Active Database System Realized by a Petri Net Approach

Lorena Chavarría-Báez and Xiaoou Li

Computer Science Department, CINVESTAV-IPN
Av. IPN 2508, Col. Zacatenco, C.P. 07360, Mexico

Abstract. An active database system executes actions automatically in response to events that are taking place either inside or outside the database. Developing an active database system, especially an active rule base, is not an easy task because some (unnoticed) errors may be introduced during its construction. In this paper, we present a Petri net-based approach to integrate active rules into traditional database system. We implemented our approach as a software system called ECAPNSim which not only has verification and simulation functionality but also allows us to develop multi-platform applications, i.e., a unique active rule base can work independently of the DBMS.

1 Introduction

An *active database system* (ADBS) is able to react automatically to meaningful events that are taking place inside or outside the database system. This reactive behavior is achieved by combining *active rules* into the traditional database system. Developing an active rule base, which is the core of an ADBS, requires considerably effort because of some of the following aspects: first, the knowledge represented by active rules into the computer-based system may have structural errors such as redundancy, inconsistency, incompleteness and circularity due to various communication problems between the expert and the active rule base designer. Second, the functionality of a rule base may be difficult to understand due to active rules are fired dynamically based on the previous flow of events. Finally, the tools associated with an active rule system may be minimal, with little support for browsing, monitoring, or debugging active rules. These points reflect a need for design rule analysis techniques and tools for debugging and explanation [1]. In our previous work we developed a Petri net-based approach, called Conditional Colored Petri Net (CCPN), for modeling and simulating active rules and their behavior. We implemented our approach as a software system, named ECA rule Petri net simulator (ECAPNSim) which turns a passive database into an active one [2]. ECAPNSim is introduced as a layer on top of the conventional database system, so it can establish communication with different database management systems (DBMS). Such integration allows us to develop multi-platforms applications, and makes it possible that an active rule base works independently of the DBMS. However, rule base verification, which is very important to guarantee that an active database works without errors, was not considered in the development of ECAPNSim.

Chavarría-Báez L. and Li X. (2008).
Active Database System Realized by a Petri Net Approach.
In *Proceedings of the 6th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems*, pages 150-159
DOI: 10.5220/0001730501500159

In this paper we focus on the integration of an error-free active rule base into a conventional database system using CCPN approach. First, we model the active rule base as a CCPN. Second, we detect structural errors by analyzing the structure of the Petri net. Taking into account the list of errors, the rule designer may perform suitable actions to correct them. Third, through the token flow in the CCPN, the active rule base behavior is simulated or performed. Thus, we can not only foresee but also ascertain the causes that trigger a rule. This work assures that a correct active rule base is integrated into a database system, in consequence, the obtained active system performs properly.

The remainder of the paper is organized as follows: Section 2 introduces active database system, knowledge and execution models are reviewed through a small example. Section 3 defines structure errors that an active rule base may have, as well as a Petri net-based method is proposed to detect them. Our software system ECAPNSim is presented in Section 4. A case study is used to demonstrate the ECAPNSim functionality. Section 5 shows related works. Finally, conclusion is drawn in Section 6.

2 Active Database System

An active database management system integrates event-based rule processing with traditional database functionality. Generally, its reactive behavior is achieved through definition of *Event-Condition-Action* (ECA) rules as part of the database. When an **event** happens the rule is triggered, then if the **condition** evaluation is true, the **action** is executed [1]. Generally, an active rule takes the following form: **ON** *event* **IF** *condition* **THEN** *action.*

An *event* is something that occurs at a point in time. The *condition* examines the context in which the event has taken place. The *action* describes the task to be carried out by the rule if the condition is fulfilled once an event has taken place. In the following there are some active rules based on the relation schema `account` (*num, name, balance, rate*) which contains registers about bank's accounts. Rules in the active rule base automatically enforce some of the bank's policies for managing customers' accounts.

Example 1. Bank's policies for managing customers' accounts.

Rule 1. When an new account is registered, if that account has a balance less than 500 and an interest rate greater than 0%, then that account's interest rate is lowered to 0%.

Rule 2. When an account's interest rate is modified, if an account has a balance less than 500 and an interest rate greater than 0%, then that account's interest rate is lowered to 0%.

Rule 3. When an account's balance is modified, if that account has an interest rate greater than 1% but less than 3%, then that account's interest rate is raised to 2%.

Rule 4. When an account's balance is modified, if that account has an interest rate greater than 1% but less than 3%, then that account is deleted.

Above policies can be represented as active rules as follows:

Rule 1. `ON` insert `account IF` insert.*balance* ¡ 500 and insert.*rate* ¿ 0
`THEN` update `account` set *rate* = 0 where *balance* ¡ 500 and *rate* ¿ 0

Rule 2. `ON` update `account`.*rate* `IF` update.*balance* ¡ 500 and update.*rate* ¿ 0 `THEN` update `account` set *rate* = 0 where *balance* ¡ 500 and *rate* ¿ 0

Rule 3. `ON` update `account`.*balance* `IF` update.*rate* ¿ 1 and update.*rate* ¡ 3 `THEN` update `account` set *rate* = 2 where *rate* ¿ 1 and *rate* ¡ 3

Rule 4. `ON` update `account`.*balance* `IF` update.*rate* ¿ 1 and update.*rate* ¡ 3 `THEN` delete from `account` where *rate* ¿ 1 and *rate* ¡ 3

Suppose the event update `account`.*balance* has been detected (*signaling phase*), so **Rule 3** and **Rule 4** are triggered (*triggering phase*). Then, conditions of **Rule 3** and **Rule 4** must be evaluated to determine if their corresponding actions can be executed (*evaluation phase*). Let's assume the condition update.*rate* ¿ 1 and update.*rate* ¡ 3 is true, rule execution is scheduled (*scheduling phase*). For simplicity, rule execution will be done by following the order of rules in the list. Therefore, **Rule 3**'s action will be executed first, only then **Rule 4**'s action will be executed (*execution phase*). After **Rule 3**'s action execution, the event update `account`.*rate* is signaled, so **Rule 2** is triggered (*immediate coupling mode*) and execution process is repeated until there is no rule eligible to trigger. On the other hand, when **Rule 4**'s action is executed, rule processing finishes since there is no rule such that is triggered by the event delete from `account`.

3 Rule Base Development

Developing an active rule base involves transferring expertise from the human expert through the active rule base designer into the computer. During this process many errors, which have to be detected to ensure a proper perfomance of the system, may arise because of communications problems between the expert(s) and the rule designer. *Verification* concerns the correctness and appropriateness of the structure of a rule base, and must be an essential part of the whole system development. In our previous works [2] - [5], we have tackled active rule base verification issues.

Structural errors include redundancy, inconsistency, incompleteness and circularity. We refer the event and condition parts as *premise* of an ECA rule, and denote an ECA rule as $R_i(E_i, C_i, A_i)$ where E_i, C_i and A_i are the event, condition and action of rule R_i, respectively. Here we take Redundancy error as an example to show our verification process.

Redundancy is characterized by *redundant* and *subsumed* rules. Given a rule *i* and a rule *j* which take the same action, if rule *i* has a more restrictive premise than rule *j*, then rule *i* is a *subsumed* rule because whenever it succeeds, rule *j* must also succeed.

Definition 1. ***(Subsumed Rules.)*** *A rule $R_i(E_i, C_i, A_i)$ is subsumed by a rule R_j (E_j, C_j, A_j) if the following conditions are met:*

(1) $E_j \subseteq E_i$,
(2) $C_j \subseteq C_i$,
(3) $A_j \subseteq A_i$

When in Definition 1, $E_j = E_i, C_j = C_i$ and $A_j = A_i$ the two rules are absolutely identical, so they are *redundant rules.*

Sometimes, rules are not subsumed (or redundant) completely, i.e., only some of its elements (event, condition, action) are subsumed (or redundant). We identify those rules as *partially subsumed (redundant) rules* since they can also cause redundancy.

Definition 2. ***(Partially subsumed Rules.)*** *A rule R_i (E_i, C_i, A_i) is partially subsumed by a rule R_j (E_j, C_j, A_j) if only two of its elements are subsumed by those of R_j.*

Rules R_i and R_j are *event - condition subsumed rules, if the following conditions are fulfilled.*
(1) $E_j \subseteq E_i$
(2) $C_j \subseteq C_i$
(3) $A_j \neq A_i$
Similarly, we can define *event - action subsumed rules and condition - action subsumed rules.*

In Example 1, **Rule 1** and **Rule 2** are partially redundant condition - action rules because they evaluate the same condition and perform the same action.

Redundancy doesn't cause a bad performance to the rule base; however, problems arise when one of the redundant rules is changed or removed but the rest don't because their behavior still will be presented in the rule base.

In order to detect structural errors in the rule base, we developed a verification process which consists of three phases: *rule normalization*, *rule modeling*, and *rule verification* which we describe in the following.

3.1 Rule Normalization

This step translates the original active rule base into a set of *atomic rules*. An atomic rule is that whose event and condition are a conjunction of one or more primitive events and conditional clauses, respectively, and its action is only one instruction. We consider the event algebra described in [1]; however, we have restricted our analysis - with no loss of generality - to disjunction and conjunction operators since the rest of the operators have the same structure, for example, if events in conjunction operator are ordered then we have the sequence operator.

A rule $R_i(E_i, C_i, A_i)$ can always be divided into several atomic rules by the following steps:

Step 1. If $Ri(E_i, C_i, A_i)$ is atomic, its OK. If not go to Step 2.

Step 2. Transform each element of $Ri(Ei, Ci, Ai)$ into disjunction form, by using rules from the boolean algebra, so that each element consists of one or more disjuncts each one of them is a conjunction of one or more instructions. If the transformed rule has no disjunctions in its elements, then it is an atomic rule according to the definition. Otherwise, go to the next step.

Step 3. Divide $Ri(E_i, C_i, A_i)$ into a set of atomic rules whose premises and actions are the obtained disjuncts in Step 2.

Active rule base of Example 1 doesn't need to be normalized since each rule is atomic.

3.2 Rule Base Modeling

An active rule base as well as its interaction can be represented by several models. However, some of them are not suitable to perform active rule base verification because they represent the rules very general. The Conditional Colored Petri Net (CCPN) model

[2], which is an extension of Colored Petri Nets, represents each element of an active rule and it provides a way to represent composite events as well as to consider condition evaluation. Since this model is an extension of Petri nets it has a sound basis to analyze active rules behavior. An active rule is mapped to a CCPN structure as follows: the rule is mapped to a transition where condition is attached, event and action parts are mapped to input and output places of a transition, respectively. Matching between events and input places has the following characteristics:

- *Primitive* places, P_{prim}, represent primitive events;
- *Composite* places, P_{comp}, represent composite events;
- *Copy* places, P_{copy}, are used when one event triggers two or more rules. An event can be shared by two or more rules, but in PN theory, one token needs to be duplicated for sharing. A copy place takes the same information as its original one;
- *Virtual,* P_{virt}, places are used for accumulating different events that trigger the same rule. For example, when the event part of a rule is the composite event OR.

All places in the CCPN are identified by P. $P = P_{prim} \cup P_{comp} \cup P_{copy} \cup P_{virt}$. Rules and transitions are related in the following form:

- *Rule* transitions, T_{rule}, represent rules;
- *Composite* transitions, T_{comp}, represent composite event generation;
- *Copy* transitions, T_{copy}, duplicate one event for each triggered rule.

All the transitions in CCPN are represented by T. $T = T_{rule} \cup T_{comp} \cup T_{copy}$.

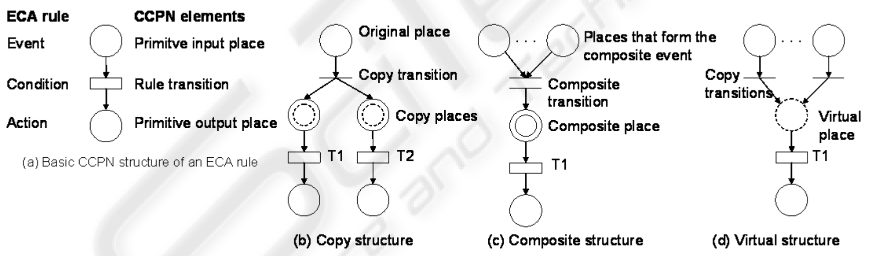
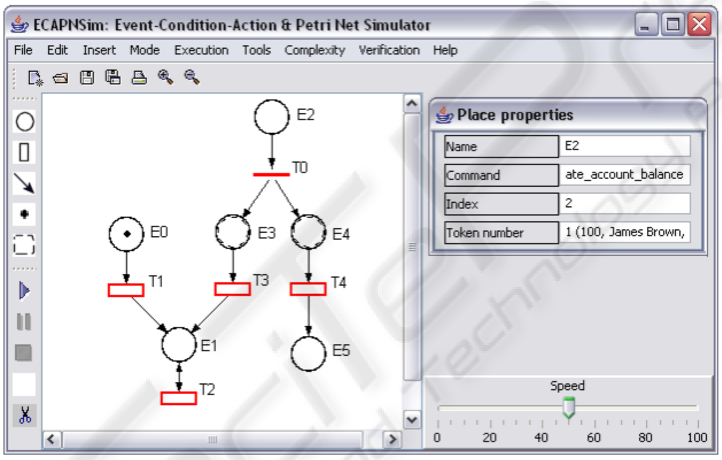


Fig. 1. CCPN structures of ECA rules. (a) Basic CCPN structure of an ECA rule (b) Copy structure (c) Composite structure (d) Virtual structure.

Figure 1 shows the basic structures to which an ECA rule can be mapped. In its most basic form, event, condition and action of a rule matches with a primitive input place, a rule transition and a primitive output place, respectively, as shown in Figure 1(a).If an event triggers two or more rules it has to be duplicated by means the copy structure depicted in Figure 1(b). Composite events formation is considered in CCPN using the composite structure drawn in Figure 1(c). Composite transition's input places represent all the events needed to form a composite event while its output place correspond to the whole composite event. Finally, we use the virtual structure to model the composite event `OR` as standing for in Figure 1(d). Virtual place acts as an "event store", i.e., when a rule is triggered by several events that place accumulates them and each one of them can be used to trigger the rule. The CCPN model of a set of ECA rules is formed by connecting those places that are output and input places at the same time, i.e., those places that represent both the action of one rule and the event of another rule. Figure 2 shows the CCPN model of rules of Example 1 in our ECAPNSim interface. Events/actions: insert `account`, update `account`.*rate*, update `account`.*balance*, and delete `account`, defined in active rules of Example 1, are represented by the primitive places E0, E1, E2, and E5, respectively. Places labeled as E3 and E4 are copy places, so, they contain the same information as place E0. Each rule in Example 1 is represented by a rule typed transition in the CCPN, so that, transition T1 represents **Rule 1**, transition T2 stands for **Rule 2**, and so on. Also each rule typed transition is attached with the condition of its corresponding rule. Since transition T0 is a copy typed transition it doesn't represent any rule; however, it allows us to trigger two rules at the same time. Condition of transition T0 is always true.


ECAPNSim: Event-Condition-Action & Petri Net Simulator
File Edit Insert Mode Execution Tools Complexity Verification Help
E2
T0
E0
E3
E4
T1
T3
T4
E1
E5
T2
Place properties
Speed
0 20 40 60 80 100

Name	E2
Command	ate_account_balance
Index	2
Token number	1 (100, James Brown,

Fig. 2. CCPN with token "insert `account`(100, James Brown, 300, 0.5)" in place E0.

In CCPN transition firing is defined by two steps: first, verify if a transition is enabled; second, verify if it can fire according to the condition evaluation result. If a transition is enabled and the condition evaluation is true, then that transitions fires.

3.3 Rule Verification

Structural errors considered in this paper have special patterns which allow us to identify them. Through the analysis of some abnormal structures we conclude that the rules which may have errors exhibit the following characteristics (either one of them or both):

1. **Rules with the Same Action.** Those transitions that have primitive output places with more than one input arc may cause errors because they represent rules with actions executed by more than one rule

2. **Rules with Common Events.** Those transitions that have primitive places which are duplicated through a copy type transition since they represent rules with events that trigger more than one rule.

During CCPN construction relevant information is generated, we can retrieve it using the following methods:

– getInputPlaces(T). It returns the input primitive places of a transition T. For example, getInputPlaces(T3) = $\{$E2$\}$. getInputPlaces(T) is different from $^{\bullet}$T since each one of them gives us the *primitive* and *all* input places of transition T, respectively.
– cond(T). It returns the conditional clauses stored in the rule typed transition T. For example, cond(T4) = $\{$update.*rate* ¿1, update.*rate* ¡ 3$\}$.

By using the CCPN model and the information obtained with above methods, we developed a set of error detection properties. Here we show our property to detect redundancy.

Property 1. Redundancy. Let t_i and t_j, $1 \leq i, j \leq m, i \neq j$; be two transitions which represent the **Rule i** and **Rule j** of **RB**, respectively, which execute the same action and whose conditions have, at least, a common conditional clause. We say that **Rule j** is subsumed by **Rule i** if t_i and t_j satisfy the following conditions:

a). $getInputPlaces\,(t_i) \subseteq getInputPlaces\,(t_j)\,;$
b). $cond(t_i) \subseteq cond(t_j);$
c). $t_i^{\bullet} = t_j^{\bullet}.$

Errors detected in the rule base of Example 1 are the following: (1) Rule 1 and Rule 2, as well as Rule 3 and Rule 4, are partially redundant condition - action rules. (2) Rule 3 and Rule 4 are conflicting rules since their actions contradicts each other under the same premise. (3) Rule 2 is a circular rule. (4) Rule 2 is an unreachable rule. (5) Rule 4 is a dead-end rule. With the complete list of errors, rule designer must perform suitable actions to correct each error.

4 Integrating Rule Base and Database System

ECA Petri Nets Simulator (ECAPNSim)was originally developed for modeling and simulating ECA rule base behavior in active database system [2]. ECAPNSim integrates reactive behavior to traditional databases without special ECA rule syntax and semantics development. Figure 3 shows the buildtime architecture of ECAPNSim. The Rule editor module of ECAPNSim takes an active rule base written in a text file and generates automatically its corresponding CCPN model (ECA-CCPN converter model).

In order to assure an error-free active rule base we add a new module "Rule Verifier" into ECAPNSim, see Figure 3, so that errors such as redundancy, inconsistency, incompleteness and circularity, can be detected automatically. Rule Verifier uses the structure of the obtained CCPN model as well as information generated during CCPN construction to check a rule base according to our verification approach. For example, in Figure 2 transitions T1 and T2 are detected partially redundant condition - action rules by Rule Verifier since both of them evaluate the same condition and execute the same action (conditions b) and c) of above Property 1). ECAPNSim produces a report of all the anomalies detected, then notify the rule base designer and suggest him do appropriated corrections.

Using ECAPNSim rule behavior can also be simulated. Suppose a token with information: insert `account`(100, James Brown, 300, 0.5) in place E0 of the CCPN of Figure 2, which means that the customer James Brown has a new bank account which is identified with the number 100, with a balance and rate of $300 and 0.5%, respectively. Then transition firing steps are verified. 1) First, since there is a token in place E0, transition T1 is enabled. 2) Second, since transition T1 is attached with the condition insert.*balance* ¡ 500 and insert.*rate* ¿ 3 and the new account's balance and rate are less than 500 and greater than 0, respectively, condition evaluation is true. So, the token is removed from place E0 and deposited in place E1. The token generated from this transition firing consists of the information: (100, James Brown, 300, 0) because it corresponds to the action of transition T1. In this moment transition T2 is enabled since there is a token in place E1. However, condition evaluation is false since transition T2 is attached with condition update.*balance* ¡ 500 and update.*rate* ¿ 0 but the updated balance's value is 0. So, the token is taken away from the CCPN and the rule base processing finishes. As you can see, even though a cycle was detected during verification phase, at runtime that cycle never happens.

5 Related Work

Rule base verification has been widely investigated particularly with respect to production rules [7], [10] - [12]. Earlier work in this area focussed on comparing in pairs the rules of the rule base, trying to discover certain relationships between their premises and conclusions. Recent techniques use graphical representations, such as directed graphs or Petri nets, of the rule base to detect the different structural errors [11], [12]. Petri nets based approaches model the rule base using a Petri net and the verification is performed through reachability analysis of paths in the graph. However, those approaches depend on the initial mark of the net to detect errors, therefore, some errors may be not detected. In this sense, our work can detect all the errors described in this paper since we focus on the structure of the net and information regarding rules.

Very few work can be found on active rule base verification. References [9] and [8] tackled this topic by verifying the rule properties, such as triggerability, join applicability, rule coverage, rule cascading, and postcondition satisfaction of an active rule base, which are mainly focused on checking semantic correctness of active rules. Our approach can work together with their approaches to check both semantic correctness and structural errors.

Unlike approaches such as Starburst [6], EXACT [1], and SAMOS [1], we verify and simulate an active rule base in the same platform ECAPNSim. Through ECAPNSim an active rule base can work with different DBMSs.

6 Conclusions

Rule Verifier tool development makes it possible that active rules integrated into a traditional database are error free. Comparing with other existing active database systems, our approach has the following advantages:

(1) Formal Petri nets analysis tools can be used to detect structural errors.

(2) ECAPNSim makes rule base verification and simulation in the same platform.

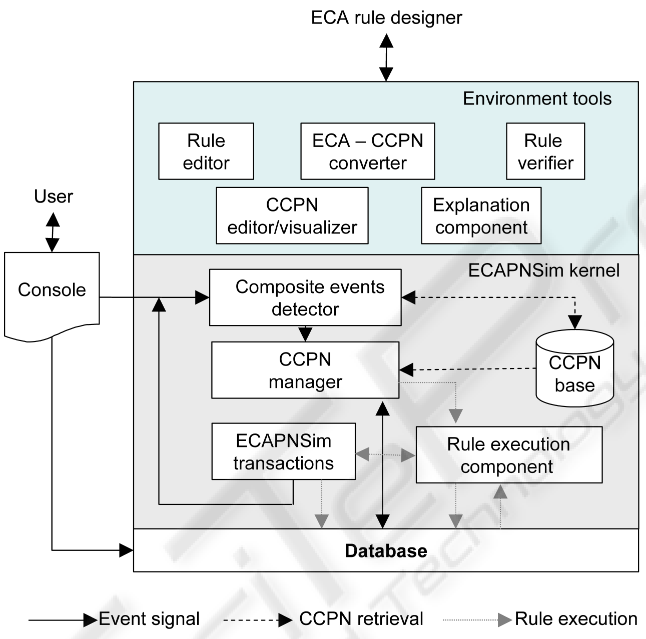


Fig. 3. ECAPNSim buildtime architecture.

References

1. Paton, N. , Díaz, O.: Active Database Systems. *ACM Computing Surveys*, Vol. 31. No. 1. (1999) 62-103
2. Li, X., Medina-Marín, J., Chapa, S.: Applying Petri nets on active database systems. *IEEE Trans. on System, Man, and Cybernetics, Part C: Applications and Reviews,* Vol. 37, No. 4. (2007) 482-493
3. Chavarría-Báez, L., Li, X.: Knowledge Verification of Active Rule-Based Systems. In D.S. Huang, K. Li, and G.W. Irwin (Eds.): *ICIC2006: Intelligent Control and Automation. Lecture Notes in Control and Information Sciences,* Vol. 344 Springer-Verlag, Berlin Heidelberg (2006) 676-687
4. Chavarría Baez, L., Li, X.: Static Verification of Active Rule-Based System, In *10th IASTED Intl. Conf. on SoftwareEngineering and Applications.* Dallas, Texas, USA, (2006) 514-084.

5. Chavarría-Báez, L., Li, X.: Verification of ECA Rule Base via Conditional Colored Petri Nets, In *IEEE Intl. Conf. on Systems, Man, and Cybernetics*. Montreal, Canada. (2007) 7-10
6. Widom, J., *The Starburst Rule System*. In: J. Widom (ed.): Active Database Systems Triggers and Rules For Advanced Database Processing. Morgan Kaufmann Publishers, San Francisco, California (1996)
7. He, X., Chu, W., Yang, H., A New Approach to Verify Rule-Based Systems Using Petri Nets, *Inf. and Soft. Tech.,* vol. 45, no. 10, pp. 663-670, 2003
8. Augusto, J. C., and Nugent, C., A New Architecture for Smart Homes Based on ADB and Temporal Reasoning, In *Toward a Human Friendly Assistive Environment* (Proc. of 2nd Intl. Conf. on Smart homes and health Telematic, ICOST2004), Assistive Technology Research Series, Vol. 14, pp. 106-113, IOS Press, Singapore, Sept. 15-17, 2004.
9. Fraternali, P., Teniente, E., Urpí, T., Validating Active Rules by Planning, In *Proc. of the 3rd Intl. Workshop on Rules in Database Systems*, Springer LNCS Vol. 1312, 1997, pp. 181-196.
10. Yang, S., Fuzzy Rule Base Systems Verification Using High-Level Petri Nets, *IEEE Trans. on Knowledge and Data Engineering*, vol. 15, no. 2, 2003, 457-473.
11. Wu, C. and Lee, S., A Token-Flow Paradigm for Verification of Rule-Based Expert Systems, *IEEE Trans. on Systems, Man and Cybernetics- Part B: Cybernetics*, vol. 30, no. 4, 2000, 616-624
12. M. Ramaswamy, S. Sarkar, and Y.S. Chen, ”Using Directed Hypergraphs to Verify Rule-Based Expert Systems”, *IEEE Trans. on Knowl. and Data Eng.*, vol. 9, no. 2, 1997, pp. 221-237