Improving Control Flow Verification in a Business Process using an Extended Petri Net

Ganna Monakova, Oliver Kopp, and Frank Leymann

Institute of Architecture of Application Systems, University of Stuttgart, Germany
{monakova, kopp, leymann}@iaas.uni-stuttgart.de

Abstract. In a business process, control flow decisions are based on the evaluation of conditions. Thus, conditions must be considered for control flow verification. This paper shows how the Petri nets based control flow verification can be improved by analysing conditions and logical relations between them. We outline a Petri net extension with predicate transitions, which are responsible for conditions evaluation based on the collected knowledge, and effect places, which contain fact tokens representing the effects of certain operations and decisions made.

1 Introduction

As the complexity of business processes grows, the need for automatic verification becomes more important. We show an approach for verification for processes modelled with Petri nets. The properties to verify represent the constraints on possible execution traces. An example of such a constraint is "the payment must always be followed by a shipment", which can be expressed using LTL as $G(Payment \rightarrow FShipment)$. To show that this constraint is fulfilled, it must be proved that there is no possible execution path that contains a payment before a shipment (temporal dependency) and that there is no execution path that contains payment without shipment (causal dependency). Note that the above constraint allows the execution of shipment without payment.

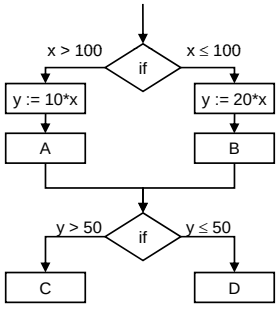


Fig. 1. Example process fragment

The process fragment depicted in Fig. 1 shows why the data relations should be considered for the control flow verification. The fragment presents two `if`-construcrts, each having two branches. We define an acivity *execution condition* as a Boolean expression that is constructed recursively by analyzing all conditions that have to be satisfied to enable the execution of this activity [1]. For example, the execution condition of activity A from Fig. 1 is $(x > 100)$ and the execution condition of B is $(x \leq 100)$.

Assume that the constraint "A must always be followed by C" must be satisfied for a process containing this fragment. This constraint is always satisfied,

since the execution condition of C is implied by the execution condition of A and the fact that y becomes equal to $10 * x$ if the left branch of the first if is executed. Current Petri net based verification approaches abstract from the data and their relations and therefore make non-deterministic choice in both of the switch-constructs, which makes the phantom execution path $(A; D)$ possible.

This paper shows how a Petri net can collect knowledge and use the collected knowledge to reason about the next step. Sect. 2 presents related work in this field. The proposed extensions are shown in Sect. 3. Sect. 4 describes how the process knowledge is collected during the Petri net analysis and Sect. 5 shows how the collected knowledge is used for the predicate transitions evaluation. Sect. 6 demonstrates the analysis of the process fragment shown in Fig. 1 using the presented approach before Sect. 7 concludes.

2 Background and Related Work

A number of non-deterministic decisions take place during the analysis of a Petri net: selection of a specific if-branch, skipping or executing an activity, and entering or exiting a loop. In the business process the decisions depend on the evaluation of the corresponding conditions: branching condition, join condition, loop condition. A join condition is a starting condition of a branch and is a Boolean formula over the states of all incoming links [2]. Typically, the translation of a business process to a Petri net ignores these conditions. The justification is that the actual data coming into the process is not known at static validation and therefore the conditions cannot be evaluated. The process model, however, contains read-write dependencies between the activities, which can be used for an advanced verification [3]. Logical relations between variables can be captured by execution conditions as described in [1]. In this paper, we show how logical relations can be used for a more precise Petri nets based control flow verification. The technique presented in this paper can be used as extension to the mapping of a BPEL process to the corresponding Petri net [4].

An overview of existing BPEL formalizations and verification approaches is provided in [5]. We presented in [1] a summary of the presented approaches and showed that none of them includes the interplay between previous and following decisions. Thus, all of the approaches include the phantom path $(A; D)$ in their analysis. The work of [1] put loops and scopes as future work. In the work presented here, we include loops and scopes, since the mapping of BPEL to Petri nets is complete [4].

3 An Extension for Petri Nets

The conditions constrain the execution of the business process. In addition, the control flow decisions made in the past can influence the decisions in the future, as the example of Fig. 1 illustrates: if the left branch of the first `if`-activity is taken, then y is set to $10 * x$, where $x > 100$ according to the branch condition. This will influence the branch selection of the second `if`-activity. This implies that the execution trace, which contains activity A and activity D, is impossible. It will, however, be considered as possible during the Petri net analysis if the data

conditions are neglected. Such phantom execution trace can only be detected if the relation between decisions leading to the execution of the activities A and D are known.

Each decision is bound to a certain condition. If a decision has to be made, the condition is evaluated and, depending on the evaluation result, a certain path in a workflow is chosen. During process runtime, the evaluation of the condition is simple, since instance data is available. At design time, however, instance data is not available and thus only relations between process variables and between conditions can be analysed. If a certain path in a workflow is chosen, then the condition of this path is true (e.g. $x > y$). Thus, even if we do not know the actual data, we know that the data relations captured in the path condition (e.g. $x > y$) are in force. We also say that the decision produces an *effect* relations.

We add a *predicate transition* to decide whether a certain path *can* be executed. A predicate transition is responsible for the evaluation of the condition for a certain execution trace based on the collected knowledge. A *decision transition* is the transition responsible for selecting the actual execution trace from all possible execution traces. A trace is considered to be an alternative if the corresponding condition was evaluated to true or unknown by the predicate transition, see Sect. 5. We record the effect of the decision made by producing a token for an *effect place* added after each fired decision transition. This token means that the condition on this path is true and thus the relation represented by this condition is in force.

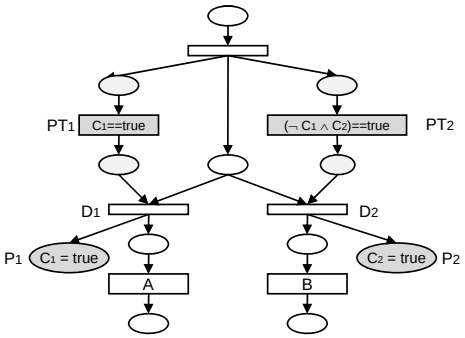
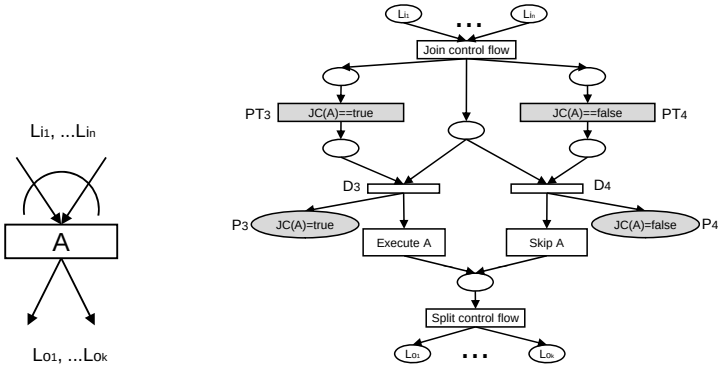


Fig. 2. `if` activity in an extended Petri net

Effect places, predicate and decision transitions for an if-activity with two branches are shown in Fig. 2. The predicate transitions PT_1 and PT_2 are responsible for the evaluation of the branch conditions: C_1 for the left and C_2 for the right branch. If the corresponding condition evaluates to true, the token will be produced in the outgoing place, which in its turn will enable the selection of the corresponding branch. If the condition evaluates to false, then the token from the incoming place will be consumed and no token for the outgoing place will be produced. Note, that the branch conditions are adjusted in such a manner that only one of the conditions can simultaneously evaluate to true. This complies with the `if`-activity execution semantic [6]. If evaluation of both conditions returns *unknown*, both predicate transitions will produce a token and the `if`-branch will be chosen non-deterministically. As soon as the branch selection decision has been made by a decision transition, the relations between process variables represented by the recorresponding branch condition come in force. This is indicated by a token in the corresponding effect place: P_1 for the left and P_2 for the right branch. Note that the effect places cannot be put directly after the decision transitions,



(a) `flow` activity

(b) `flow` activity extended Petri net

Fig. 3. `flow` activity

to honor the fact that both branch conditions can evaluate to unknown and thus the actual decision will be made by the decision transition.

Fig. 3(a) shows an activity in a BPEL `flow`. The activity has n incoming links and a join condition defined on the status of these links. Fig. 3(b) shows an extended Petri net for this activity. Here, the predicate transitions PT_3 and PT_4 are responsible for evaluation of the join condition, which helps the decision transitions D_3 and D_4 to decide whether the activity A is to be executed or skipped. The extension required for loop constructs is similar to the if-activity extension and is not shown in this paper due to the space limitations. An `assign`-activity establishes relations between process variables which are come in force after the activity has been executed. Therefore, an additional effect place is added after the transition representing an `assign` activity. The next section presents how the produced effects can be stored to enable reasoning on the colected knowledge.

4 Collecting Knowledge

The variable and condition relations currently in force are represented by the tokens in the effect places. The effect places, and thus the knowledge about the data relations, may either result from a decision effect or from a relation between process variables introduced by an `assign` activity. A decision effect is a result of a decision made for transition conditions, join conditions, `if` and `pick` branches. Let C be the condition of the path selected by the decision transition. Then a new fact $C = true$ is added to the knowledge base. The relations between process variables captured in the assign statements can also influence decisions. For example, if an assignment $y := x + a$ was executed and it is known that $a > 0$, then it can be derived that $y > x$. If an assign activity $x := f(y_1, \ldots, y_n)$ is executed, then the statement $x = f(y_1, \ldots, y_n)$ becomes a fact and is added to the knowledge base. Thereby, every occurrence of the same variable on the left side of assignment gets a new index each time an assignment fact is added to the knowledge base. If a variable occurs on the right side of the assignment, the variable with the highest index currently available in the knowledge base is used. Note that this is different to the CSSA approach [7], as in this case there is no need to consider the

exclusive or concurrent read-write accesses to the same variable (expressed with the ϕ and π-functions in CSSA). The reason for this is the step-by-step analysis considered in this paper, which implies that the knowledge base cannot contain contradictory information: only the relations captured in the assign statements on the chosen branch are added to the knowledge base and the order of the concurrent assignments is the one selected by the Petri net navigator. A `receive`, `pick` and `invoke` activity can be considered to contain an implicit assign of the message content to the process variables.

5 Reasoning on the Collected Knowledge

A predicate transition represents an invocation of a reasoner. A reasoner evaluates the transition condition based on the current knowledge in the knowledge base. This section shows how the evaluation of the condition can be reduced to the satisfiability problem. Let $F_1, \ldots, F_n$ be the current facts in the knowledge base, let C be the condition to be evaluated. The condition evaluates to true if it can be proved that C can be derived from the current facts in the knowledge base. Formally speaking, the following must hold: $F_1, \ldots, F_n \vdash C$. To prove this, the following formula is checked for its satisfiability: $\left(\bigwedge_{i\in\{1,\ldots,n\}} F_i\right) \wedge \neg C$. If this above formula is unsatisfiable, then C will always evaluate to true for this execution path and therefore a token will be produced by the predicate transition responsible for the evaluation of the condition C. If the above formula is satisfiable, the following formula is checked: $\left(\bigwedge_{i\in\{1..n\}} F_i\right) \wedge C$. If this formula is unsatisfiable, then C will always evaluate to false for this execution path. In this case the transition will not produce any token. If both formulas are satisfiable, then the decision can only be made based on the concrete data and therefore both cases should be considered for the analysis. In this case, the predicate transition produces a token which will compete with other tokens. The actual decision will be made non-deterministically by the decision transition in the same way as for the non-extended Petri net: one of the tokens will be consumed, the others will remain in their places and wait for the backtracking and selection/consumption of the next token. The collected facts in the knowledge base represent the logical relations between process variables currently in force. The satisfiability of the above formulas based on the current relations is checked using the Satisfiability Modulo Theories (SMT) solver Yices [8]. An SMT solver solves satisfiability problems for Boolean formulas containing predicates of underlying theories. Such theories can be, for example, theories of arrays, lists and strings [9]. In addition, an SMT solver can be extended with new theories as shown in [10].

6 Example

Fig. 4 illustrates the analysis of the process fragment from Fig. 1. Fig. 4(a) shows the first invocation of the reasoner on the current knowledge base. Since no previous information is available, the knowledge base is empty and therefore returns *unknown* for both conditions. Fig. 4(b) shows the status after the left branch was non-deterministically taken. This decision transition produces a token

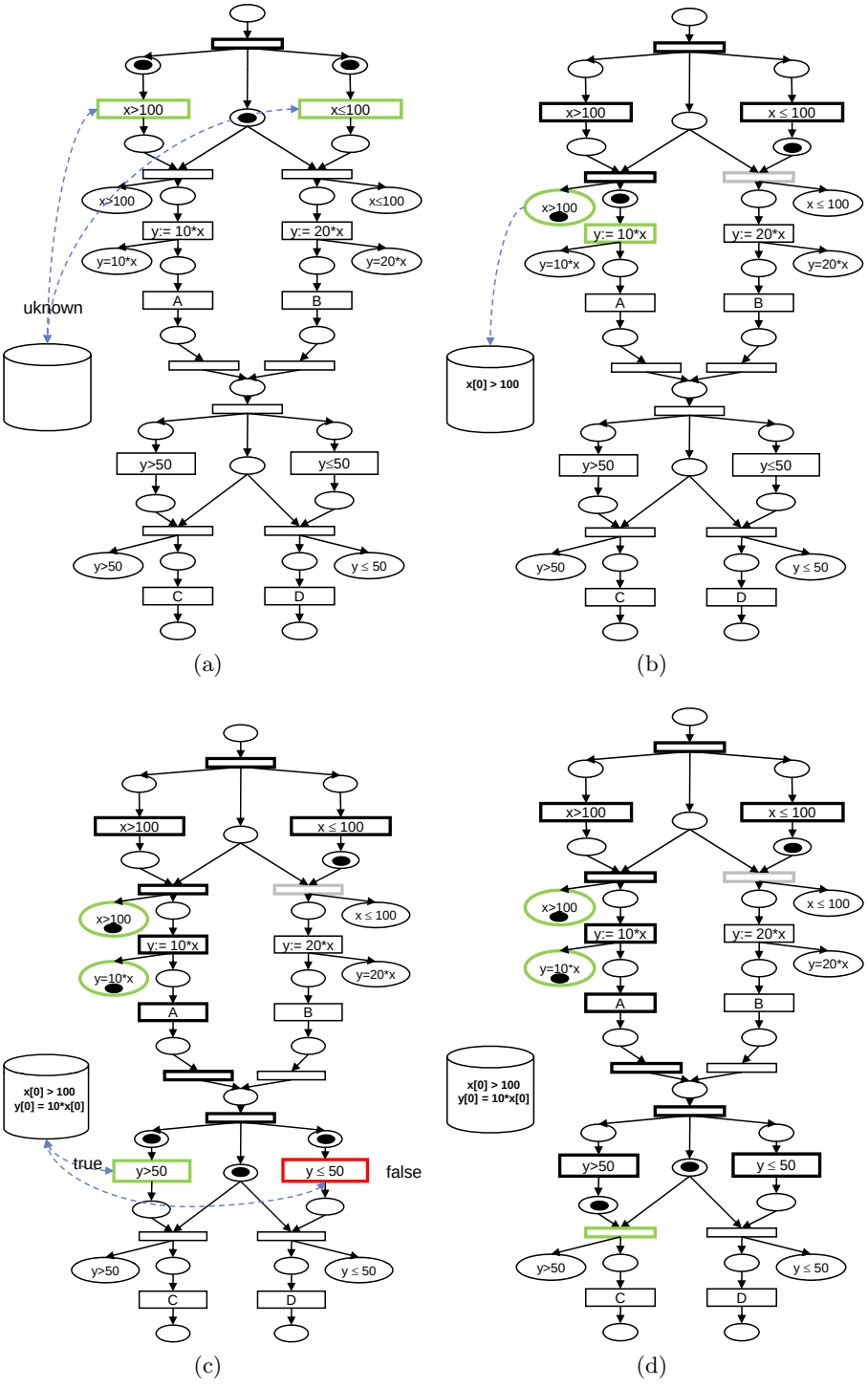


Fig. 4. Analysis using knowledge base and a reasoner

in the effect place $x > 100$ and the corresponding relation is added as a fact to the knowledge base. Fig. 4(c) shows the next invocation of the reasoner. There, the condition $y > 50$ evaluates to true and $y \leq 50$ evaluates to false. Fig. 4(d) shows the status after the firing of the predicate transitions for the second `if` statement. The predicate transition of the right branch consumes the token on its input place, but does not produce an output token, since $y \leq 50$ evaluates to false, while the predicate transition of the left branch produces an output token. Thus, only the left branch of second `if` activity is enabled.

7 Conclusions and Outlook

This paper showed how a Petri net based verification of a business process can be enhanced by adding effect places and predicate transitions. We showed how the conditions on the predicate transitions can be evaluated using the knowledge collected during the Petri net analysis. This enables resolving the non-deterministic decisions if the current decision strongly depends on the previously made decisions. Thus the "phantom" paths can be removed from the reachability graph which makes the analysis more effective and precise.

The presented approach can also be used to analyze compositions of business processes, called choreographies. In this case, the knowledge base is shared by all processes and thus each process is aware of the constraints on the input data.

Our future work is to investigate the impacts of our work on current Petri net reduction techniques. We are going to integrate the presented approach in LoLA [11] to prove the applicability of the approach.

References

1. Monakova, G., et al.: Verifying Business Rules Using an SMT Solver for BPEL Processes. In: BPSC. (2009)
2. Leymann, F., Roller, D.: Production Workflow – Concepts and Techniques. Prentice Hall PTR (2000)
3. Moser, S., et al.: Advanced Verification of Distributed WS-BPEL Business Processes Incorporating CSSA-based Data Flow Analysis, IEEE Computer Society (2007) 98–105
4. Lohmann, N.: A Feature-Complete Petri Net Semantics for WS-BPEL 2.0. In: WS-FM. (2007)
5. Breugel, F.v., Koshkina, M.: Models and Verification of BPEL. `http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf` (2006)
6. OASIS: Web Services Business Process Execution Language Version 2.0. (2007)
7. Lee, J., Midkiff, S.P., Padua, D.A.: Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs. In: International Workshop on Languages and Compilers for Parallel Computing, Springer (1997)
8. Dutertre, B., de Moura, L.: The YICES SMT Solver (2008) Available at `http://yices.csl.sri.com/`.
9. Beckert, B., et al.: Intelligent Systems and Formal Methods in Software Engineering. IEEE Intelligent Systems **21**(6) (2006) 71–81
10. Nelson, G., D., O.: Simplification by Cooperating Decision Procedures. ACM Transactions on Programming Languages and Systems **1**(2) (1979) 245–257
11. Schmidt, K.: LoLA: A Low Level Analyser. In: ICATPN. (2000) 465–474