LUPA: A Workflow Engine

Emely Arráiz[1], Ernesto Hernández-Novich[1] and Roger Soler[1]

Universidad Simón Bolívar, Caracas, Venezuela
arraiz@ldc.usb.ve, emhn@ldc.usb.ve, soler@ldc.usb.ve

Abstract. Workflow Management Systems depend on a Workflow Enactment Service having several interfaces to establish communication with external applications, manage persistent information and exchange it with similarly capable systems. The study of business processes has shown multiple workflow patterns that have been modelled and implemented in several engines. The Workflow Management Council has combined commercial and academic efforts towards a standard structure for engines and information exchange regarding workflow processes. LUPA is a Workflow Engine designed around the Workflow Management Council Reference Model that implements basic workflow patterns with a graphical syntax, establishing their semantics on Interpreted Petri Nets with extensions. It provides a new Cancellation workflow pattern, that has also been used to provide an Iteration pattern with guaranteed termination.

1 Introduction

The Unified Language for Administrative Processes (LUPA) focuses on the programming and communicating tasks associated with the transaction management needed to fulfill business processes of an organization. It precisely models de Routes and Rules that information must follow in order to comply with the organizational policy.

LUPA's syntactic specification consists of Process Expressions built over a small set of operations that allow writing formulas which describe the Routes. It has operational semantics expressed with Interpreted Petri Nets covering the networked Process structure along with an Environment containing the associated information. A Cancellation operator is particularly interesting, it being able to model two transactions operating in parallel, keeping only the operations of the first one that finishes.

A simple interpreter component has been implemented following the formal syntactic and semantic specifications, leading to a prototype that is able to instantiate, execute and control processes defined under this schema. The prototype has been built following the WFMC recommendation regarding the Workflow System Reference Model.

This paper has been structured in sections. Section 2 gives an introduction regarding processes, process expressions and some of the associated graphs.

Section 3 defines the semantic for the process expressions, along with its operational semantics. Section 4 presents a brief review of Workflow concepts. Section 5 shows how LUPA fits into the Worflow Reference Model and Section 6 briefly comments on related and similar work both commercial and academic, while Section 7 briefly explains LUPA's implementation model. Finally, conclusions and further directions are presented in Section 8 .

2 Process Expressions

Each transaction has an associated process formed by the set of tasks the organization completes in order to effectively complete such transaction, and by the order and precedence in which those tasks have to be undertaken [1].

The set of tasks that make up the process has a "natural" graph structure given by the "programming". The idea of effective completion is seen as the fact that the transaction finishes and all its associated events have taken place. Therefore, a Process is understood as both the tasks and its programming. Being able to define the state of a process, seen as the set of tasks that are taken place at any given time along the process flow, becomes interesting. This notion can be modeled as the marking of the Petri Nets [2].

The Process Expressions (PE) are defined inductively as:

- T the finite set of simple tasks.
- EP the process expressions.
- pr_j predicates.
- $Clock$, a process that measures time.

Table 1. Process Expressions

If $t_i \in T$	then 1. t_i	$\in EP$ (simple task)
If $Z, Q \in EP$ then		
	2. $Z \bowtie Q$	$\in EP$ (sequence)
	3. $Z \oplus Q$	$\in EP$ (conjunction)
	4. $Z \circledcirc{pr} Q$	$\in EP$ (disjunction)
	5. $Z \oslash Q$	$\in EP$ (cancellation)
	6. $Z^{\circledcirc{pr}, Clock}$	$\in EP$ (iteration)

Each process expression has an associated graph, with a general form G(E) shown in Figure 1.

The transaction (a document) enters the net through the place **i** , is handled by the transition E, where the transaction is properly "processed", exiting the net through the place **o** afterwards.
All associated graphs have a single input place **i** and a single output place **o**.
Since we are working with bipartite graphs, the lexicon is similar to the one used when talking about Petri Nets.

Fig. 1. Generic Graph

We have the following cases with their associated graphs:

1. Simple Task: Process $E = t_i$ (t_i being the simple task which is atomic). This process E ends as soon as task t_i ends.
2. Sequence: Process $E = Z \bowtie Q$, stating that process Z is processed until it finished and its followed by process Q until it finishes. Process E ends when Q ends. Figure 2 shows its associated graph G(E) = G($Z \bowtie Q$). Where t is a "dummy process" that helps preserve process' Z output independent from process' Q input. That is, process t only copies its input to its output.

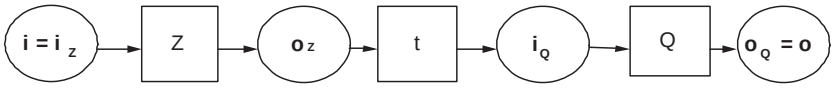

Fig. 2. Sequence

3. Conjunction: Process $E = Z \oplus Q$, stating that both processes Z y Q execute in parallel until both have ended thus ending process E.
4. Disjunction: Process $E = Z \textcircled{pr} Q$, stating that one and only one of Z or Q will execute depending on predicate's pr truth value. Process E ends when the process chosen to execute ends.
5. Cancellation: Process $E = Z \oslash Q$, stating that both processes Z y Q execute in parallel. When one of them finishes, process E will also finish. It also has the effect of cancelling (rolling-back) all the tasks completed by the other process. Figure 3 shows its associated graph G(E) = G($Z \oslash Q$). Where $\circledcirc$ is a "control place" that copies the input to both input places i_Z and i_Q, while $\circledcirc^*$ is another "control place" which will determine which sub-process finished first (be that Z or Q), handing its output to place **o**, while remembering to cancel (rollback) the output of the sub-process finishing last.

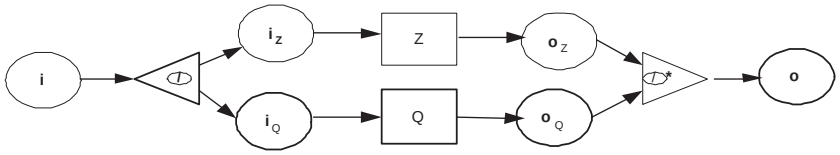

Fig. 3. Cancellation

The Cancellation operator ($\oslash$) is useful to start processes in parallel when only the results of one of them is needed, but which one will end first is not known or doesn't matter for the correct outcome of the process.

6. Iteration: Process $E = Z^{@pr,Clock}$, stating that process E will be repeatedly executed until predicate's pr truth value is false, or until the amount of time controlled by $Clock$ has passed. Figure 4 shows its associated graph.

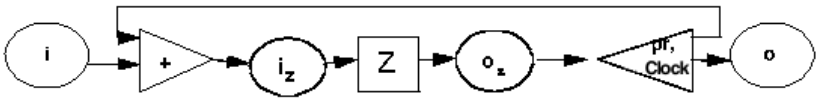

Fig. 4. Iteration

3 Operational Semantic

Unlike [3], we use Interpreted Petri Nets as [4], simplifying of standard definition and adapting them, thus partially using its potential. Our nets fall into Category 3 according to the Petri Net classification in [5].

Using Interpreted Petri Nets allows us to have and Environment carrying all the information needed for the transaction in process, and a classical Petri Net which establishes when to transform or create information on the Environment.

Definition 1 (Interpreted Petri Nets (IPN)) *An Interpreted Petri Net consists of a classical Petri Net* $< P, T, A >$*, an Environment* $Env =< D, OP, PR >$*, and two functions* φ *and* ψ *that link Net and Environment together by completing operations. Thus* $IPN =<< P, T, A >, Env, \varphi, \psi >$.

1. $D = D_{form} + D_{control}$ *, the disjoint union of the Environment states, the "form" and "control".*
2. $OP = \{op_1, op_2, \cdots, op_s\}$*, a set of operators*

$$op_i : D \to D$$

3. $PR = \{pr_1, pr_2, \cdots, pr_k\}$*, a set of predicates over* D

$$pr_i : D \to \{true, false\}$$

4. $\varphi : P \to OP$*, defining an operator* op_i *for each place of the Net.*
5. $\psi : T \to PR \times OP$*, defining a pair <predicate,operator> for each transition of the Net, being the specific task to complete there.*

An IPN works like a classical Petri Net, except that before firing a transition its associated predicate must have a true truth value when applied on the Environment, thus allowing the application of the operator.

The function φ will generally be the identity function, except for the place **i** which stands "isolated", that is $\varphi : (P - \{\mathbf{i}\}) \rightarrow \mathcal{I}$
A precise construction of φ for each input place **i**, is given for each case, being particularly important for cases 5 and 6.

3.1 Operational Semantics for Process Expressions

These semantics are given by the Interpreted Petri Net associated with the expression, constructed inductively according to the particular case. Being IPN_E the Interpreted Petri Net associated to the Process Expression E, that is: $IPN_E =<< P_E, T_E, A_E >, Env, \varphi_E, \psi_E >>$ where the Environment $Env =< D, OP, PR >$ is general while constructing the net. For all cases it must be true that: $P_Z \cap P_Q = \emptyset$ and $T_Z \cap T_Q = \emptyset$. We use the lambda notation [6] to specify each case. Case 5, the Cancellation operator, is specified as follows (all the specifications and the mathematical properties of the IPN are in [7]):

Case 5 Let $E = Z \oslash Q$.
restricted to:

$$(P_Z \cup P_Q) \cap \{\mathbf{i}, \mathbf{o}, \mathbf{b}, u, v, r, s\} = \emptyset$$
$$(T_Z \cup T_Q) \cap \{t_1, t_2, t_3, t_4, t_5\} = \emptyset$$
$$(\varphi_Z(P_Z) \cup \psi_Z(T_Z) \downarrow 1 \cup \psi_Z(T_Z) \downarrow 2) \perp (\varphi_Q(P_Q) \cup (\psi_Q(T_Q) \downarrow 1 \cup \psi_Q(T_Q) \downarrow 2)$$
$$\lambda d.M_{k1} \perp \lambda d.M_{k2}$$

and IPN_E:

$P_E = P_Z \cup P_Q \cup \{\mathbf{i}, \mathbf{o}, \mathbf{b}, u, v, r, s\}$
$T_E = T_Z \cup T_Q \cup \{t1, t_2, t_3, t_4, t_5\}$
$A_E = A_Z \cup A_Q \cup \{< \mathbf{i}, t_1 >, < t_1, s >, < t_1, u >, < t_1, i_Z >, < t_1, i_Q >, < u, t_2 >,$
$< u, t_3 >, < v, t_4 >, < v, t_5 >, < o_Q, t_3 >, < o_Z, t_2 >, < t_4, \mathbf{o} >, < s, t_4 >,$
$< t_4, u >, < t_4, r >, < t_2, v >, < t_3, v >, < r, t_5 >, < t_5, \mathbf{b} >\}$
$\varphi_E(\mathbf{i}) = \lambda d.(\varphi_Q(i_Q) \circ \varphi_Z(i_Z) \circ M_{k1} \circ M_{k2})$
$\varphi_E(P_E - \{\mathbf{i}\}) = \mathcal{I}$
$\psi_E(t_1) \downarrow 1 = \lambda d.d \downarrow k1$ (unique codomain index of M_{k1})
$\psi_E(t_1) \downarrow 2 = \lambda d.(\overline{M_{k1}} \circ Copy(D_{form}))$
$\psi_E(T_Z) \equiv \psi_Z(T_Z)$ (i.e. $\forall t \in T_Z,\ \psi_E(t) = \psi_Z(t)$,
working over $D_{form} + D_{control}$)
$\psi_E(T_Q) \equiv \psi_Q(T_Q)$ (i.e. $\forall t \in T_Q,\ \psi_E(t) = \psi_Q(t)$,
working over a copy of D_{form} and the unique $D_{control}$)
$\psi_E(t_2) \downarrow 1 = \psi_E(t_3) \downarrow 1 = \psi_E(t_4) \downarrow 1 = \psi_E(t_5) \downarrow 1 = \lambda d.true$
$\psi_E(t_2) \downarrow 2 = \lambda d.$ { $(\overline{M_{k2}} \circ Keep(Z))(d)$ si $(d \downarrow k2)$ *in* $D_{control}$; $(M_{k2} \circ Destroy(Z))(d)$ if not
$\psi_E(t_3) \downarrow 2 = \lambda d.$ { $(\overline{M_{k2}} \circ Keep(Q))(d)$ if$(d \downarrow k2)$ *in* $D_{control}$; $(M_{k2} \circ Destroy(Q))(d)$ if not
$\psi_E(t_4) \downarrow 2 = \mathcal{I}$
$\psi_E(t_5) \downarrow 2 = \lambda d.M_{k1}$

This case will have one of the following execution flows, starting from an initial marking μ_0:

$$S \in \{<< \mathbf{i}, t_1, < Z, t_2, t_4, \{\mathbf{o}\} > \| \ < Q, t_3, t_5, \{\mathbf{b}\} >>>, \\ << \mathbf{i}, t_1, < Q, t_3, t_4, \{\mathbf{o}\} > \| \ < Z, t_2, t_5, \{\mathbf{b}\} >>>\}$$

The set of output places is $P_o = \{\mathbf{o}, \mathbf{b}\}$.

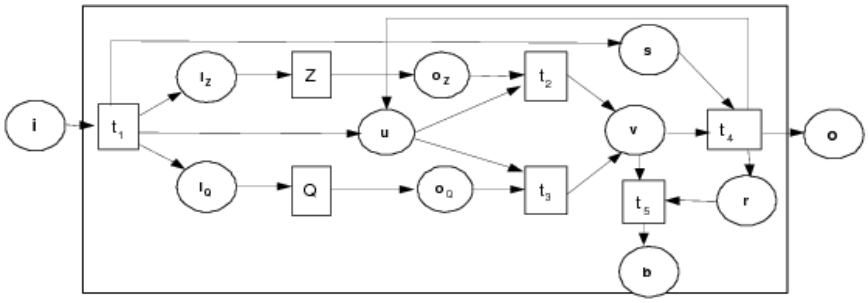

Fig. 5. Cancellation Operator underlying Petri-Net

Place **i** has a special φ component built (two M_{kj} for each cancellation operator ($\oslash$)). Each "memory" M_{kj} will keep track of specific conditions. M_{k1} makes sure that the cancellation operator remains blocked until both processes have finished, the last one being properly cancelled (rolled-back); while M_{k2} signals which process has to be cancelled. The output place **b** becomes a sink for all the tokens kept in the net by transitions that must be cancelled (rolled-back). Function *Keep* does permanently modify the form part of the Environment, whereas function *Destroy* discards any changes that were made by the process finishing last and thus being cancelled.

The iteration operator has been built as a specialized cancellation operator in such a way that all iterations will finish, either by completion or by the Clock timing out.

4 Workflow System and the Reference Model

The Workflow Management Coalition (WFMC) defines Workflow as the total or partial systematic automation of Business Processes during which documents, information or tasks are exchanged among participants, determined by a set of rules [8, 9]. It also defines a Workflow Management System as a set of software components used to support the definition, administration and execution of Workflow Processes [8, 9].

Research by the WFMC [10] have shown the feasibility of establishing a general model for Workflow Management System's implementation that fits the majority of existing solutions, as well as a basis for interoperability among them.

Since all Workflow Systems have a number of generic components interacting in predefined ways, the general model was built after identifying the main functional components of those systems as well as the interfaces between them. Several levels of functional capabilities have been established for each interface, thus specifying each one's minimal requirements.

In practice, the reference model of a Workflow System centers around at least one Workflow Engine [11] in charge of storing, activating and interpreting instances of processes as modeled by the organization. In fact, several engines can act simultaneously in a cooperative fashion, becoming a Workflow Enactment Service. It must provide interfaces that ease interaction with Process Definition Tools, External Support Tools such as Human or Cybernetic Agents, Control and Administration Tools, while helping exchange information with other Engines.

The Workflow Engine works on Process Definitions, which are nothing more than a Business Process presented in a way that eases its automatic manipulation either for analysis or application. This representation is just a network of activities and its relations, with several criteria allowing starting a finishing processes, while keeping process' relevant information at hand, who is involved and which applications are needed in order to complete it. There's a hierarchical relationship among process definitions, and the concept of a sub-process gives organizations the opportunity of reusing automation efforts, by solving simpler processes first and then tackle complex ones in a *"bottom-up"* fashion. Figure 6 shows the high-level Reference Model with its components and interfaces.

A simple *Workflow Enactment Service* was implemented, providing a runtime environment in which processes are instantiated and activated. A LUPA-based Workflow Engine handles interpretation and activation of the needed tasks, as described in the particular process definition, interacting with external resources in order to complete them.

5 LUPA and the Reference Model

While studying the LUPA proposed model and the many refinements of both "high-level" and "low-level" representations, LUPA's roles in the Reference Model were clearly identified.

5.1 LUPA as a Pre-Processor

The Workflow Enactment Service receives a process definition from a process design tool, and transforms it into an internal in-core representation which helps interpret it. During this conversion process [1] all the syntactic and semantic conditions are verified following LUPA's specification, checking if the process can be effectively constructed and executed, whether it has been defined directly or indirectly by means of combination of simpler existing processes. Once converted to the internal representation, the process is available for activation on

[1] A sort of "compilation" from a process "source form" closer to the designer, to an "executable form" closer to the engine that will execute it.

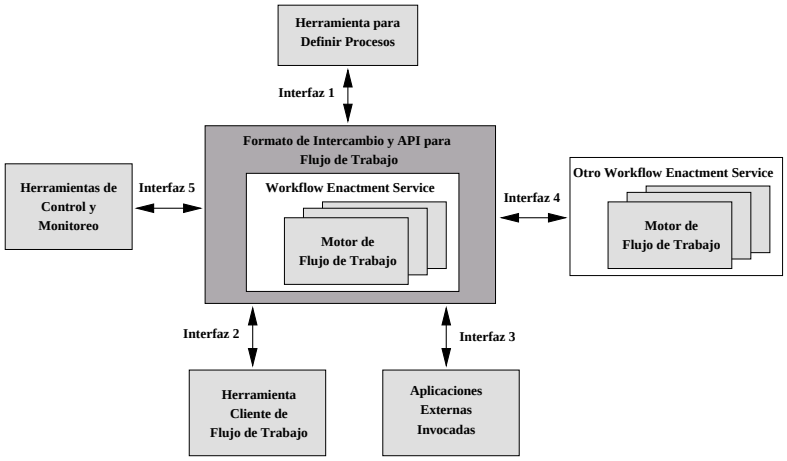

Fig. 6. Workflow Reference Model. Components and Interfaces.

any authorized user's request, or even moved to another Engine with a compatible intermediate representation. Our reference implementation has shown that LUPA fulfills these roles because:

- It receives a "high level" representation in terms of the graphical LUPA syntax. Even though during this work we did not develop a process designing application based on LUPA's graphical syntax, the XML process definition given as input to the engine is based on LUPA's regular expressions, showing not only it's feasibility but also suggesting an usable representation.
- It is able to convert this input definition into a more efficient one for execution purposes. During this conversion process, all the conditions required by LUPA's semantics are enforced and only properly conditioned definitions are accepted.
- It is able to turn this internal efficient representation into a textual "low level" one that helps ensure the persistence of it inside the work area of the Workflow Enactment Service. That is, if the service is stopped for whatever reason, the available process definitions and the executing instances will be stored in a persistent and consistent repository that allows resuming operations in the same Workflow Enactment Service, or in any other one to which they can be transported as long as its based on LUPA.

5.2 LUPA as an Interpreter

The Workflow Enactment Service takes a previously processed low level definition from its repository, and activates it as many times as necessary to handle specific cases. This activation and consequent workflow, must keep Control Information and Case Relevant Information, changing them according to the concrete operations associated with each task to be performed. At any given time there will be many types of processes, each one having several instances,

all of them executing simultaneously; knowing which tasks are pending, how far has any case advanced, suspending and resuming, finding out whether or not a process has finished, an even terminating o moving instances are some of the needed capabilites on such a system. Our reference implementation has shown that LUPA fulfills these roles because:

- The "low level" in core representation is based on a direct representation of the LUPA Net, with or without markings, be it active or not, sharing all the control (φ) and operation (ψ) functions in a single representation. A single representation of the flow structure (a single net) and a single set of operations, can model an abstract definition or as many instances as needed by means of the Petri Net Color extension to the LUPA Net and the Environments. The execution ability that models the operational semantic of the LUPA Nets as Interpreted Petri Nets is able to work on the appropriately colored marks and environment as needed by each case.
- The "low level" exchange representation is built from the "low level" in core representation, being able to select either the process definition (network and functions, without environment), a single instance (network, functions, and a single color marking and environment) or all instances.
- The "low level" in core representations are easily exchanged over a text-based protocol service, allowing for simple starting, stopping, checkpointing, cancelling and other simple operations over the available processes including exporting to other LUPA-based engines.

5.3 LUPA as an External Application Integrator

The Workflow Enactment Services must help the bidirectional exchange of information between the Workflow Engine and any external application needed to complete the tasks for each transaction. If the task must be completed by a human agent, it must be able to notify her of "pending tasks"; if the task must be completed by a cybernetic agent, it must be able to initiate execution possibly sending data and check its completion possibly retrieving processed data. At any given time there may be several ongoing external interactions, and knowing which ones are pending and checking for their current status must be easy. Our reference implementation has shown that LUPA fulfills these roles because:

- The Environment's representation provides a simple mechanism for procedural manipulation through a simple class interface. This eases writing operations (ψ) that exploit all the power and flexibility of Perl [12].
- The Environment's representation provides a simple mechanism for exporting and importing equivalent XML documents [13, 14] to external applications.
- The invocation of external applications is clearly split in such a way that the start of execution is separated from checking if it has finished. This allows for exporting process information prior to executing the external applications, and importing of results when finished. It also allows external applications to be executed asynchronously, while the Engine continues processing other tasks.

6 Related Work

An in depth analysis of thirteen commercial WorkFlow systems can be found in [15], while [16] does a similar analysis over ten WorkFlow languages proposed by academic communities. A large number of open source and free software initiatives [17] have worked on the process automation problem, providing different tools and systems. This project belongs on that list, but being different insofar as having a graphical syntax and a formal mathematical basis and expressive power of Petri Nets that combine with the Interpreted Petri Net techniques in order to link the workflow net, the case-specific and workflow-related information, and the operations over them. Only YAWL [18] has a similar approach, except that they use web-services and XML.

7 Modelling and Implementation

The interpreter for the language defined by LUPA's syntax and semantics has been modeled as closely as possible so as to be able to apply all the theory, analysis and verification tools available for Petri Nets [19]. It also has been built around the WFMC's Workflow Reference Model by following all the recommendations and basing it on open standards for information exchange.

The Execution Environments have been built as symbol tables, with total or partial exporting abilities to an XML representation. This eases portability and manipulation for the persistence infrastructure and external application interaction.

The Operators and Predicates have been modelled as functions in the native programming language chosen, with explicit information regarding domain and co-domain needed to perform the integrity checks while combining simple processes to build complex ones. Operators may be asked to perform asynchronous tasks, therefore their execution has been broken up in two phases: startup and test for finish. Introspective features provided by the programming language have been used to export the actual executable code for this functions (even if they were dynamically built at runtime) to an XML representation for persistence and interoperability purposes. Functions φ and ψ have been modelled in a similar fashion.

LUPA Nets have been modelled as Petri Nets, building the operational semantics using the aforementioned models for Environments and Functions and utility libraries provided by the programming language, in particular the ability to build closures. Each process operator has been implemented in such a way that it builds new LUPA nets as long as it is possible based on the restrictions imposed by each construction case. The network structure can be exported to an XML representation for persistence and interoperability purposes.

The programming language of choice is Perl. It allowed object-oriented techniques and functional programming techniques simultaneously, thus making the

programming as close as possible to the mathematical model. Perl's own abilities and the availability of several utility libraries allowed for the quick and easy development of a working prototype that is as compact and elegant as the formal structures it represents.

The Environment with the Petri Net and its markings, become an exact representation of the process status, and by using the color extension on Petri Nets, of all instances. The mechanisms in place for inspecting and exporting this structures provide clear and direct means of finding out completed, enabled and active tasks of any business process modeled with LUPA.

8 Conclusions and Further Directions

This work has formally defined the LUPA language as a way to model business processes.
It has the advantage of being highly expressive and adequate, by means of its fundamental flow patterns, including the cancellation and an iteration that guarantees termination.

The Petri Nets were chosen as the semantic model for process expressions because of their easily understandable and simple functionality, and the availability of analysis and graphical presentation tools [20].

Having developed a working Workflow Engine [19] shows that the LUPA language is useful and effective at its purpose. Having developed it using free software tools [21] enabled us to easily build a basic infrastructure following the Workflow Reference Model.

The reference implementation is highly portable. Choosing Perl as a programming language and libraries that are platform independent, ensure that the engine runs on GNU/Linux (development platform), any Unix, Win32, MacOS and even VMS, since Perl has been ported to all of them. The programming style also guarantees portability of all the programs without modification nor conditional execution. Selecting XML [13, 14] to build the low level representation of the many structures, combined with the introspective abilities of Perl to export source code out of its executables and dynamically generated data structures, makes then interchangeable across platforms and eases writing additional tools for analysis and verification.

References

1. R. Endl G. Knolmayer and M. Pfahrer. Modeling processes and workflows by business rules. *Lecture Notes in Computer Science*, 1806:16–29, 2000.
2. J. Peterson. Petri nets. *ACM Computing Surveys*, pages 223–252, Sept 1977.
3. W.M.P. van der Aalst. Workflow verification: Finding control-flow errors using petri-net-based techniques. *Lecture Notes in Computer Science, 1806*, pages 161–183, 2000.
4. G. W. Brams. *Réseaux de Petri: Théorie et pratique*. Texte de l' Agence de l' Informatique. Masson, 1983.

5. L. Bernardinello and F. De Cindio. A survey of basic models and modular net classes. *Advances in Petri Nets*, pages 304–351, 1992.
6. A. Church. *The Calculi of Lambda Convertion*. Princenton University Press, 1941.
7. E. Arráiz. Las expresiones de procesos y su semántica a través de las redes de petri interpretadas. Technical Report, Universidad Simón Bolivar, 2004.
8. Workflow Management Coalition. The workflow reference model. Disponible en URL:http://www.wfmc.org/standards/docs/tc003v11.pdf, 1995.
9. Workflow Management Coalition. Terminology & glossary. Disponible en URL:http://www.wfmc.org/standards/docs/TC-1011_term_glossary_v3.pdf, 1999.
10. The Workflow Management Coalition. Workflow management coalition terminology and glossary. Disponible en URL:http://www.wfmc.org, 1999.
11. S. Jablonski. Workflow management between formal theory and pragmatic approaches. *Lecture Notes in Computer Science, Volumen 1806*, 2000.
12. T. Christiansen L. Wall and J. Orwant. *Programming Perl, 3rd Edition*. O'Reilly & Associates, 2000.
13. World Wide Web Consortium (W3C). Extensible markup language (xml). Disponible en URL:http://www.w3.org/XML.
14. World Wide Web Consortium (W3C). Extensible markup language (xml) 1.1. Disponible en URL: http://www.w3.org/TR/2004/REC-xml11-20040204/, 2004.
15. B. Kiepuszewiski & A.P. Barros W.M.P. van der Aalst, A.H.M. ter Hofstede. Workflow patterns. 2003.
16. W.M.P. van der Aalst & A.H.M. ter Hofstede. Yawl: Yet another workflow language. *QUT Technical Report, FIT-TR-2003-04*, 2003.
17. C. E. Perez. Open source workflow engines written in java. 2004.
18. M. Dumas & A.H.M. ter Hofstede W.M.P. van der Aalst, L. Aldred. Implementation of the yawl system. *Lecture Notes in Computer Science, 3084*, 2004.
19. E. Hernández-Novich. Mirando procesos con lupa: Un motor de flujo de trabajo. Tésis de Maestría en Ciencias de la Computación, USB, 2005.
20. CNP group. Petri nets world. Disponible en URL:http://www.daini.au.dk/PetriNets, 2002.
21. D. A. Wheeler. Why open source software/free software(oss/fs)? look at the numbers! Disponible en URL:http://www.dwheeler.com/oss_fs_why.html, 2004.