Design of a Petri Net-based Workflow Engine

Simone Pellegrini, Francesco Giacomini

INFN CNAF
Viale Berti Pichat, 6/2
40127 Bologna, Italy
Outline

- Introduction
- Background
- The Workflow engine
- Implementation and benchmarks
- Conclusions and Future Works
Outline

- Introduction
- Background
- The Workflow engine
- Implementation and benchmarks
- Conclusions and Future Works
Petri Nets are becoming a popular modeling formalism for Workflows because of their:

- Formal semantics
- Analysis tools
- Graphical representation

Only few Workflow Management Systems are based on such formalism

No clear guidelines about the design of a Petri Net-based Workflow engine
This paper proposes the design of a Workflow engine facing several aspects of Petri Nets:

- Non-determinism
- Firing mechanism
- Sub-Workflows

The engine is currently implemented in a WfMS http://wfms.forge.cnaf.infn.it
Outline

- Introduction
- Background
  - The Workflow engine
  - Implementation and benchmarks
- Conclusions and Future Works
Petri Nets (1/3)

- Formal representation of a discrete distributed system introduced by Carl Adam Petri (1962)

- It graphically depicts the structure of a distributed system as a directed bipartite graph

A Place/Transition net is a structure $N = (P, T, \text{Pre}, \text{Post})$ where:

- $P$: is a set of places $P = \{ p_1, p_2, q \}$
- $T$: is a set of transitions $T = \{ t \}$
- $\text{Pre}$: $P \times T \rightarrow N$ arcs directed from places to transitions $\text{Pre} = \{ a_1, a_2 \}$
- $\text{Post}$: $T \times P \rightarrow N$ arcs directed from transitions to places $\text{Post} = \{ b \}$
Petri Nets (2/3)

- The **non-deterministic** dynamic behavior of a net is modeled by means of *tokens*
- The state of a net (*marking*) is formally represented by the set of tokens which populates each place of the net.
The non-deterministic dynamic behavior of a net is modeled by means of tokens.

The state of a net (marking) is formally represented by the set of tokens which populates each place of the net.

\[ M_i = [1, 1, 0]^T \]

\[ M_{i+1} = [0, 0, 1]^T \]
Petri Nets (3/3)

Sequence

Choice (non-deterministic)

Concurrency
High Level Petri Nets (HLPN)

- The *classical* definition of the Petri Nets is *not* suitable for describing *complex processes* (e.g. Workflows)

- Several *extensions* have been introduced:
  - **Coloured Petri Nets**: tokens and places are *typed*, and tokens can also be *tested* and *manipulated* by means of *functions*
  - **Timed Petri Nets**
  - **Hierarchical Petri nets**: Petri Nets can be *nested*
The POV-Ray Workflow
Petri Net-based Workflow Languages

- Only few Workflow languages based on the Petri Nets formalism exist:
  - Yet Another Workflow Language (YAWL)
    - Extends the Petri Net formalism with several constructs introduced to meet the needs of workflow patterns
  - Grid Workflow Description Language (GWorkflowDL)
    - Formally based on High Level Petri Nets
    - Used by the the Grid Workflow Execution Service (GWES) in the Grid environment
Outline

- Introduction
- Background
- The Workflow engine
- Implementation and benchmarks
- Conclusions and Future Works
The Workflow Engine

Several issues have to be faced in the design and implementation of a Petri Net-based engine:

- **Mapping** the (High-Level) Petri Net formal model into an Object Oriented one
- **Emulation of non-determinism** in an *imperative paradigm* provided by mainstream languages
- **Definition of a reliable firing mechanism** which guarantee state consistency and high parallelism
The Static Model

- The engine keeps internally a representation of the Petri Net

The use of parametric programming helps in guarantee type safety of the P/T net
The Engine dynamics

- **Executing** a Petri Net means:
  - **Load** the P/T net model (from a *textual* description)
  - **Establish** the initial marking ($M_0$)
  - **Detect** and **Fire** *enabled* transitions until no further transitions are enabled

- Transitions are fired in a *non-deterministic* way:
  - Multiple-enabled (*non-conflicting*) transitions can trigger in parallel
The *Firing* Algorithm

- In order to solve **conflicts** and keep the design simple, the following firing *algorithm* has been defined:

  1. The engine **collects** enabled transitions

  2. If one or more transition are enabled
     a) The engine **selects** randomly **one** transition
     b) **Fires** the selected transition
     c) Return to point 1

  3. Otherwise, the *final marking* has been reached
In a Workflow, transitions can be associated with complex (*time-consuming*) tasks.

How to emulate parallelism if transitions are fired sequentially?

In the **phase 1** tokens are moved from incoming places and stored into the edge variables (x).
In a Workflow, transitions can be associated with complex \((\text{time-consuming})\) tasks.

How to emulate parallelism if transitions are fired sequentially?

The **phase 2** performs the operation \((f(x))\) and stores the result in the outgoing edge variables \((y)\).

Phase 2 does not affect the P/T net marking.
In a Workflow, transitions can be associated with complex (*time-consuming*) tasks

How to emulate parallelism if transitions are fired sequentially?

In the **phase 3** the results of the operation are stored to the outgoing places

Phase 3 affects the P/T net marking
All phases 1 and 3 must be executed with a **mutual exclusive access** to the Petri Net state.

Phases 2 can be executed on a separate thread.

- The duration of phases 1 and 3 is usually **negligible** if compared to phases 2 (which involves interaction with external services).
- Operations are **collected** and **served** by a FIFO queue.

The engine behavior can be easily described using simple **Finite State Machine**
Sub-Workflows

- A Petri Net \((\text{sw})\) can be embedded in a transition \((T_1)\)
- When \(T_1\) is fired the sub-Petri Net \((\text{sw})\) is executed

State consistency is guaranteed if all the phases \(1^*\) and \(3^*\) are executed *mutually*

The Sub-Workflow can be executed in a *separate instance* of the engine
Outline

- Introduction
- Background
- The Workflow engine
- Implementation and benchmarks
- Conclusions and Future Works
Implementation (1/2)

- The engine has been implemented in C++
  - Better performance
  - Strong type checking (*templates*)
- Workflows are described using the GWorkflowDL language
  - Transition *operations* and *guards* are expressed using a scripting language (Python)
- **Checkpointing** is also supported, thanks to the *state-based* nature of the Petri Nets formalism
The engine is currently used in the CppWfMS:

- A generic approach to Grid Workflows
- Abstract into Concrete Workflow mapping is based on the Sub-Workflow invocation mechanism

Currently working on the top of the EGEE gLite Grid middleware and infrastructure:

- Execution of Grid jobs is made possible by relaying on the Workload Management System (WMS) which take care of resource scheduling
The CppWfMS Architecture

Concrete Workflow

- **WS-I**: jobRegister
- **Local**: moveData
- WS-I: jobStart
- [fails]
- SubW: wait_for_termination
- Local: retrieveResults

Petri Net engine

- WMS WS-I
- LB WS-I
- glite middleware
- Service 1 WS-I
- Service N WS-I
- Local Operations
- Sub Workflows
- C/C++
- Python
In order to show the performance of the engine, the following Workflow has been defined.

The two values: **spread** and **level** define a family of workflows.

The more is the value of the spread, the more is the number of jobs which must be executed in parallel:

\[
\text{# of parallel tasks} = \text{spread}^{\text{level} - 1}
\]
The engine introduces a *fixed* overhead (~10%) referred to the ideal case.

The more is the *duration* of the activities, the less the overhead grows.

The fixed overhead can be reduced by introducing optimizations (mainly caching).
Conclusions and Future Works

- The proposed approach has demonstrated high reliability and good performance.
- Further work has nevertheless to be done in order to provide a full-featured WfMS.
- For the future we are going to focus in:
  - Making the system more robust and fast.
  - Porting existing applications to our platform.
  - Adding support to existing Grid infrastructures.
The End

Questions?