Modeling and Engineering Software Systems using Petri Networks

by

Michaël Visée
Master of Science, Computer Science and Management
Faculté Polytechnique de Mons, Belgium

Submitted to the System Design and Management Program
in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Engineering and Management
at the
Massachusetts Institute of Technology
May 2007
[June 2007]

Signature of Author_____
Michaël Visée
System Design and Management Program
May 2007

Certified by_____
David Simchi-Levi
Thesis Supervisor
Engineering Systems Division

Certified by_____
Patrick Hale
Director
System Design and Management Program

This page is left intentionally blank.

Modeling and Engineering Software Systems using Petri Networks

by

Michaël Visée

Master of Science, Computer Science and Management
Faculté Polytechnique de Mons, Belgium

Submitted to the System Design and Management Program
on May 11, 2007 in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Engineering and Management.

ABSTRACT

A model of software applications for business process management based on colored Petri Networks is proposed and the corresponding application development process is exposed. A language is proposed to specify the enabling rules of the transitions. An algorithm to solve the binding problem is proposed and detailed. These elements allow the developers to isolate themselves from the very complex details of business process orchestration, transaction management, multi-threading issues, and to concentrate on the implementation of the transitions themselves.

As a proof of concept, a lightweight business process engine based on that model has been implemented as well as the associated development and code generation tools.

Thesis supervisor: David Simchi-Levi
Title: Professor of Engineering Systems

BIOGRAPHICAL NOTE

Michael Visée is an engineer in computer science and management. He has worked in the department of mathematics and operation research at the Polytechnic Faculty of Mons, Belgium where he developed optimization software for garbage collection routes.

He later joined ContentEurope.com and later Cedron.com as software architect where he developed content management systems. He joined Relaystar.com where he participated to the development of some supply chain management software. He finally joined the SDM program at MIT in January 2006.

In parallel, he works for his family owned business in the medical services sector where he manages IT and accounting since 1987 and has directed the construction of a new medical facility in 1999.

Objectives

Business process management is increasingly at the core of any business for the following reasons:

- The economic activities are becoming more and more complex as new products and services are offered on a more global market.
- Increasing competition and global markets drive the need for an ever faster adaptation of processes to changing market conditions.
- Even when applied properly, classical software applications and classical software development approaches are not necessarily flexible and rapid enough to respond to these challenges.

The main objective of the present work is to provide the core implementation of a lightweight business process management engine for small and medium size enterprises. We have excluded big enterprises from this work because they use large-scale systems like ERP and CRM systems that already contain workflow management engines. These systems are usually not affordable for small and medium scale enterprises.

We will use high level Petri networks to model business applications. The structure and behavior of an application can be specified as a high-level Petri net. The resulting application will be a set of communicating high-level nets.

We will develop a code generator that generates the application code from a description of the Petri Nets model in order to accelerate application development and improve the quality and reliability of the resulting applications.

Part I Petri Networks theory - Literature Review

I.1 General Introduction

Petri nets were introduced by Carl Adam Petri in his Ph.D. thesis in 1962 [12]. Today, Petri networks designate a class of network models for concurrent systems. They provide a rigorous and scientific basis for the synthesis [1, 2, 5, 13], analysis, verification [5], validation [3] and simulation of such systems. Petri networks come today essentially in three different forms [14, 15]:

1. Low-level Petri networks. The basic model here is that of elementary net systems. They are well suited for the investigation of properties of concurrent systems and as a basis for other models but not for practical applications because the size of the models explodes even for simple applications.
2. Place/Transition networks: The main model here is that of place transition systems. They fold repetitive features of elementary net systems in order to obtain more compact models.
3. High-level Petri networks: The main models here are those of predicate transition nets and colored Petri nets. Those models use algebraic and logical tools to yield models that are very compact and practical to use in real applications.

The structure of a Petri network is given by a bipartite directed graph. There are two types of nodes called places (or states) and transitions. Directed links can only connect nodes of different types. Transitions represent tasks or activities that have to be done in the process, and places represent identifiable steps or milestones in the process. A directed link from a place to a transition represents a precondition to be met or an input that is necessary for the execution of the transition. A link from a transition to a place represents the production of some result or output by that transition when it occurs.

The global dynamic state of the system is represented by the presence of tokens in the places. The presence of tokens in the input places of a transition determines if that transition is enabled, i.e. if it can occur. The effect of the occurrence of the transition is to consume some tokens in its input places and to produce new tokens in its output places. The three flavors of Petri networks cited previously essentially differ by the type of their tokens and by their enabling and occurrence rules.

1. In low-level Petri networks, there can be at most one token in each place. In fact, places in these networks can be considered as Boolean variables. A transition is enabled if all the input places of the transition have a token and no output place has a token. When the transition occurs, tokens are removed from the input places and added to the output places of the transition. In this case, the global state of the network can be represented as an array of Boolean variables or as a Boolean function.
2. In place-transition networks, places can contain any number of tokens and tokens are indistinguishable. A transition is enabled if all the input places of the transition contain enough tokens. When the transition occurs, tokens are removed from the input places and added to the output places of the transition. The numbers of tokens involved in the enabling rules and in the occurrence rules are specified as a function of the arcs of the network. In this case, the global state of the network can be represented as an array of integer variables or as an integer function.
3. In colored Petri networks, places can contain any number of tokens and tokens are distinguishable. They carry some information with them that is used in the enabling rules and in the occurrence rules and that is accessible when a transition occurs. Each token has a type called its color. The information carried by the tokens represents the parameters and the results of the execution of a transition. In this case, the global state of the network is represented by an array of multi-sets of tokens.

Petri Networks can be also used in combination with queuing theory [10, 11] in order to obtain stochastic process models for the simulation of process systems [6].

The rest of this section presents a summary of the essential definitions and properties of high level Petri networks that we will use in the following parts of this work. A very detailed treatment of theses results can be found in the Kurt Jensen's books and papers [7, 8, 9].

I.2 Definitions

Places or **states** are nodes of the network that can contain tokens. They store the dynamic state of the system. They are usually represented graphically as ellipses. **Transitions** are the nodes of the network that represent actions or behaviors that can occur in the system. They are usually represented graphically as rectangles. **Arcs** represent pre-conditions and post-conditions of transitions.

A node x is called an **input node** of another node y if and only if there exist a directed arc from x to y. Analogously, we can define **output nodes, input places, output places, input transitions, output transitions, input arcs and output arcs**.

Each place in a network may contain a dynamically variable number of elements called **tokens**. An arbitrary distribution of tokens on the places is called a **marking**. The initial distribution is called the **initial marking** and is usually noted M_0. The initial marking is determined by evaluating **initialization expressions**.

A transition is said to be **enabled** in a given marking if, in this marking, there are enough tokens of the right type in all the input nodes of that transition. If it is not the case, the transition is said to be **disabled**.

When a transition is enabled, if may take place. When it happens, we say that the transition occurs. The effect of the occurrence is that tokens are removed from the input places and added to the output places. The number and type of removed and added tokens is specified by the arc expressions of the corresponding arcs.

A marking M_1 is **directly reachable** from marking M_0 if there is a transition T_1 that leads from M_0 to M_1. A marking M_2 is **reachable** from M_0 if there is a sequence of transitions that lead from M_0 to M_2.

In a given marking M_0, it is possible that several transitions are enabled simultaneously and that there are so many tokens that they can operate on disjoints sets of tokens. These transitions are said to be **concurrently enabled** in M_0. This means that these transitions can occur at the same time or in parallel. We call a **step** a multi-set of concurrently enabled transitions.

I.3 Multi-sets

A multi-set m or bag over a non-empty set S is a function from S to N. The integer m(s) is the number of appearances of the element s in the multi-set m and is called the coefficient of s. The multi-set m is usually represented by a formal sum: $m = \sum_{s \in S} m(s)`s$

The set of all multi-sets over S is denoted S_{MS}. The non-negative integers $\{m(s) | s \in S\}$ are the coefficients of the multi-set m. An element $s \in S$ is said to belong to the multi-set m if $m(s) \neq 0$ and we write $s \in m$.

Usual operations are defined on multi-sets in the following way, $\forall m, m_1, m_2 \in S_{MS}$ and $\forall n \in N$.

- Addition: $m_1 + m_2 = \sum_{s \in S} (m_1(s) + m_2(s))`s$
- Scalar multiplication: $n * m = \sum_{s \in S} (n * m(s))`s$
- Comparison:

$m_1 \neq m_2 \Leftrightarrow \exists s \in S : m_1(s) \neq m_2(s)$

$m_1 \leq m_2 \Leftrightarrow \forall s \in S : m_1(s) \leq m_2(s)$

- Size: $|m| = \sum_{s \in S} m(s)$

When $|m| = \infty$, m is infinite. Otherwise m is finite.

- When $m_1 \leq m_2$, subtraction is defined by $m_2 - m_1 = \sum_{s \in S} (m_2(s) - m_1(s))`s$

Multi-sets operations have the following classical properties:

$\forall m, m_1, m_2, m_3 \in S_{MS}$ and $\forall n, n_1, n_2 \in N$.

1. $m_1 + m_2 = m_2 + m_1$
2. $m_1 + (m_2 + m_3) = (m_1 + m_2) + m_3$
3. $m + \emptyset = \text{m}$
4. $1 * m = m$
5. $0 * m = \emptyset$
6. $n * (m_1 + m_2) = (n * m_1) + (n * m_2)$
7. $(n_1 + n_2) * m = (n_1 * m) + (n_2 * m)$
8. $n_1 * (n_2 * m) = (n_1 * n_2) * m$
9. $|m_1 + m_2| = |m_1| + |m_2|$
10. $|n * m| = n * |m|$

I.4 Structure of Non-Hierarchical Colored Petri Networks

The formal definition that follows does not depend on the language in which the net expressions are written. It only assumes that the syntax exists and allows to talk unambiguously about the following elements:

- The elements of a type T. The set of all elements in T is denoted by the type name T.
- The type of a variable v, denoted by Type(v).

- The type of an expression expr, denoted by Type(expr).
- The set of variables in an expression expr, denoted by Var(expr).
- A binding of a set of variables, V associating each variable $v \in V$ an element $b(v) \in Type(v)$.
- The value of an expression expr, in a binding b denoted by expr<b>. Var(expr) must be a subset of the variables of b and the evaluation is obtained by substituting for each variable $v \in$ Var(expr) the value $b(v) \in Type(v)$

By definition, a non-hierarchical CP-net is a tuple $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ satisfying the following conditions:

1. Σ is a set of non-empty types called color sets. It indicates the types, operations and functions that can be used in the net inscriptions.
2. P is a finite set of places.
3. T is a finite set of transitions. We will use $X = P \bigcup T$ to describe the set of all nodes.
4. A is a finite set of arcs such that: $P \bigcap T = P \bigcap A = T \bigcap A = \emptyset$.
5. N is a node function. It is defined from A into $P \times T \bigcup T \times P$. It maps each arc to a pair where the first element is the source node and the second is the destination node. The two nodes must be of different kind. When multiple arcs are present between the same nodes, they will be combined into a single arc by adding their arc expressions. It is always possible because they share the same multi-set type.
6. C is a color function. It is defined from P into Σ. It maps each place p to a color set C(p). Each token on p must have a token color that belongs to C(p).
7. G is a guard function. It is defined from T into expressions such that: $\forall t \in T : Type(G(t)) = B \wedge Type(Var(G(t))) \subseteq \Sigma$. The guard function maps each transition t to a Boolean expression or predicate. We allow the guard expression to be missing and we consider this to be a shorthand for the closed expression true.
8. E is an arc expression function. It is defined from A into expressions such that: $\forall a \in A : Type(E(a)) = C(p(a))_{MS} \wedge Type(Var(E(a))) \subseteq \Sigma$ where p(a) is the place of N(a). The arc expression function maps each arc a to an expression of type $C(p(a))_{MS}$, which means that the evaluation of the arc expression yields a multi-set over the color set that is associated to the corresponding place.

9. I is an initialization function. It is defined from P into closed expressions such that: $\forall p \in P : Type(I(p)) = C(p)_{MS}$. It maps each place p into a closed expression of type C(p)MS, a multi-set over C(p). It describes the initial marking of the network. We allow the initialization function to be missing and we consider this to be a shorthand for the empty multi-set.

We also define the functions below describing the relationship between neighboring elements of the net structure. Each function name indicates the range of the function.

- $p \in [A \rightarrow P]$ maps each arc a to the place of N(a), i.e. the component of N(a) which is a place.
- $t \in [A \rightarrow T]$ maps each arc a to the transition of N(a), i.e. the component of N(a) which is a transition.
- $s \in [A \rightarrow X]$ maps each arc a to the source of a, i.e. the first component of N(a).
- $d \in [A \rightarrow X]$ maps each arc a to the destination of a, i.e. the second component of N(a).
- $A \in [(P \times T \cup T \times P) \rightarrow A_S]$ maps each ordered pair of nodes (x_1,x_2) to the set of its connecting arcs, i.e. the set of arcs that have the first node as source and the second node as destination. $A(x_1, x_2) = \{a \in A : N(a) = (x_1, x_2)\}$.
- $A \in [X \rightarrow A_S]$ maps each node x to the set of its surrounding arcs, i.e. the set of arcs that have x as source or destinations.
  $A(x) = \{a \in A : \exists x' \in X : [N(a) = (x, x') \vee N(a) = (x', x)\}$
- $In \in [X \rightarrow X_S]$ maps each node x to the set of its input nodes, i.e. the set of nodes that are connected to x by an input arc.
  $In(x) = \{x' \in X : \exists a \in A : N(a) = (x', x)\}$ It is also denoted $\bullet x$
- $Out \in [X \rightarrow X_S]$ maps each node x to the set of its output nodes, i.e. the set of nodes that are connected to x by an output arc.
  $Out(x) = \{x' \in X : \exists a \in A : N(a) = (x, x')\}$ It is also denoted $x \bullet$
- $X \in [X \rightarrow X_S]$ maps each node x to the set of its surrounding nodes, i.e. the set of nodes that are connected to x by an arc.
  $X(x) = \{x' \in X : \exists a \in A : [N(a) = (x, x') \vee N(a) = (x', x)]\}$

All these functions can be extended to take sets as input and return sets as result.

We also define:

- $\forall t \in T : Var(t) = \{v : v \in Var(G(t)) \vee \exists a \in A(t) : v \in Var(E(a))\}$. Var(t) is called the set of variables of transition t.
- $\forall x_1, x_2 \in (P \times T \cup T \times P) : E(x_1, x_2) = \sum_{a \in A(x_1, x_2)} E(a)$. $E(x_1,x_2)$ is called the expression of (x_1,x_2). The summation indicates addition of arc expressions and is well defined because the expressions have a common multi-set type.

I.5 Behavior of Non-Hierarchical Colored Petri Networks

A **binding** of a transition t is a function b defined on Var(t) such that:

1. $\forall v \in Var(t) : b(v) \in Type(v)$
2. G(t)<b>

B(t) denotes the set of all bindings for transition t.

A **token element** is a pair (p,c) where $p \in P$ and $c \in C(p)$. The set of all token elements is denoted by TE.

A **binding element** is a pair (t,b) where $t \in T$ and $b \in B(t)$. The set of all binding elements is denoted by BE.

A **marking** is a multi-set over TE. The initial marking M_0 is obtained by evaluating the initialization expressions. $\forall (p,c) \in TE : M_0(p,c) = (I(p))(c)$. The set of all markings is denoted by M.

A **step** is a non-empty and finite multi-set over BE. The set of all steps is denoted by Y.

A step Y is **enabled** in a marking M if and only if $\forall p \in P : \sum_{(t,b) \in Y} E(p,t)\langle b \rangle \leq M(p)$.

Let the step Y enabled in the marking M.

1. When $(t,b) \in Y$, t is enabled in M for the binding b. We say that (t,b) is enabled in M and that t is enabled in M.
2. When $(t_1,b_1),(t_2,b_2) \in Y$ and $(t_1,b_1) \neq (t_2,b_2)$, (t_1,b_1) and (t_2,b_2) are concurrently enabled.
3. When $|Y(t)| \geq 2$, t is concurrently enabled with itself.
4. When $Y(t,b) \geq 2$, (t,b) is concurrently enabled with itself.

When a step Y is enabled in a marking M_1 it may occur, changing the marking M_1 to another M_2, defined by: $\forall p \in P : M2(p) = (M1(p) - \sum_{(t,b)\in Y} E(p,t)\langle b\rangle) + \sum_{(t,b)\in Y} E(t,p)\langle b\rangle$

The first sum is called the **removed tokens** while the second is the **added tokens**. M_2 is said directly reachable from M_1 by the occurrence of the step Y, which we denote $M_1[Y>M_2$.

A **finite occurrence sequence** is a sequence of markings and steps:

$M_1[Y_1>M_2[Y_2>M_3\ldots M_n[Y_n>M_{n+1}$ such that $n \in N$, and $M_i[Y_i>M_{i+1}$ for all $i \in 1..n$. M_1 is the **start marking** and M_{n+1} is the **end marking**. n is the number of steps or the **length** of the occurrence sequence.

An infinite occurrence sequence is a sequence of markings and steps:

$M_1[Y_1>M_2[Y_2>M_3\ldots$ such that $M_i[Y_i>M_{i+1}$ for all $i \in N_+$.

The set of all finite occurrence sequences is denoted by OSF, the set of all infinite occurrence sequences is denoted by OSI and the set of all occurrence sequences is denoted by $OS = OSF \cup OSI$.

A marking M'' is **reachable** from a marking M' if and only if there exists a finite occurrence sequence with start marking M' and end marking M''. The set of markings reachable from M' is denoted by [M'>. As a shorthand, a marking is said reachable if and only if it is reachable from the initial marking M_0.

I.6 Static Properties of CP-Nets

Let an arc $a \in A$ with arc expression E(a), a transition $t \in T$ and a non-negative integer $n \in N$.

1. E(a) is uniform with multiplicity n if and only if $\forall b \in B(t(a)) : |E(a)\langle b\rangle| = n$
2. t is uniform if and only if all the surrounding arcs of t have a uniform arc expression
3. t is conservative if and only if $\forall b \in B(t) : \sum_{p\in In(t)} |E(p,t)\langle b\rangle| = \sum_{p\in Out(t)} |E(t,p)\langle b\rangle|$
4. T has the state machine property if and only if

   $\forall b \in B(t) : \sum_{p\in In(t)} |E(p,t)\langle b\rangle| = \sum_{p\in Out(t)} |E(t,p)\langle b\rangle| = 1$

I.7 Boundedness Properties

Let a set of token elements $X \subseteq TE$ and a non-negative integer $n \in N$.

1. n is an upper bound for X if and only if $\forall M \in [M_0 >: |(M \mid X)| \leq n$

2. n is an lower bound for X if and only if $\forall M \in [M_0 >: |(M \mid X)| \geq n$

The set X is bounded if and only if it has an upper bound.

Let a place $p \in P$, a multi-set $m \in C(p)_{MS}$ and a non-negative integer $n \in N$.

1. n is an upper integer bound for p if and only if $\forall M \in [M_0 >: |M(p)| \leq n$
2. m is an upper multi-set bound if and only if $\forall M \in [M_0 >: M(p) \leq m$

Lower bounds are defined analogously. A place p is bounded if and only if it has an integer upper bound.

I.8 Home Properties

Let a marking $M \in$ M and a set of markings $X \subseteq$ M

1. M is a home marking if and only if $\forall M' \in [M_0 >: M \in [M' >$
2. X is a home space if and only if $\forall M' \in [M_0 >: X \cap [M' > \neq \emptyset$

We use HOME to denote the set of all home markings.

$\forall M \in \mathrm{M} : M \in HOME \Leftrightarrow [M >= HOME$

I.9 Liveness Properties

Let a marking $M \in$ M and a set of binding elements $X \subseteq BE$

1. M is dead if and only if no binding element is enabled. $\forall x \in BE : \neg M[x >$
2. X is dead in M if and only if no element of X can become enabled. $\forall M' \in [M >, \forall x \in X : \neg M'[x >$
3. X is live if and only if there is no reachable marking in which X is dead. $\forall M' \in [M_0 >, \exists M'' \in [M' > \exists x \in X : M''[x >$

We say that X is dead if and only if X is dead in M_0.

$\forall X, Y \subseteq BE$, we have

1. $X \supseteq Y \Rightarrow (X\ dead \Rightarrow Y\ dead)$
2. $X \subseteq Y \Rightarrow (X\ live \Rightarrow Y\ live)$

Part II Business Process Engine Implementation

II.1 Requirements for the CPN Model Representation

The CPN model representation must fulfill the following main requirements:

1. Ease of use: The model representation must be easy to create and to manipulate and must integrate smoothly with version control tools.
2. Extensibility: This requirement has two parts. First the model should be extensible in the sense that an existing business process model should be easy to extend with new places and new transitions, etc. Second, the model representation must be open to the later addition of new concepts and to the specification of specialized components.
3. Easy to validate: Like a compiler, the associated tools must be able to detect any error or mistake in a process model way before the corresponding application is deployed.
4. The model must represent the following concepts:
   - Colors: They are the token types. They are essentially objects containing fields.
   - Transition Types: They specify the type of transitions, their input and output types.
   - Places.
   - Transitions and arcs.
   - Triggers: They are special places where token may appear based on time.
   - Message senders/receivers: They may produce or consume tokens in relation with a messaging infrastructure.
5. The representation must contain the necessary information to drive code generation and database generation.

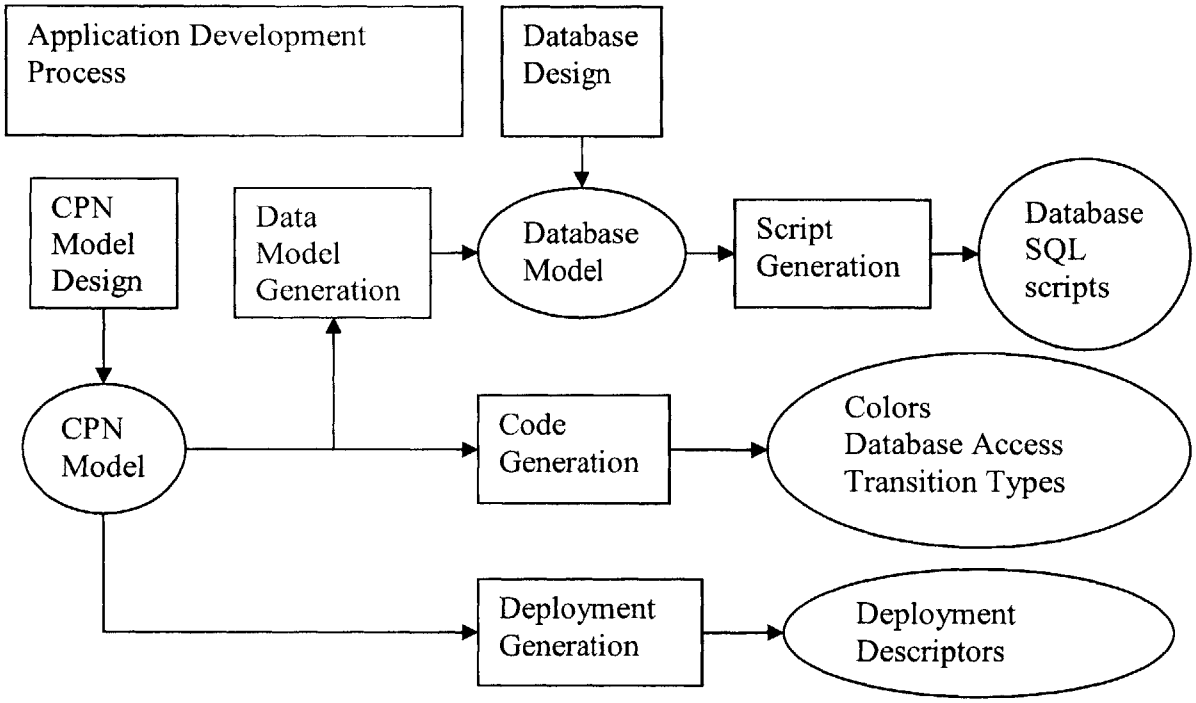
II.2 Proposed Development Process

Based on the CPN model representation described in the previous section, we can propose the following development process for software applications represented on the figure below. Rectangles represent activities and ellipses represent artifacts. The following list describes the essential steps of this process.

1. CPN Model Design: This is the most important step in the process. From the requirements of the application and from his knowledge of the business and external

systems, the designer creates a new business process model. The result of this step is the CPN model.

2. Database design: Although the system will automatically generate the database needed to represent the state of the process model, parts of the database are not related to the process model itself and must be designed manually or using other tools.
3. Data Model Generation: This step generates the data model necessary to represent the state of the process.
4. Script generation: This step generates the SQL scripts necessary to create the system database.
5. Code generation: This step generates several components from their description contained in the CPN model: Color classes, database access classes and transition types.
6. Deployment generation: This step generates the deployment descriptors that will be used by the business process engine to load and manage the necessary components.

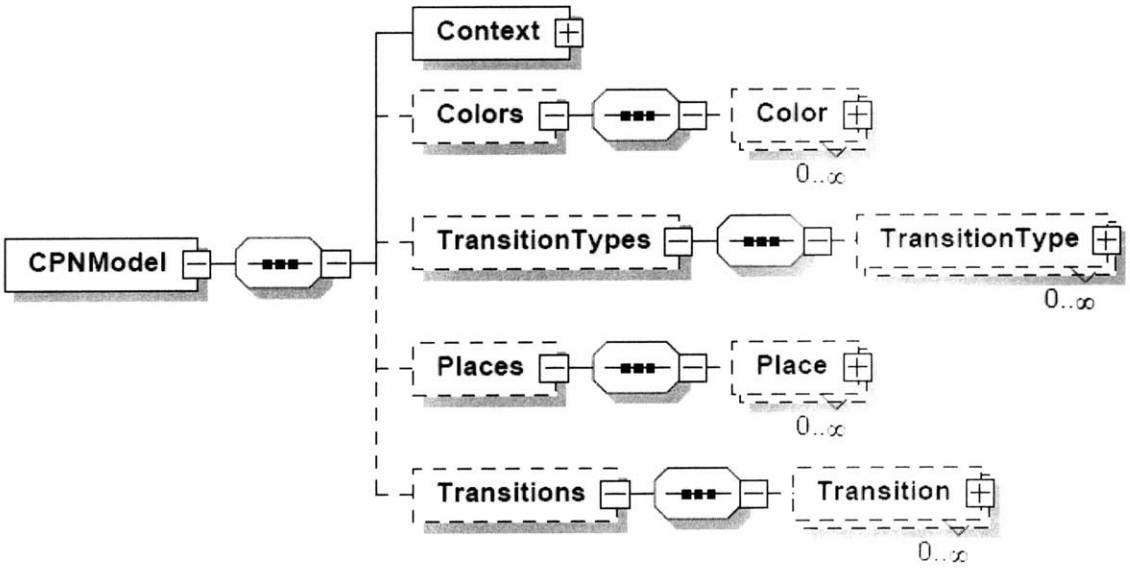


II.3 Detailed CPN Model Representation

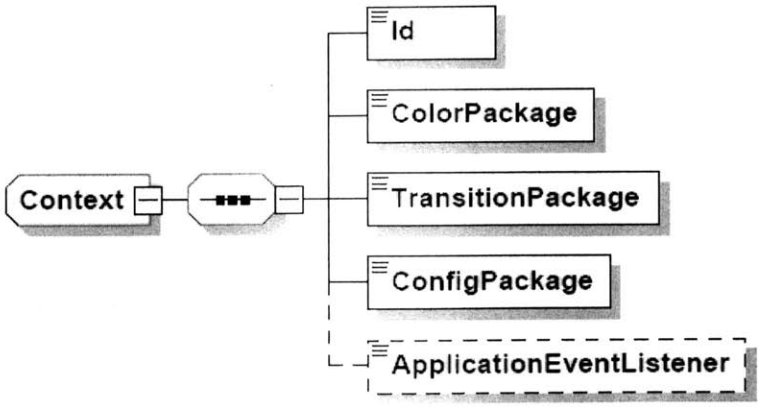
In order to fulfill the requirements specified in section 1 above, I have chosen to represent the CPN model using an XML file. The format of this file is specified using an XML schema. It is a simple solution that has the following characteristics:

1. Ease of use: XML files are text files. They are easy to create and manipulate and version control systems are able to compare their successive versions.
2. Extensibility: As shown in the detailed description below, new elements can easily be added to an existing model. Moreover, XML schemas allow the easy definition of new types and entities in an existing schema.
3. Validation: Generic XML tools allow the validation of an XML file with respect to its XML schema. This means that any constraint that can be expressed as an XML schema constraint can be validated without the need of any specific tool.

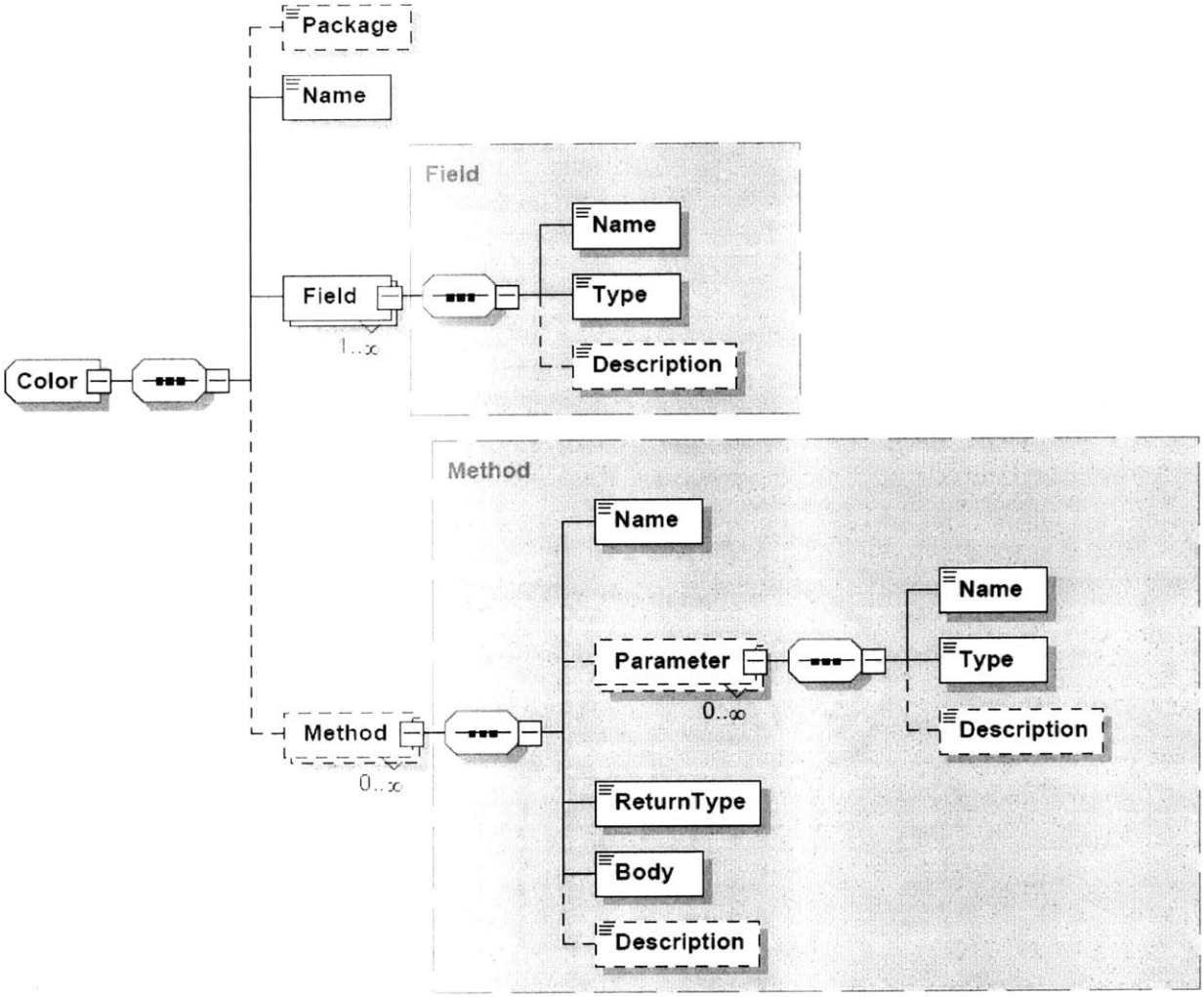
The figures below present the detailed view of the model:



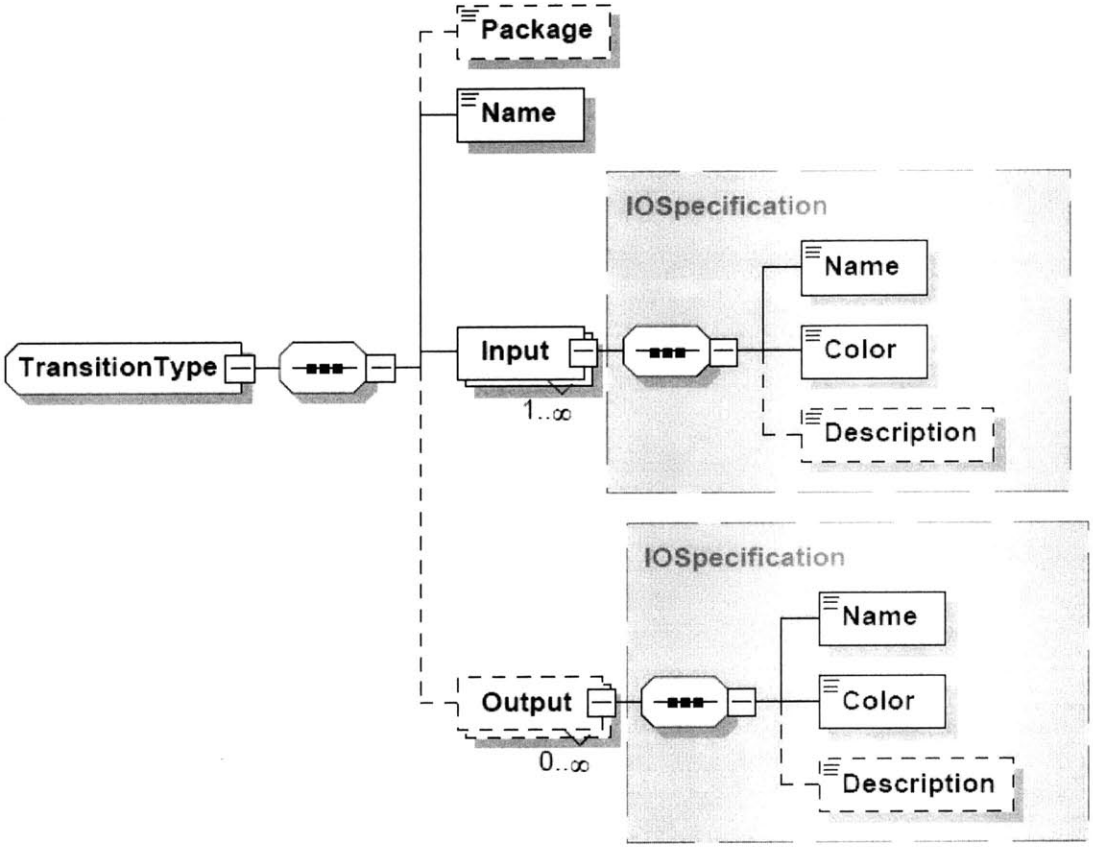
A CPN model contains a context, a list of colors, a list of transition types, a list of places and a list of transitions.



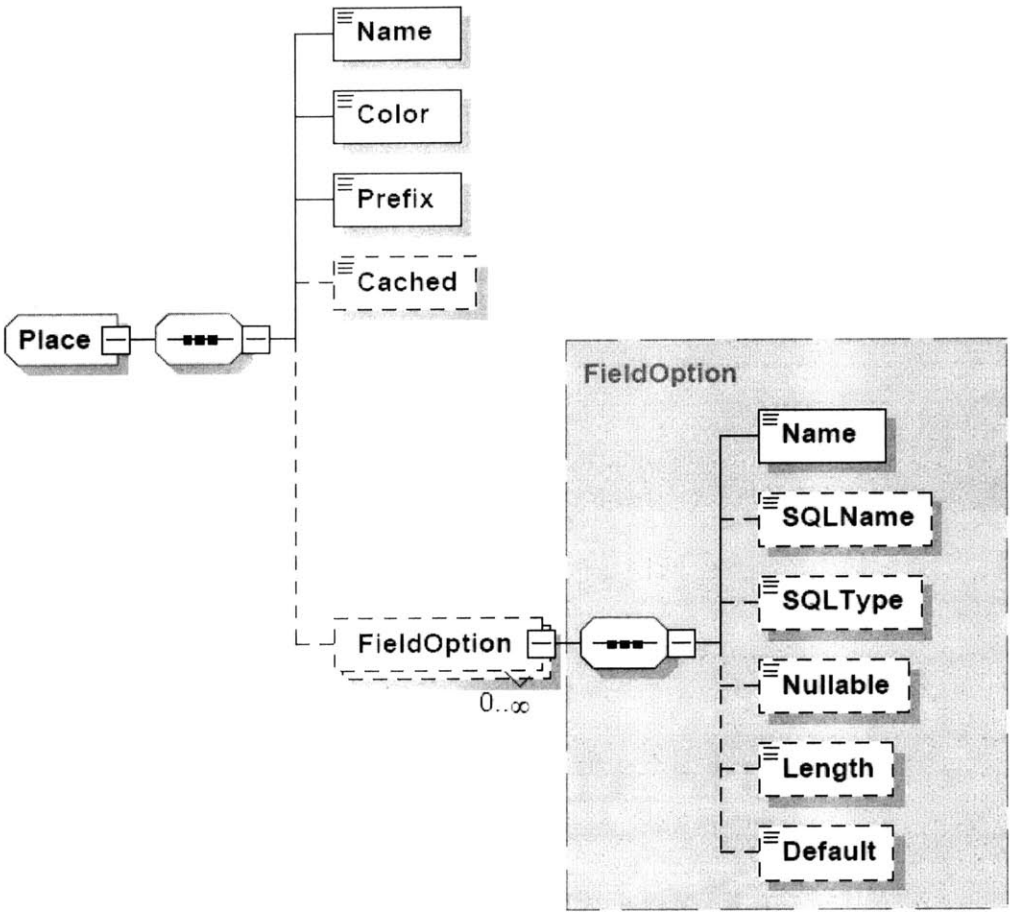
The context contains the identifier of the model, the default package name for the colors, the default package name for the transition types, the default configuration package for the engine runtime and the optional fully qualified class name of the application event listener.



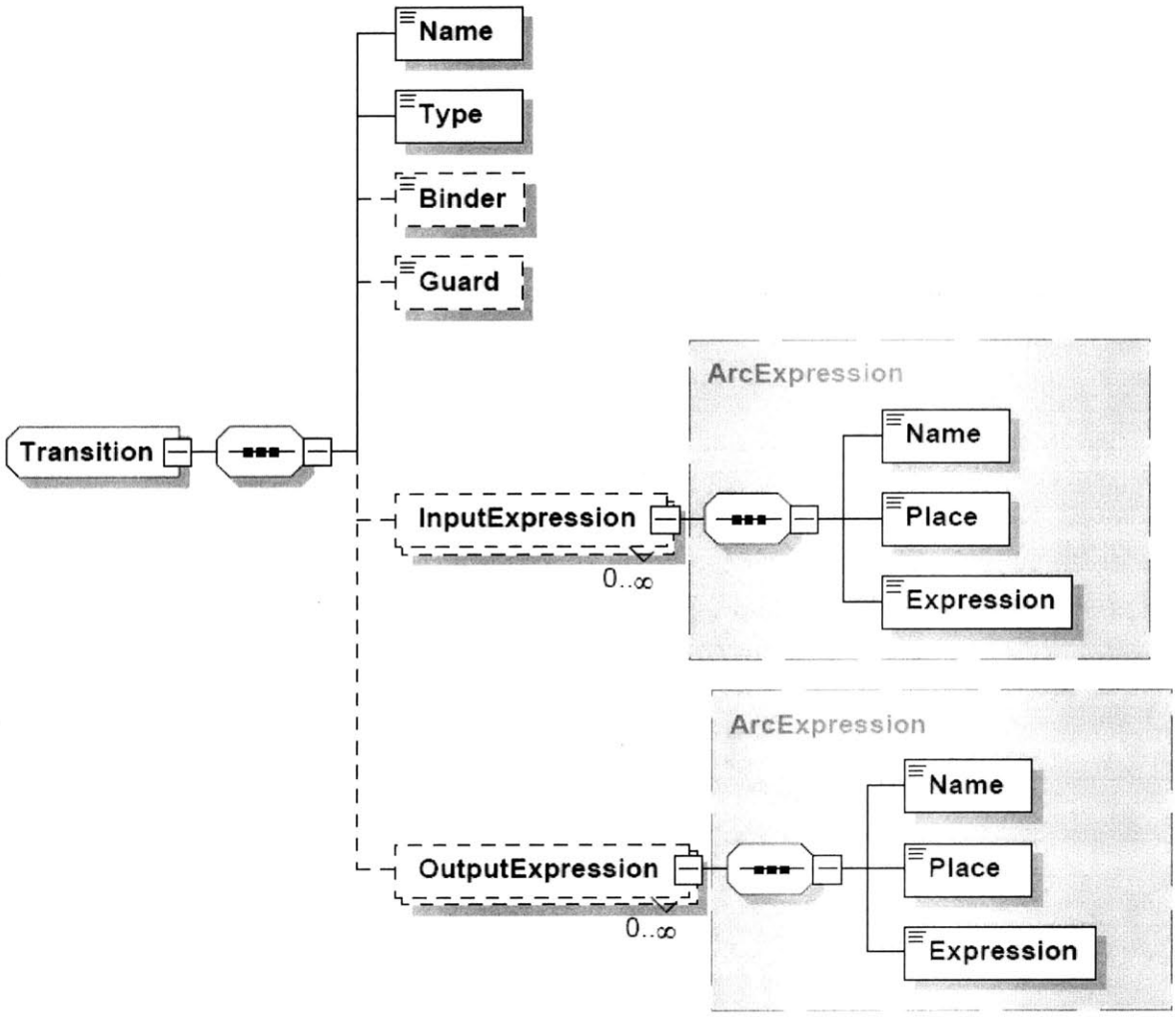
Each color has an optional package name, a class name, a list of fields and a list of methods. Each field has a name, a java type and a description. Each method has a full description with a method name, a list of parameters, a return type, the body of the method and a description.



Each transition type has an optional package name, a class name and a list of inputs and a list of outputs. Each input and output has a name, a color and an optional description.



Each place has a name, a color and a prefix to use in database column names. The tokens of a place may be cached or not by the engine. It also has a list of field options that allows to specify the parameters for database generation for each field. The field options allow to specify if a field is nullable, its length and default value and its SQL name and type if they are not the default.



Transitions have a name, a type, an optional binder, a guard condition and a list of arc expressions for inputs and outputs. Each arc expression applies to the input or output with the given name and connects the transition to the given place.

The core of the problem of the execution or simulation of the process described in the model is to find the bindings that enable the transitions in a given marking. That's the function of a binder class. Given a marking or more precisely given the multi-sets of tokens present in the input places of a transition, the binder class has to find the tokens that make a binding for that transition. The possibility to specify a specific binder class for each transition allows to write any kind of binder as plain java code without the need for the guard condition and the arc expressions.

However, the subject of the next two sections is to define a language for arc expressions and the guard condition in order to provide a default binder implementation that can be used in the majority of cases so that a specific binder does not have to be written for each transition.

II.4 Language Specification for Arc Expressions and Guard Conditions

The purpose of input arc expressions is to specify an expression that evaluates to a multi-set defined over the color of the place connected to the considered input arc. As we will see, the expression will also define variable names that we will use in the guard condition.

II.4.1 Language Grammar

We will use the traditional notation of language grammars. For example, the following production means that an arc expression is an 'if' expression or a 'multi-set' expression.

ArcExpression:

IfExpression

MultisetExpression

IfExpression:

if (Condition) then ArcExpression else ArcExpression

MultiSetExpression:

MultiSetTerm

MultiSetTerm + MultiSetTerm

MultiSetTerm:

IntegerLiteral*Identifier

Condition:

Term

Term || Term

Term:

Factor

Factor && Factor

Factor:

! Factor

RelationalExpression

(Condition)

RelationalExpression:

VariableReference RelationalOperator Literal

VariableReference RelationalOperator VariableReference

Both sides of the relational expression must be of compatible types.

VariableReference:

InputName . VariableName . FieldOrMethodName

InputName is the name of an input of the considered transition.

VariableName is an identifier introduced in a term of the corresponding input expression.

FieldOrMethodName is the name of a field or the name of a method of the color of the considered input.

RelationalOperator:

==

!=

<=

<

<=

>

Literal:

BooleanLiteral

CharacterLiteral

StringLiteral

IntegerLiteral

FloatLiteral

The different literals have their usual definition as constants of the considered type.

II.4.2 Multi-Set Expressions

Multi-set expressions are essentially a sequence of terms separated by + signs. Each term specifies an integer coefficient and an identifier. The corresponding multi-set will be the union of the multi-sets specified by each term.

For each term, the coefficient specifies the cardinality of the multi-set corresponding to this term and the identifier defines the name by which the corresponding tokens will be referred in the

guard condition. As an additional condition, field values or method results of the tokens belonging to a given term must be the same if they are used in any part of the guard condition. This guarantees that the variable references used in the guard condition may refer indifferently to any token of the same term.

For example, suppose we have the multi-set expression 2x+3y and we have the guard condition input1.x.name=="John Doe". It results from the last condition that the two tokens used for the term 2x must have the same value for their name field, in that case "John Doe"

More formally, for each term, this condition defines an equivalence relation on the tokens belonging to the corresponding input. All the tokens used in a given term must be equivalent for the equivalence relation defined for this term.

II.4.3 If Expressions

The main purpose of the if expressions is to define input arc expressions whose cardinality depends on the values of fields of tokens from another input. Based on the evaluation of a condition, the result is the arc expression defined in the then or in the else part of the expression.

An obvious restriction is that the condition specified in such an expression can only depend on variables from the other inputs of the considered transition. We will impose the additional restriction that the conditions used in if expressions depend only on terms defined in multi-set expressions.

II.4.4 Parser Implementation

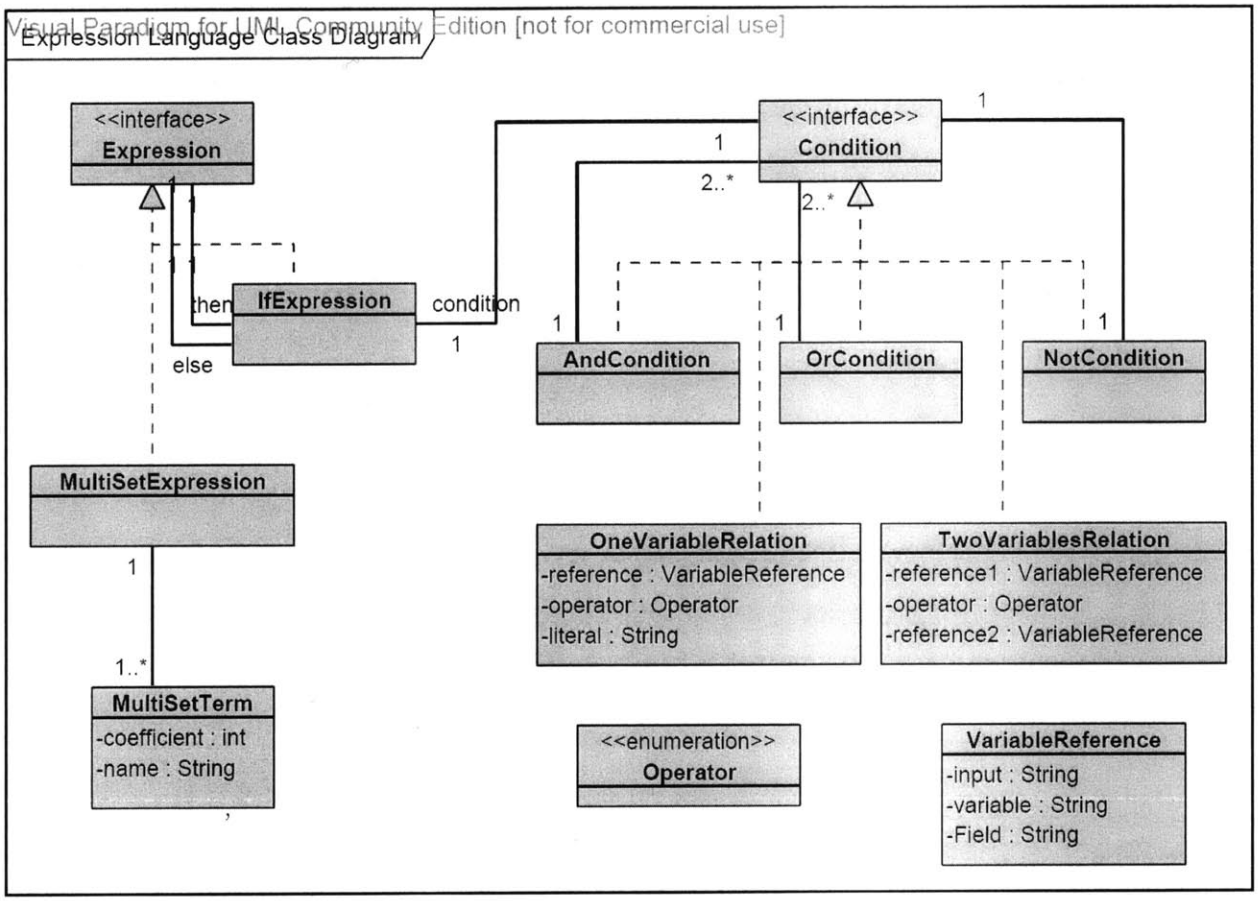
To parse the arc expressions and the guard condition, we use a classical recursive parser implementation that generates an object representation of the expressions. That representation is based on the well-known composite and interpreter design patterns [4] and is shown on the class diagram below.

Classes MultiSetExpression and IfExpression implement a common interface Expression and represent arc expressions. A MultiSetExpression instance has a collection of MultiSetTerm that represent the terms of the multi-set expression. An IfExpression has a Condition, a then expression and an else expression.

Classes AndCondition, OrCondition and NotCondition represent composite conditions based on the corresponding boolean operator (And, Or, Not). Classes OneVariableRelation and TwoVariableRelation are the leaf conditions and represent different types of relations with one

or two variables and one of the six possible comparison operators. All these classes implement the common interface Condition.

We have also defined a visitor interface [4] that will allow the separate implementation of several operations defined for the whole structure (Type checking, expression evaluation, etc).



II.5 Algorithm for the Determination of Bindings

The problem of the determination of bindings from a marking can be formulated as a constraint satisfaction problem. The problem is to choose tokens in the given marking so that the constraints expressed in the arc expressions and in the guard condition are satisfied. So the algorithm we will use to find the bindings is based on constraint programming techniques. It is essentially a depth first search algorithm that tries to choose the tokens to put in each input multi-set. The choices are driven by an incremental computation of the domains of the variables of the problem. The algorithm will use steps like domain reduction, constraint propagation, variable

selection heuristics and value selection heuristics in order to accelerate the convergence to solution binding.

At the highest level, the inputs of the algorithm are:

- The considered transition.
- For each input place of the transition, the multi-set of tokens present in that place in the current marking of the network.
- The arc expressions
- The guard condition

The output of the algorithm must be a list (possibly empty) of enabled bindings.

Each term appearing in the arc expressions will define a number of variables equal to the coefficient of the term. The domains for those variables are a priori the multi-sets of tokens present in the corresponding input place.

In order to avoid a huge memory use, computations on the domains are done incrementally in a single domain representation. Each value in a domain can be marked in three ways: Allowed, initially forbidden or forbidden for the first time at step i. This allows to restore de domains of the previous steps during backtracking.

II.5.1 Fast Checking

Any arc expression has a minimum cardinality. For a multi-set expression, it is the sum of the coefficients of the terms in the expression. For an if expression, it is the minimum value of the minimum cardinalities of the multi-set expressions appearing in the if expression.

Obviously, the cardinalities of the token multi-sets in the given marking must be greater or equal to the minimum cardinalities of the corresponding arc expressions. If not, no binding exists and the problem is solved.

II.5.2 Initial Reduction

During initial reduction, we will build the equivalence classes defined for each term in the arc expressions and compute the cardinalities of these equivalence classes. Classes with a cardinality that is smaller than the corresponding term coefficient can not supply tokens for this term. This allows an initial reduction of the term variables domains. If any of the domains obtained is empty, no binding exists and the problem is solved.

II.5.3 Main Search Loop

The main search loop is a classical depth first search algorithm. Each iteration proceeds as follows:

1. Constraint propagation (See next section). If constraint propagation is successful, go to step 2 else backtrack and repeat step 1 if possible.
2. If all variables are set, build the new solution and add it to the result, reset the search and leave the loop if the initial reduction of the new search detects any empty domain.
3. Choose the next variable and next value to set and repeat step 1

II.5.4 Constraint Propagation

Constraint propagation proceeds in 4 steps:

1. Constraint propagation at the term level: All the variables corresponding to a given term must belong to the same equivalence class in the equivalence relation defined by this term. So, after a variable has been set, all the other free variables belonging to the same term are restricted to the same equivalence class and their values belonging to other classes are marked forbidden at this step. If any of the domains becomes empty, the actual partial solution is infeasible and we must backtrack.
2. Constraint propagation at the input level: All the variables belonging to a given input must be different. So, after a variable is set, its value is forbidden in the domain of all other free variables in the same input. If any of the domains becomes empty, the actual partial solution is infeasible and we must backtrack.
3. Guard condition evaluation: The objective is to find a solution such that the guard condition is true. After each variable is set, we will evaluate the guard condition based on the fixed variables and the reduced domains of the free variables. This evaluation is done using three states logic. Given the partial knowledge of the variables that we have at any given point, the guard condition can take 3 values: true, false or unknown. If the guard condition evaluates to false, the actual partial solution is infeasible and we must backtrack.
4. Creation and initial reduction of if expressions: If the transition has some if input, we use the same three states logic to evaluate the conditions in the if expressions. This allows to create the corresponding multi-set expressions, to create the corresponding variables and

to apply the initial reduction to them. If any of the domains of the new variables is empty, the actual partial solution is infeasible and we must backtrack.

II.5.5 End test, Building of a New Solution and Reset of the Search

The end test simply checks if all the variables have been set. If it is the case, the tokens currently selected make the solution. This solution is extracted from the domains and added to the final result.

After the new solution is built, the search is reset in the following way:

1. The tokens that were part of the solution are forbidden in the domains of all the variables of the input to which they belong.
2. The initial reduction algorithm is reapplied. If any empty domain is found, no more solution exists and the algorithm ends.

II.5.6 Variable and Value Choice Strategies

At each step of the procedure, we must choose which variable to set. We simply choose the one whose domain has the smallest cardinality.

As the problem formulation does not give any clue about which value of a variable could preferably lead to a solution, we simply choose the values in the order they are in the given tokens lists.

II.5.7 Backtracking

Each time a "dead end" choice of the variables has been reached, the algorithm must backtrack to the next unexplored partial solution. Two additional tasks are done during backtracking:

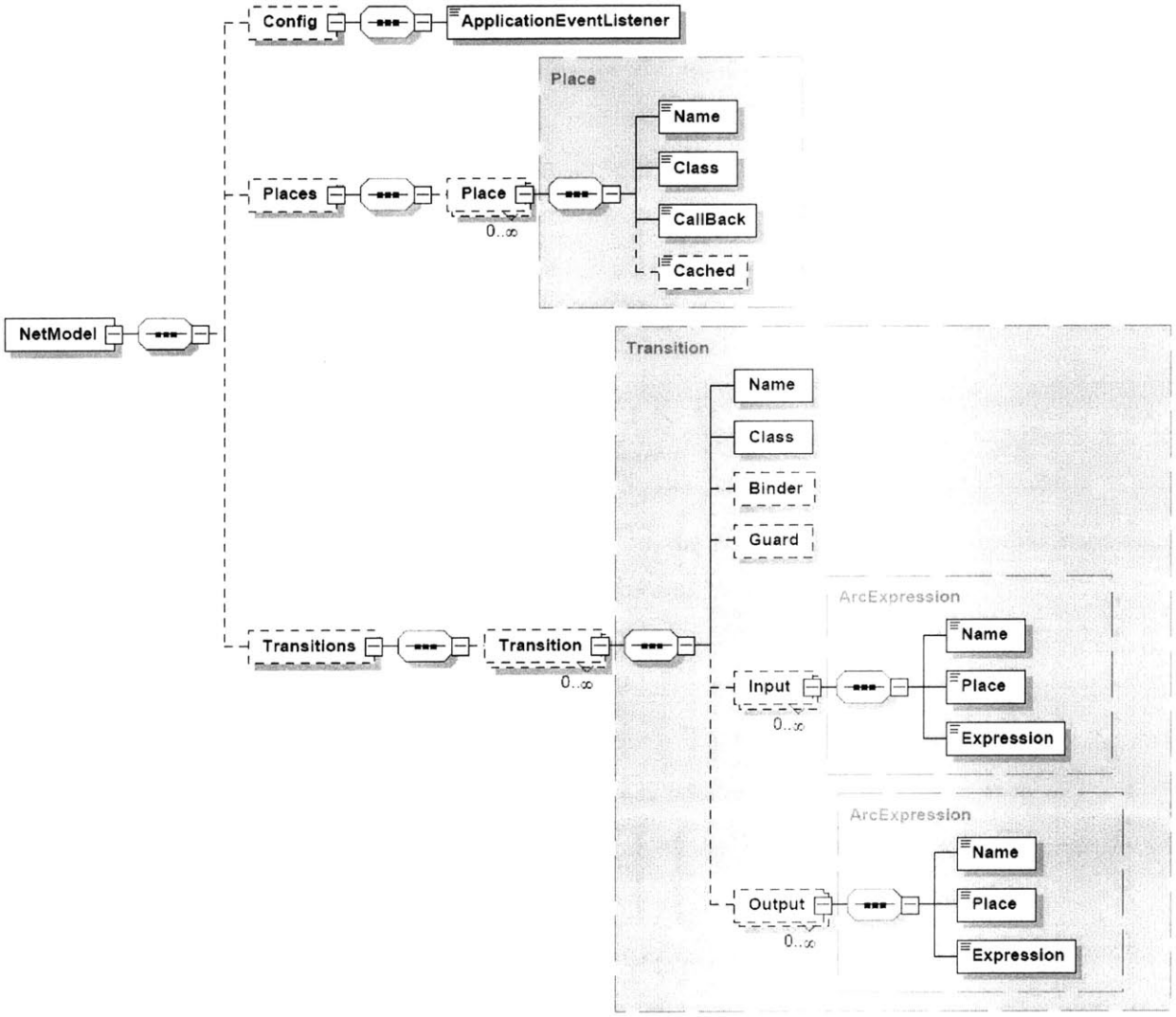
1. The domains of the variables are restored to their previous state by allowing the values that were forbidden during the steps that are backtracked.
2. The variables of if expressions that were created during the steps that are backtracked are destroyed.

When no unexplored partial solution remains, the algorithm ends.

II.6 Business Process Engine Main Algorithm

II.6.1 Detailed Engine Configuration or Deployment Descriptor

The detailed engine configuration is also represented by a NetModel XML file that will usually be stored in the configuration package. This file is essentially identical to the CPN model but with all the design time information removed. Its schema is shown is the figure below.



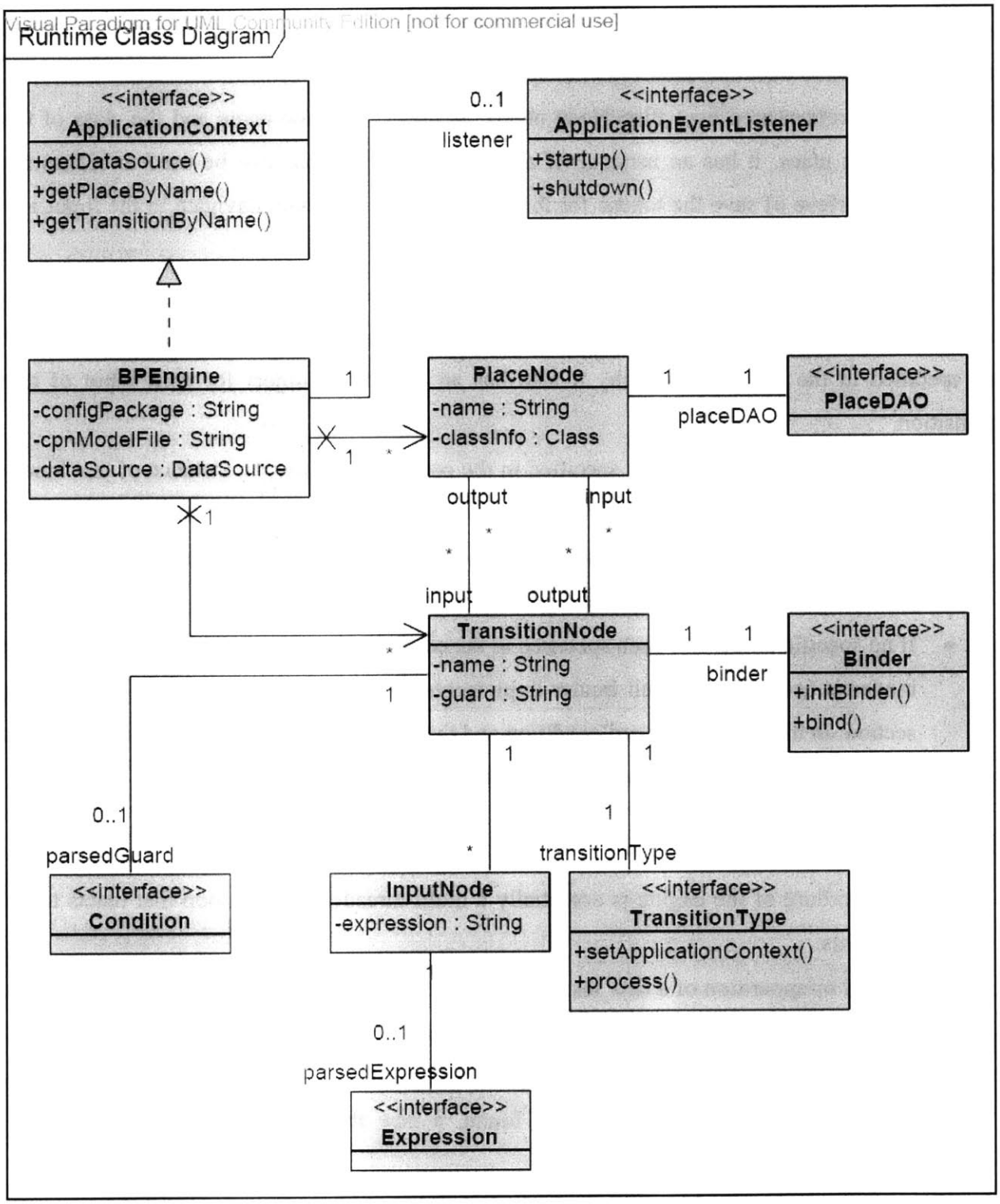
The config element specifies the class name of the application event listener

The place elements specify de configuration of places. Each place has a name, a class name for the tokens, a callback class name for access to the database and may be cached or not.

The transition elements specify the configuration of transitions. Each transition has a name, a transition type class, a binder class, a guard condition and arc expressions.

II.6.2 Startup

The startup procedure essentially loads the configuration file and creates the object model as specified in the class diagram shown below.



The engine is represented by an instance of the BPEngine class. The configuration package name, the configuration file name and the datasource must be injected at creation time. The engine may have an ApplicationEventListener instance as specified in the configuration file. This listener and its startup method should be used to execute the additional configuration of transitions when they need access to external resources or to special configuration information.

Each place is represented by a PlaceNode object. It stores the place name and the class of the tokens for that place. It has an instance of a PlaceDAO object that will be used to access the database to retrieve of save the tokens for that place. It also allows to navigate to the input and output transitions of that place.

Each transition is represented by a TransitionNode object. It stores the transition name and guard condition of that transition. It has an instance of a Binder object and of a TransitionType object as specified in the configuration file. It also has an InputNode object for each input of that transition.

- If a specific binder has been specified in the configuration file, an instance of that binder is created. The guard condition and the arc expressions are stored respectively in the TransitionNode object and in the InputNode objects but are not parsed. (The specific binder is not supposed to use the expression language described above)
- If no specific binder has been specified in the configuration file, an instance of the default binder is used. That default binder implements the algorithm described in the previous section. In that case, the guard condition and the arc expressions are stored respectively in the TransitionNode object and in the InputNode objects and are parsed.

II.6.3 Main Procedure

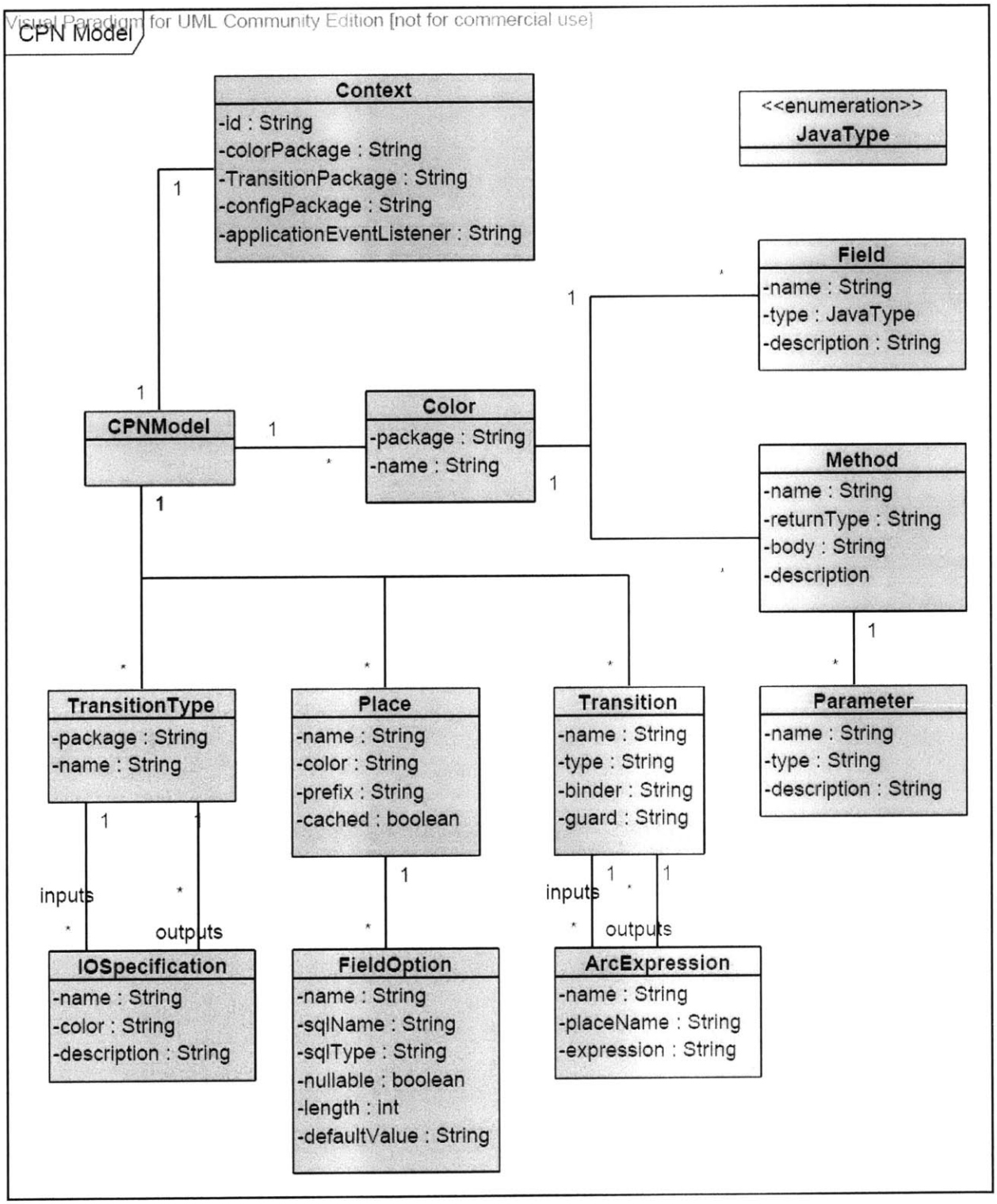
The main procedure of the engine is essentially a multi-threaded infinite loop that reacts to the following events:

- Arrival or apparition of a new token: when a new token is produced either by a transition execution of by an external entity like a message receiver, the token will be stored in the database in the corresponding place and all the transitions that have that place as input are checked. For each enabled binding found, a new thread is created that execute the transition for the considered binding.
- Reset request: All the transitions are checked for enabled bindings.

- Shutdown of the engine: The current executing transitions are continued until they are completed but no additional transition is started.

II.7 Implementation of the Different Generators.

II.7.1 CPN Model Object Representation



Our XML parser creates the object representation presented in the class diagram above from the XML file. This object model contains all the necessary information to realize the different code and script generations.

1. Color generation: Each color is described in detail in the model with its package, name, list of fields and list of methods.
2. Database model generation: The state of the system is stored in a relational database that has one table for each place in the network. Knowing the description of places and their colors, it is easy to generate the database model as described in the next section.
3. Callback classes' generation: The callback classes essentially contain the SQL commands that allow to insert, read, update, and delete tokens. They can also easily be generated from the model above.
4. Transition types' generation: The system generates only the skeleton of transition types.
5. Net Model generation: As we have seen previously, the net model is essentially the CPN model with all the design time information discarded.

II.7.2 Database Model

A database model is also described in an XML file that is specified by an XML schema. Each model contains sequences, tables and triggers. This allows us to describe a database model independently of the specific database implementation. This has the following advantages:

- The first step that generates the database model from the CPN model is independent of the database implementation.
- We can add manually generated tables to the model generated from the CPN model.
- We can generate the SQL scripts for several database implementations based on the same database model. In our case, we have implemented script generation for ORACLE10

III Conclusion

III.1 General Conclusion

The present work has shown the feasibility of the rapid implementation of business process management applications by using a development approach based on the Petri Network model and on several code generation tools. In particular:

- Developers specify the inputs and outputs of their transitions and their enabling rules based on the requirements of their application using a very simple language. The engine implementation takes care of the interpretation and enforcement of these rules.
- Developers don't have to write any database access code to persist the state of the process. The only database work they have to do concerns the additional information that is specific to their application.
- Developers don't have to care about multithreading issues or transaction management issues. These issues are solved by the engine itself.
- The system doesn't use anything else than plain old java code and standard libraries so it can be integrated in a wide variety of environments from standalone java applications to sophisticated application servers.

III.2 a Word about Simulation

I have not implemented a simulator due to the short time frame. However we have identified the following issues for the implementation of a complete simulator:

1. A simulation necessitates a model that contains not only the system under development but also a description of other systems with which it communicates.
2. The binding algorithm presented in section II.5 must be extended to generate all the possible bindings in a given marking and not only one set of distinct enabled bindings. This is also complicated by the fact that a given transition or a given network may be non deterministic.
3. Simulation necessitates the detailed description of outputs of the transitions as functions of their inputs. This description necessitates a full featured language. That's why we have kept the implementation of the transitions as plain java code.

4. Transitions may have side effects like database access or communication activities that are necessary in the execution but are a disturbance in the simulation process.

III.3 Further Developments

Beyond our proof of concept implementation, the following further developments of the business process engine may be considered:

- Clustering implementation. The actual implementation of the main algorithm is a standalone single server. An obvious extension would be to implement a clustered server implementation with fail-over.
- Integration with Spring. The integration or the use of application frameworks like Spring may ease the integration of the business process engine with other technologies (like distributed transaction processors, distributed caches or other database technologies) in a smooth and non intrusive way.
- Development of a complete simulator.

Bibliography

[1] G.A. Agha, F De Cindio, G. Rozenberg (Edts.) : *Concurrent Object-Oriented Programming and Petri Nets.* Lecture Notes in Computer Science 2001, Springer-Verlag, 2001 ISBN 354041942X

[2] J. Desel, W. Reizig, G. Rozenberg (Edts) : *Lectures on Concurrency and Petri Nets.* Lecture Notes in Computer Science 3098, Springer-Verlag, 2004 ISBN 3540222618

[3] J. Fitzgerald, P.G. Larsen, P. Mukherjee, N. Plat, M. Verhoef : *Validated Designs for Object-Oriented Systems.* Springer-Verlag, 2005 ISBN 1852338814

[4] Gamma and al.: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley ISBN 0201633612

[5] C. Girault, R. Valk : *Petri Nets for Systems Engineering. A Guide to Modeling, Verification and Applications.* Springer-Verlag, 2002 ISBN 3540412174

[6] P.J. Haas : *Stochastic Petri Nets. Modeling, Stability, Simulation.* Springer Series in Operations Research, Springer-Verlag, 2002 ISBN 0387954457

[7] K. Jensen : *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol 1. Basic Concepts.* EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1992 ISBN 3540609431.

[8] K. Jensen : *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol 2. Analysis Methods.* EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1997 ISBN 3540582762

[9] K. Jensen : *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol 3. Practical Use.* EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1997 ISBN 3540628673

[10] L. Kleinrock : *Queuing systems Volume I: Theory.* John Wiley & Sons, 1975 ISBN 0471491101

[11] L. Kleinrock : *Queuing systems Volume II: Computer Applications.* John Wiley & Sons, 1975 ISBN 047149111X

[12] C. A. Petri : *Kommunikation mit Automaten:* Dissertation presented at the University of Bonn, 1962

[13] W. Reizig : *Elements of Distributed Algorithms. Modeling and Analysis with Petri Nets.* Springer-Verlag, 1998 ISBN 3540627529

[14] W. Reizig, G. Rozenberg (Edts) : *Lectures on Petri Nets I: Basic Models.* Lecture Notes in Computer Science 1491, Springer-Verlag, 1998 ISBN 3540653066

[15] W. Reizig, G. Rozenberg (Edts) : *Lectures on Petri Nets II: Applications.* Lecture Notes in Computer Science 1492, Springer-Verlag, 1998 ISBN 3540653074

3369 - 32