that works over sets of elements called $\text{count}$. It is used to count the number of elements that verify a given boolean expression. We call this the $\text{card}$ operator. Note that APN use multisets as the markings of places, i.e. each element can be present more than once. If this is the case, $\text{card}$ expressions will count each element multiple times (see the last example of Section 6). We define:

- $it_c \in Iterators_c, e \in expr_B \Rightarrow \text{card}(it_c : e) \in expr_Z$
- $it_c \in Iterators_c \Rightarrow \text{card}(it_c) \in expr_Z$ (the expression to be checked is implicitly equal to $\text{TRUE}$)
For example:
\[
\text{card($x$ in } P : $x = \text{zero})\text{ counts the number of tokens in } P \text{ equal to zero.}
\]
\[
\text{card($x$ in } P \text{) simply count all the tokens in } P, \text{ it is equal to card($x$ in } P : \text{TRUE}).
\]
\[
\text{card($x$ in } P_1, \text{ } y \text{ in } P_2 \text{) counts the number of elements in the set } P_1 \times P_2. \text{ This is different from computing:}
\]
\[
\text{card($x$ in } P_1) + \text{card($y$ in } P_2)
\]

3.4 Syntax shortcut for cardinalities

There is a small syntax shortcut for cardinalities, if the condition to be checked is \text{TRUE} and there is only one iterator, they can be both omitted:
\[
\text{card($x$ in } P : \text{TRUE}) \text{ can be simply written card(P). The same also applies for sort iterators. Thus, we get:}
\]
\[
\begin{align*}
\bullet & \ p \in P \Rightarrow \underbrace{\text{card}(p)}_{\in expr_Z} \\
\bullet & \ s \in S \Rightarrow \underbrace{\text{card}(s)}_{\in expr_Z}
\end{align*}
\]

4 Named expressions

To help the user with the properties definition, we added the possibility to define boolean expressions with a name and use those expressions simply by referencing the name. For example, if the user wants to write a big property that is a conjunction of two big boolean expressions, i.e.,

\[
\text{BigBooleanExpression1 \& BigBooleanExpression2}
\]

He can write the two boolean expressions separately and then create a boolean expression that references the two declarations. This allows some modularity boolean expressions.

Formally, we define a set of name expressions namedExpr:
\[
\begin{align*}
\bullet & \ \text{id} \in \text{String, } e \in expr_B \Rightarrow \underbrace{\text{id} : e}_{\in namedExpr}
\end{align*}
\]

Let \text{Ids} the set of identifiers used in named expressions, we can use this id to reference the contained expression:
\[
\begin{align*}
\bullet & \ \text{id} \in \text{String} \Rightarrow \underbrace{\text{@id}}_{\in expr_B}
\end{align*}
\]

It is extremely important to understand that this mechanism is purely syntactical: making a reference to \text{@Expr} is exactly like copying and pasting the expression referenced by the identifier \text{Expr}. This heavily affects the variables substitutions. For example, it’s possible to write:

\begin{align*}
\text{Equals} : x = \text{zero}; \\
\text{AlwaysEquals} : \forall(x \in P : \text{@Equals});
\end{align*}

In this example the expression \text{Equals} cannot be checked alone, because there is no iterator to define values for the variable \text{x}. The second expression is strictly equivalent to:

\begin{align*}
\text{AlwaysEquals} : \forall(x \in P : x = \text{zero});
\end{align*}

It has an iterator for \text{x}, so it is a correct boolean expression that can be used in a check\footnote{as long as the sort of \text{x} and the sort of the place \text{P} are both of the same sort as the term \text{zero}, as seen in Section 3.}.
5 Properties

A property to check in AlPiNA is a boolean expression. Only one property (thus only one boolean expression) can be checked at a time. We give here an idea of the syntax used to define the properties in the tool. The complete syntax is presented in Section 7.

```
<Import>**
Expressions
  <namedExprB>*
Check
  <exprB>
Variables
  <VariablesDeclarations>*
```

The file begins with a list of imports. All the files that define the used Sorts, and the file containing the APN must be imported. The section Expressions contains a list of named expressions (see Section 4). The section Check contains a single boolean expression. This expression will be the actual property checked on the model.

6 Examples

6.1 Producer Consumer

For the following examples, we will consider the Petri net from Figure 1. It originates from [Jen97]. The model describes producers (place Prod), who produce (transition produce) and send (transition send) a packet — identified by a producer and a consumer — via a buffer (place Buff). Consumers (place Cons) receive (transition receive) and consume (transition consume) packets. All used ADTs are presented in the Appendix. Some elements of the model should be highlighted:

- In the example of Figure 1, there are only two producers and one consumer. More producers (resp. consumers) would be added by adding some tokens in the place Prod (resp. Cons).
- The output arc of transition produce contains variable $c$ of sort consumer. This is a free variable, an output variable that is not bound to an input variable. Therefore the term pk(prod, c) will be unified to the input variable prod and to any consumer.
- The transition send has a guard: isNotFull(b). This enables the firing of send only if the input variable $b$ — that represents the current state of the buffer — is not full (thus if isNotFull(b) is evaluated to true). The function push(p, b) on the output arc queues packet $p$ in buffer $b$. On the other output arc, getProd(p) extracts the producer from packet $p$.
- The guard of transition receive is a conjunction of two predicates: isNotEmpty(b) that is true when the buffer is not empty and getCons(head(b))= c that checks whether the current consumer is the actual recipient of the packet extracted from $b$ (using the function head).
Figure 1: Algebraic Petri net of a producers/consumers model
There are some simple yet interesting properties that we could check in this model. For example, we see that the place Buff is never empty: there is always a token representing the buffer, that can be empty, full, or have some values. We can check that this token is always there by checking that the place Buff is never empty:

\[ \text{exists}(b \text{ in Buff}) \]

A better check would be to verify that there is always one and only one token in that place:

\[ \text{card}(\text{Buff}) = 1 \]

Moreover, one could mistakenly think that the buffer in place Buff will never be full. To perform this check, we could write:

\[ \forall b \in \text{Buff} : \text{isNotFull}(b) = \text{true} \]

Here we are comparing two terms: \( \text{isNotFull}(b) \) and \( \text{true} \). It is very important to understand that \( \text{true} \) is a term from the booleans ADT, not the boolean constant \( \text{TRUE} \), written in caps (see Section 2.1). \( \text{isNotFull} \) is an operator, also defined in the ADTs, that takes a buffer as parameter and returns the term \( \text{true} \) if this buffer is not full.

The latest property will not hold in our Petri net: indeed, the transition send cannot be fired if the buffer is already full, thus it may be fired until this condition is met. What we want to check in fact is that there is not a buffer overflow, i.e., a situation where the buffer has more than the allowed number of elements. For this, we could use the operator \( \text{size} \) in the Buffers ADT, and the operators \( \text{st} \) ("smaller than") defined in the Naturals ADT:

\[ \forall b \in \text{Buff} : \text{size}(b) \text{ st suc}^3(\text{zero}) = \text{true} \]

This is equivalent to:

\[ \exists b \in \text{Buff} : \text{size}(b) \text{ st suc}^3(\text{zero}) = \text{false} \]

Another interesting property is that the producers never disappear: they are always either idle, either producing a packet (places Prod and Pk_p). To check this, we could check that there are always two tokens in the union of both places:

\[ \text{card}(\text{Prod}) + \text{card}(\text{Pk_p}) = 2 \]

\(^4\)The variable b must have been declared with the sort B, the same sort as the place Buff, see the ADTs in the appendix for details.
We could also check that the same producer is never present in both places at the same time. This is a bit trickier, because the place \( P_{k,p} \) stores packets, i.e., pairs of producers and consumers. To make the comparison, we must use the \( \text{getProd} \) operator. With this operators, we check that a producer cannot be in the place \( \text{Prod} \) and \( P_{k,p} \) at the same time:\footnote{The variable \( p \) must be of the sort \( P \), and the variable \( pa \) of the sort \( K \).}

\[
\forall (p \in \text{Prod}, \, pa \in P_{k,p} : p \neq \text{getProd}(pa))
\]

The check of this property will take each pair of tokens from places \( \text{Prod} \) and \( P_{k,p} \) and check that they do not correspond to the same producer.

Finally, we could check that the producers never disappear: each producer must be either in the place \( \text{Prod} \), or in a packet from place \( P_{k,p} \). For this, we could iterate over the sort \( P \) of producers:\footnote{Note that iterating over sorts is not implemented in AlPiNA 1.0, to perform this check in the current implementation, the user should create one expression for each producer.}

\[
\forall (p \in P : \\
\exists (p1 \in \text{Prod} : p = p1) \lor \\
\exists (pa \in P_{k,p} : p = \text{getProd}(pa)))
\]

For each producer \( p \), we check that either there is a token \( p1 \) in the place \( \text{Prod} \) equal to \( p \), either there is a token in place \( P_{k,p} \) whose producer is equal to \( p \). We use the "or" operator to perform the conjunction between both expressions.

### 6.2 A more complex example

We will now use another APN for a more elaborate property example. This should show some more complicated features of the language. Consider a model of a client/server protocol: several clients make requests to access one of several processes. An excerpt of such a net is shown in Figure 2.

The place \( \text{Requests} \) contains the clients requests (note that there can be more than one request for each client), the place \( \text{Idle} \) contains the list of idle servers, and the transition \( \text{connect} \) links a client request with a server. Such system is naturally very concurrent. We could imagine that we are testing some protocol that would turn off the concurrency in this system: somehow, the protocol limits the possible connections using permissions. These permissions are represented by pairs with the operator \( a \) in our example, in the place \( \text{Auth} \). Thus, a request \( r \) can only be handed to a given server \( s \) if the term \( a(r,s) \) belongs to the place \( \text{Auth} \). In Figure 2, only one permission is present: the request \( r_0 \) can be handled to the server \( s_1 \). The objective of our hypothetical protocol would be that, at any given time, there is at most one way to connect a server with a client.

First, we could count how many combinations of a request \( r \) and a server \( s \) can be considered by the transition \( \text{connect} \). For this, we can use the expression:

\[
\text{card}(r \in \text{Requests}, \, s \in \text{Idle})
\]

With this expression, we compute the cardinality of the Cartesian product of the contents of places \( \text{Requests} \) and \( \text{Idle} \). On the marking shown in Figure 2, the result of this expression would be 15:
6.2 A more complex example

Figure 2: A clients/servers example

- \( 2 \times \langle r = r_0, s = s_0 \rangle \)
- \( 2 \times \langle r = r_0, s = s_1 \rangle \)
- \( 2 \times \langle r = r_0, s = s_2 \rangle \)
- \( 3 \times \langle r = r_1, s = s_0 \rangle \)
- \( 3 \times \langle r = r_1, s = s_1 \rangle \)
- \( 3 \times \langle r = r_1, s = s_2 \rangle \)

Knowing all the combinations of requests and servers may be useful, but here we would want now to check if our protocol has been correctly modeled. For this, we must count all the combinations that enable the transition `connect`. The following named expression (see Section 4) checks if a combination of a request \( r \) and a server \( s \) that satisfies the guard of the condition:

\[
\text{enable} : \exists (a \in \text{Auth} : \text{getRequest}(a) = r \& \text{getServer}(a) = s)
\]

This expression contains free variables: \( r \) and \( s \), which denote a request and a server. Thus, it cannot be used to perform a check without giving some context: an outer expression where the variables would be bound. The objective of the check is to count the pairs of requests and servers that enable the transition, i.e., those where the expression `enable` returns `TRUE`. We want the number of such pairs to be at most 1.
The resulting expression is:

\[
\text{check} : \text{card}(\langle r \text{ in } \text{Requests}, s \text{ in } \text{Idle} : \@enable \rangle) =< 1
\]

Note that the expression check gives a scope for variables \( r \) and \( s \), used in the expression \( @enable \). The computation of that expression would instantiate all the pairs \( \langle \text{request}, \text{server} \rangle \) from the places Requests and Idle, filter those that do not validate the expression \( @enable \), and then count the result. Finally, the entire file containing the properties would look like what is shown in Figure 3.

Figure 3: A property on the clients/servers example

This figure shows the AlPiNA editor for the properties. This editor was created with XText [Ecl10]. XText is an Eclipse tool that allows the definition of Domain Specific Languages (DSLs). It provides nice features like autocompletion, on-the-fly syntax checking and tight integration with the other Eclipse metamodeling tools. XText is heavily used in all AlPiNA’s textual languages.
7 Concrete syntax

We give here the complete syntax of the properties declaration language. We use a simplified XText syntax for this; i.e., a modified version of EBNF. In this syntax:

- An optional component of a rule is denoted with a question mark;
- A component that can be repeated any number of times (including none) is denoted with a *
- A component that can be repeated any number of times, but at least once, is denoted with a +
- a reference to another object is written using square brackets. For example, the component [Variable] of the rule SortIterator denotes a reference to a variable that has been declared elsewhere. In the case of the variables, this declaration must be done at the end of the file. In the case of places and sorts, the declaration must have been made in the imported files.

Please note that this listing is a simplified version of the actual language definition in the tool. We removed a lot of syntactical components that would not be useful here.

PropertiesDeclaration:
   (Import)*
   ("Expressions" (BoolExpressionWrapper ";")*)?
   "Check" BoolExpression ";"
   ("Variables" (Variable ";")*)?

Import:
   "import" STRING ;

NamedBoolExpression:
   ID ":=" BoolExpression;

BoolExpression:
   ComposedBoolExpression | CompareNatExpression | EqualsExpression;

ComposedBoolExpression:
   TerminalBoolExpression
   (BooleanOperator TerminalBoolExpression)?

BooleanOperator:
   "&" | "|" | "=>";

TerminalBoolExpression:
   BooleanConstant | DeadlockExpression | NegativeExpression | BoolExpressionRef | ForAllExpression | ExistsExpression | "(" BoolExpression ")" ;

BooleanConstant:
   "TRUE" | "FALSE";
DeadlockExpression:
  "Deadlock";

NegativeExpression:
  "!" TerminalBoolExpression;

BoolExpressionRef:
  "@" [NamedBoolExpression];

ForAllExpression:
  "forall"
  "(" Iterator ("," Iterator)* ":" BoolExpression "")";

ExistsExpression:
  "exists"
  "(" Iterator ("," Iterator)* (":" BoolExpression)? ")";

Iterator:
  PlaceIterator | SortIterator;

SortIterator:
  [Variable] "of" [Sort];

PlaceIterator:
  [Variable] "in" [Place];

CompareNatExpression:
  NatExpression CompareNatOperator NatExpression;

CompareNatOperator:
  ">" | ">=" | "<<" | "=<";

NatExpression:
  PlusMinusExpression | MinusExpression;

PlusMinusExpression:
  TimesDivExpression
  (PlusMinusOperator TimesDivExpression)?;

PlusMinusOperator:
  "+" | "-";

TimesDivExpression:
  TerminalNatExpression
  (TimesDivOperator TerminalNatExpression)?;

TimesDivOperator:
  "*" | "/";

TerminalNatExpression:
  INT | CardExpression | "(" NatExpression ")";
CompleteCardExpression | SimpleCardExpression;

CompleteCardExpression:
    "card" "{" Iterator (""," Iterator)* (":" BoolExpression)? "}";

SimpleCardExpression:
    "card" "{" SimpleIterator"}"

SimpleIterator:
    "{" [Sort] "}" | [Place];

MinusExpression:
    "-" TerminalNatExpression;

//Here NotEqualsExpression denotes any expression
//that is not an EqualsExpression
EqualsExpression:
    NotEqualsExpression EqualsOperator NotEqualsExpression;

EqualsOperator:
    "=" | "!=";

NotEqualsExpression:
    NatExpression | Term | NotEqualsBoolExpression;

NotEqualsBoolExpression:
    ComposedBoolExpression | CompareNatExpression;

Variable:
    ID ".:" [Sort];
References


Figure 4: The ADTs used by the model of Figure 1
SMV Technical Reports

This report is in the series of SMV technical reports. The series editor is Didier Buchs (Didier.Buchs@unige.ch).

Any views or opinions contained in this report are solely those of the author, and do not necessarily represent those of SMV group, unless specifically stated and the sender is authorized to do so.

You may order copies of the SMV technical reports from the corresponding author or the series editor. Most of the reports can also be found on the web pages of the SMV group (http://smv.unige.ch/).
Abstract

Model Checking consists in verifying if a model of a given system meets a set of requirements. The model and the properties can be specified with various formalisms. We chose Algebraic Petri Nets (APNs), a powerful formalism used to model concurrent systems, for the model specification. This report proposes a language for defining invariant properties on Algebraic Petri Nets (APNs). The expressiveness of the language proposed is similar to first order logic, with some extensions (deadlocks and cardinalities). These properties can be used to define the requirements for model checking. This language was created for the tool AlPiNA, an APN model checker developed at the University of Geneva.

Keywords: Model checking, Algebraic Petri Nets, Properties, Specification language, first-order logic, AlPiNA.