Welcome
to the 2009 Spring Simulation Multiconference
(SpringSim'09)
March 22 - 27, 2009
Doubletree Hotel San Diego-Mission Valley; San Diego, CA, USA

Sponsored by The Society for Modeling and Simulation International (SCS)

Co-located with the 2009 SISO Spring Simulation Interoperability Workshop (SIW)

The 2009 Spring Simulation Multiconference (SpringSim'09) is an annual conference which covers state-of-the-art developments in computer simulation technologies, as well as scientific, industrial, and business applications. Areas covered include high-performance computing technologies, models and algorithms, GUI visualization technologies, communications and much more. Application disciplines covered include advanced telecommunication; computer systems; military, government & aerospace; environment, energy, and other industries. The conference includes keynote speeches presented by technology and industry leaders, technical sessions, professional development courses and seminars, as well as vendor exhibits. Scientists, engineers, managers, educators, and business professionals who develop or use simulation tools are invited to participate and present original papers. Proposals are solicited for papers, panels, tutorials, workshops, seminars, exhibits, social activities and for other presentation, discussion and sponsorship formats. People are always welcome to benefit by taking an organizing role. SpringSim'09 offers many ways to promote simulation products and to enhance corporate images. You are invited to use the Spring Simulation Multiconference in ways that best serve your interests.

All accepted papers will also be published in the ACM Digital Library.
An Approach for Formal Verification of Business Processes

Armen Kostanyan, Vardan Matevosyan, Samvel Shoukourian, Anna Varosyan

IT Educational and Research Center
Yerevan State University
1 A.Manooogian Str., Yerevan, Armenia, 0049
Email: armko@ysu.am, vardan@armsoft.am, samshouk@sci.am, annavarosyan@yahoo.com

Keywords: Business Process, formal verification, cycle transformation

Abstract
Business process (BP) verification has become an essential step in ensuring quality of designed BPs. The BP correctness can be checked by the process graph structural verification. Functional/behavioral verification of BPs is considered as another case of verification to verify the process semantic correctness. This can be done either formally or via process simulation. The most commonly used case of functional verification is a simulation of the process execution for all possible cases, but this can lead to exhaustion. That is the reason why the formal verification of processes should be applied whenever possible. This paper contains an extension of the acyclic BP verification algorithm for the general case, i.e. a new algorithm is suggested for verification of BPs that contain cycles. The solution is based on the notion of reducing verification of a given cyclic BP to verification of some acyclic one. Results obtained from applying the algorithm to some BPs from the IT Infrastructure Library (ITIL) are brought forward for consideration.

1. INTRODUCTION

Business Process Management is a key technology in the overall enterprise strategy of an organization [1]. The business processes have to be specified by a language called workflow specification language [1, 2]. It is very important to verify the business process correctness prior to the operation [3].

The business process correctness can be checked by the process graph structural verification, such as cycle analysis, branching path synchronization, etc [3, 7, 8].

The functional/behavioral verification of a process can be considered as another case of verification to verify the process semantic correctness. This can be done either formally or via process simulation. The most commonly used case of functional verification is a simulation of the process execution for all possible cases, but this can lead to exhaustion. That is the reason why the formal verification of processes should be applied whenever possible.

The formal verification requires knowledge about the underlying business process (internal structure of tasks, data flow, etc.). Two assertions have to be specified for the process. First, the process input assertion, called precondition, has to be satisfied prior to the process execution. Second, the process output assertion, called postcondition, has to be checked at the end of each process execution. The process is considered to be correct if the value of postcondition is “true” for all possible executions.

In general, insolubility (or exhaustion) hampers direct application of formal verification to complex cases. Therefore, the desire to apply a strict criterion must be neglected and replaced with finding effectively verifiable, acceptable conditions for verification. More explicitly, algorithms should be constructed, which are applicable to any object considered for verification. These algorithms must give one of the three answers: correct, incorrect, or unknown under restrictions on execution time. An algorithm is called partial-recognizing if it answers correct (incorrect) only for correct (incorrect) objects. A partial-recognizing algorithm (PRA) can answer “the correctness is unknown” in the case of a correct object. But when it answers “the object is correct,” the object is correct for certain.

By adopting the described approach to a business process (BP) [1] formal verification, one way to create a PRA is to approximate the input and output assertions of a BP by special language constructions called molds [4,5]. A mold specifies the form of a relation between variables at the input and the output of a BP. Filling the form with concrete values results in a particular assertion described by a mold. In many cases, molds simplify and increase the effectiveness of procedures for determining BP correctness. From a BP and its input assertion, it is possible to construct a special intermediate mold that describes a complete set of invariant relationships at the BP’s output. We call a relationship invariant if it is true after any BP interpretation or execution. A BP is correct if the output assertion implies from the obtained intermediate mold.

The widest class of acyclic BPs was constructed for the Workflow Model of BPs [1], for which the use of molds leads to a sufficiently effective (polynomial complexity) verification algorithm [6]. In the meantime, cyclic BPs are also used currently in many applications [7, 8].
This paper contains an extension of the algorithm for the general case, i.e. a new PRA is suggested for verification of BPs containing cycles. The solution is based on the notion of reducing a cyclic BP verification to an acyclic BP verification. The transformation of a process graph is based on the idea of intervals suggested for program data flow analysis by F.E.Allen and J.Cocke [9]. This idea, initially used in optimization of compilers, was used afterward in many other applications [13].

The general representation of the approach is presented in Figure 1:

Figure 1: General schema of the approach

Section 2 presents model definition that is based on the IBM’s MQSeries Workflow model [1]. The problem definition is brought forward for consideration in section 3. The next section contains the approach for transformation of the given BP to an acyclic BP. Results obtained from applying the approach to the IT Infrastructure Library (ITIL) of BPs [10-12] is available in the next section. Further instructions on the presented approach are explained in the last section.

2. MODEL DEFINITION

This section presents model definition by extending the IBM’s MQSeries Workflow model in two directions. The first direction includes the necessary formalism required for consideration of the formal verification problem. The second direction extends the process semantics to include a description of the cyclic processes.

The main model components are activities and connectors. The activities are associated with a context being defined as data passing to an activity. It is called input container. An activity also returns data called output container. Some output container elements of an activity can be passed to the input container elements of other activities or to the external memory. All data elements are collected in the set V. Control and data connectors provide connections between the activities. A control connector has an associated Boolean predicate called transition condition. A directed graph based on sets of activities and control connectors is called control flow of a workflow/business process.

The execution of an acyclic business process can be described in following general stages. At first, initial activities have to be executed and transition conditions of their outgoing control connectors should be calculated. Then, the set of ready activities have to be determined by selecting the activities, whose all incoming transition conditions have already been calculated. The activity with the highest priority should be selected from this set, its join condition has to be calculated and the activity must be executed in case if the value of the join condition is “true”. The process execution will be stopped if the set of ready activities is empty.

The semantics of an acyclic business process are presented in [1] in more detail.

The W-process extends the formal model of acyclic BPs by adding the concept of memory for BPs.

The following formal definition of W-processes is given by using the designations below:

- If \( x = \langle x_1, \ldots, x_n \rangle \) is a tuple, and \( 1 \leq i_1, \ldots, i_k \leq n \) is a set of indices, then \( \pi_{i_1, \ldots, i_k}(x) \) denotes the tuple \( \langle x_{i_1}, \ldots, x_{i_k} \rangle \).
- If \( X \) is a set, the \( \rho(X) \) denotes the set of all subsets of \( X \).

**Definition:** A tuple \( P = \langle T, N, C, E, \Phi, i, o, \Psi, \Delta, M \rangle \) is called W-process, if

1. \( T \) is a finite set of types. The set of values for \( t \in T \) is denoted by \( \text{DOM}(t) \) and is called the domain of \( t \).
2. \( N \) is a finite set of activities having some priorities.
3. \( C \) is the finite set of transition conditions.
4. The set \( E \subseteq N \times N \times C \) is the set of connectors with the following restrictions
   - \( E \) is unified: \( \forall e, e' \in E : \pi_{1,2}(e) = \pi_{1,2}(e') \Rightarrow \pi_3(e) = \pi_3(e') \).
   - The \( G=(N, E) \) graph is acyclic.
5. \( \Phi : N \rightarrow F \), where \( F \) is the set of all activity join conditions. The join condition of activity \( A \in N \) is defined as a Boolean expression \( \Phi(A) \) depending on elements of set \( A^-=[e \in E \mid \pi_2(e)=A] \). Suppose \( \Phi(A) \equiv 1 \), if \( A^- = \emptyset \).
6. \( i : N \cup C \rightarrow IV \) is input container map where \( IV \) is the set of input variable tuples of activities and conditions. Each input variable is unique and belongs to any type from \( T \).
7. \( o : N \rightarrow OV \) is output container map where \( OV \) is the set of output variable tuples of activities. Each output variable is unique and belongs to any type from \( T \).
8. \( \Psi : N \cup C \rightarrow E \), where \( E \) is the set of activities and conditions of all possible implementations.

The implementation of activity \( A \in N \) is defined as a map \( \Psi(A) : \times_{e \in A} \text{DOM}(v) \rightarrow \times_{e \in A} \text{DOM}(v) \). The implementation of condition \( p \in C \) is defined as a map \( \Psi(p) : \times_{e \in p} \text{DOM}(v) \rightarrow \{0, 1\} \).
9. M is an external memory. Each variable from M is unique and belongs to any type from T. Let \( A^* = \{e|e \in E \} \) is the set of outgoing transition conditions from the activity A.

10. \( \Delta : N \times (N \cup C \cup \{M\}) \rightarrow \cup_{B \in N, \epsilon \in NC} \Phi(o(A) \times (i(B) \cup M)) \) is data connector map with following restrictions:
   a) \( \{A \in N, B \in N \cup C \cup \{M\}\} \equiv \Delta(A, B) \subseteq o(A) \times i(B) \).
   b) \( A \in N \Rightarrow \Delta(A, M) \subseteq o(A) \times M \).
   c) \( \{A \in N, B \in N \cup C \cup \{M\}\} \), \( <v1, v'> \in \Delta(A, B) \)
   \( <v2, v'> \in \Delta(A, B) \Rightarrow v1 = v2 \).
   d) \( \forall v, v' \in A \Delta(X, Y), \text{DOM}(v) = \text{DOM}(v') \).

Let us extend the definition of a W-process to cover a cyclic business process. An activity of the cyclic business process can be a cycle activity which can be a cycle head. The cycle head is the cycle entry point. The behavior of activities that are heads of cycles differs from the behavior of other activities. They have two different join conditions. The first join condition is made on control connectors that are connecting cycle activities with its head (inner connectors). The second join condition is made on control connectors that are connecting the cycle head with other activities (outer connectors). The cycle head executed in one of those join conditions would be satisfactory.

Let us now give some necessary definitions and define the cyclic business process.

**Definition (path)** A path is an ordered sequence of activities and their connecting control connectors \( a_{i1}, e_1, a_{i2}, e_2, \ldots, a_{ik}, e_{k-1}, a_{i1} \) in which each \( a_{i1}, e_1, a_{i2}, e_2, \ldots, a_{ik}, e_{k-1}, a_{i1} \) is an immediate predecessor of activity \( a_{i+1} \) \( (\pi_0(e) = a_i \& \pi_1(e) = a_{i+1}) \).

**Definition (cycle)** A closed path or cycle is a path in which the first and last activities are the same - \( a_{i1}, e_1, a_{i2}, e_2, \ldots, a_{ik}, e_{k-1}, a_{i1} \) and the \( a_{i1} \) is the cycle head which is the single entry point of the cycle.

**Definition (localized cycle)** The cycle is called a localized cycle if the only point of intersection with other cycles can be the cycle head.

**Definition (cyclic business process)** The cyclic business process is defined as follows:
1. A tuple \( <T, N, C, E, \Phi, i, o, \Psi, \Delta, M> \) as defined for W-process.
2. All cycles are localized cycles.
3. \( N_C \subseteq N \) is the set of activities, each of which is the head of a cycle. Suppose \( K_A \) is the set of activities, contained in the cycle having the head \( A \in N \).
4. \( C^{i+1}(A) := \pi_0(e) = A \& \pi_1(e) \in K_A \) is the set of inner transition conditions for \( A \in N \).
5. \( C^{i+1}(A) := \pi_0(e) = A \& \pi_1(e) \in K_A \) is the set of outer transition conditions for \( A \in N \).
6. \( \forall A \in N, \Phi(A) \in \Phi^{i+1} \), where \( \Phi^{i+1} := \{A \& \vee p | p \in \{p',p'' | p' \in C_{\Phi}^{i+1}(A)\} \} \)

7. The set of inner join conditions for \( \forall A \in N \) is \( \Phi_{AC} := \{A \& \vee \pi | \pi \in \{p',p'' | p' \in C_{\Phi}^{i+1}(A)\} \} \)

We will assume that \( \forall A \notin N, \Phi_{AC} \equiv \Phi_{AC} \equiv \emptyset \).

8. \( A \in N, A \) can be activated \( \subseteq (\Phi(A)(\pi_0(p)), \pi_1(p)) = 1 \) \( \vee \Phi_{AC}(\pi_0(q)), \pi_1(q)) = 1 \), \( C_{\Phi}^{i+1}(A) = \{p_1 \ldots p_n \} \), \( C_{\Phi}^{i+1}(A) = \{q_1 \ldots q_2 \} \).

**Definition (Branching activity)** A cycle activity, having at least one outgoing control connector with the cycle outside, is called a branching activity.

**Definition (State registers)** Each branching activity has a state register that can have one of the values <0; 1; 2; ...> with following meanings:
1. 0 means that the branching activity has not been executed yet,
2. n; (n >= 1) means that branching activity has been executed n times.

To present the business processes graphically, the following shapes are used:

3. **CORRECTNESS OF BUSINESS PROCESSES**

**Definition (process correctness)** Let \( P \) be a business process. Let \( \alpha, \beta \) be two predicates depending on the state of the external memory M of \( P \). Let us call \( \alpha \) and \( \beta \) a precondition and a postcondition, respectively. In this case, \( P \) is correct with respect to \( \alpha \) and \( \beta \), if
\[
[\alpha(M) \Rightarrow \beta(M(P)) = 1]
\]
where MP denotes the external memory state after the execution of the \( P \) process.

Let \( P \) be a cyclic business process. The goal is to transform \( P \) to an acyclic process \( Q \) such that if:
\[
[Q \text{ is correct/incorrect} \Rightarrow P \text{ is correct/incorrect}]
\]

4. **TRANSFORMATION OF CYCLIC PROCESSES TO ACYCLIC PROCESSES**

Figure 2: BP graphical representation
The general scheme of the transformation approach is presented in Figure 3.

**Figure 3: General schema of the transformation approach**

At first transformation conditions have to be checked. There are two cases of a process when the transformation can be unsuccessful. The first one is when the process graph is not an interval graph. The second one is when the graph is an interval graph, but it is not transformable.

All steps of the transformation approach are described as basic operations used by the transformation algorithm.

### 4.1. Basic Operations

This section presents basic operations and definitions used by the transformation algorithm based on the idea of intervals suggested for program data flow analysis procedure [9]. Presented below are the appropriate definitions of an interval and first order graph [9].

**Definition (Interval).** Given a node \( h \), an interval \( I(h) \) is the maximal, single entry subgraph, where \( h \) is the only entry node and all cycles contain \( h \). The unique interval node \( h \) is called the interval head, or simply the header node. An interval can be expressed in terms of the nodes contained in it: \( I(h) = (n_1; n_2; \ldots; n_m) \).

**Definition (First order graph).** The intervals described so far have been formed from the nodes given in the initial control flow graph. These intervals are called the basic or first order intervals and the graph, from which they were derived, is called the basic or first order graph.

Basic transformation operation high level descriptions are listed below:

1. **Interval set construction;**
   a. Find all intervals,
   b. Select intervals containing a cycle;
   c. Remove from the interval activities, that are not cycle activities;

2. **Replacement of each interval with an equivalent activity,**
   a. Remove cycle activities with their control and data connectors;
   b. Define new activity;
   c. Construct a new activity input container from input data elements of cycle activities connected with the cycle outside;
   d. Construct a new activity output container from output data elements of cycle activities connected with cycle outside and variables defined for the cycle branching activity state registers;
   e. Add corresponding data connectors between the new activity and other activities for each removed data connector between the cycle activities and the cycle outside.

3. **Interval condition construction for each interval branching activity;**
   a. Construct the set of branching activities that are supposed to be activated before the given branching activity;
   b. Construct the set of branching activities that have to be activated after the given branching activity;
   c. Build a branching activity condition to support two possible executions of a branching activity. The first one is the cycle execution termination without completing the cycle. The second one is all cycle activities execution at least once.

4. **Construction of interval transition conditions based on interval conditions;**
   a. For all branching activities construct interval conditions;
   b. Instead of control connectors connecting cycle head with the cycle outside, add control connectors connecting them with the new activity;
   c. Instead of control connectors connecting branching activities with the cycle outside, add control connectors connecting the new activity with those activities and formulate their transition conditions based on previous transition conditions and branching activity interval conditions;
   d. If the activity was connected with more than one branching activity, combine all transition conditions in one with an OR.

Formal descriptions of the basic operations are detailed bellow.

**Operation 1: construct IntervalSet**

**Input:** Cyclic business process \( P_b = (\mathcal{T}, \mathcal{N}, \mathcal{E}, \Phi, \Phi_{\mathcal{C}}, \alpha, \Psi, \Delta, M \) with graph \( G(N,E) \)

**Output:** \( Sc \), set of intervals

**Method:**

1. Construct the set of intervals \( S \) by interval finding algorithm [9].
2. \( Sc = \{ I(h) \in S \ | \ \exists (a, b) \in E \ & (a, b) \in I(h) \} \).
3. For each \( I(h) \in Sc \), do
   \[ L = \{ a \in I(h) | a \in \mathcal{N}, \text{path}(a, h) = \text{false} \}, I(h) = I(h) \setminus L; \]
   \[ I(h) = I(h) \setminus \{ e \in I(h) | e \in E, \pi_e(a) \in L \ OR \ \pi_e(a) \in L \}; \]
4. return \( Sc \)
Influence on correctness conditions:
External memory variables: None.
Process semantics: None.
Precondition, Postcondition: None.

Operation 2: remove Interval

Input:
- Cyclic business process \( P_1 \) with graph \( G_1(N_1,E_1) \)
- \( I(h) \) interval
- \( R_{ab} \) set of interval state registers

Output: Cyclic business process \( P_2 \) with graph \( G_2(N_2,E_2) \)

Method:
1. \( P_2 \leftarrow P_1, N_2 \leftarrow (N_1 \cup R_{ab}) \setminus \{ a \in N_1 \mid a \in \text{Out}(I(h)) \} ; \)
2. \( \text{if}(R_{ab}) \leftarrow \emptyset ; \text{Out}(R_{ab}) \leftarrow \emptyset ; \) \( \text{out}_{\text{app}}(R_{ab}) \)
3. for all \( a \in \text{In}(I(h)), b \in A, b \notin I(h), \) do 
   - \( \Delta_2(b, A_{ab}) \leftarrow \{ (w, v') \mid v \in \text{in}(a), w \in \text{out}(b), (w, v) \in \Delta_2(b, a) \) and \( v' \) is a new defined variable such that \( \text{Dom}(v') = \text{Dom}(v) \), \( V_2 = V_1 \cup \{ v' \} \); \( \text{In}(A_{ab}) \leftarrow \text{in}(A_{ab}) \cup \{ v' \} \) 
   - \( \Delta_2(A_{ab}, b) \leftarrow \{ (v', w) \mid v \in \text{in}(a), w \in \text{out}(b), (w, v) \in \Delta_2(a, b) \) and \( v' \) is a new defined variable such that \( \text{Dom}(v') = \text{Dom}(v) \cup \{ v' \} \), \( V_2 = V_1 \cup \{ v' \} \); \( \text{Out}(A_{ab}) \leftarrow \text{out}(A_{ab}) \cup \{ v' \} \) 
4. for all \( a \in \text{Out}(I(h)) \) do 
   - \( \Delta_2(A_{ab}, M) \leftarrow \{ (v', w) \mid v \in \text{in}(a), w \in M, (w, v) \in \Delta_2(a, M) \) and \( v' \) is a new defined variable such that \( \text{Dom}(v') = \text{Dom}(v) \cup \{ v' \} \), \( V_2 = V_1 \cup \{ v' \} \); \( \text{Out}(A_{ab}) \leftarrow \text{out}(A_{ab}) \cup \{ v' \} \) 
5. define the set of state variables:
   - \( o^*(A_{ab}) = \{ (v, r) \mid r \in R_{ab} \) and \( v' \) is a new defined variable for the \( r \) such that \( \text{Dom}(v') = \{ 0, 1, 2 \} \), \( V_2 = V_1 \cup \{ v' \}, \) \( \text{in}(A_{ab}) \leftarrow \text{in}(A_{ab}) \cup \{ v' \} \) 
   - \( o^*(A_{ab}) \leftarrow \text{out}(A_{ab}) \cup \{ v' \} \) 
6. \( \Psi(A_{ab}) = \{ a \in N \mid a \in \text{In}(I(h)), \{ e \in E \mid e \in \text{In}(I(h)) \} \}; \) 
7. return \( P_2 \).

Influence on correctness conditions:
External memory variables: Changed domains (step 4)
Process semantics: None. All data connectors are replaced by equivalent connectors.
Precondition: None
Postcondition: None, because all connectors between the cycle activities and the memory are replaced by equivalent connectors and the new variables, which are added instead of the cycle activity output variables, have additional value \( \perp \) for representing the possibility of skipping its activity execution in case of cycle termination without completing it entirely.

Operation 3: construct BranchActCond

Input:
- Cyclic business process \( P_1 \) with graph \( G_1(N_1,E_1) \)
- Activity \( A \) from the \( I(h) \) interval
- \( I(h) \) interval
- \( BR_{ab} \) set of interval branching activities
- \( R_{ab} \) set of interval state registers

Output: \( p^e \) and \( p^o \) conditions

Method:
1. \( A^d = \{ v_r \mid r \in R_{ab} \) is the state register of \( b \in BR_{ab} \) and \( \text{path}(A, b) = \text{true} \) and \( v_r \neq o^*(A_{ab}) \) is a state variable defined for \( r \}' \) 
2. \( A^t = \{ v_r \mid r \in R_{ab} \) is the state register of \( b \in BR_{ab} \) and \( \text{path}(A, b) = \text{true} \) and \( v_r \neq o^*(A_{ab}) \) is a state variable defined for \( r \}' \) 
3. \( r_{o^d} = \{ w \mid \exists b \in \text{In}(I(h)), \exists v \in \text{Dom}(I(h)); \text{path}(b, A) = \text{true} \), \( (w, v) \in \Delta(b, d) \) \} 
4. \( r_{o^t} = \{ w \mid \exists b \in \text{In}(I(h)), \exists v \in \text{Dom}(I(h)); \text{path}(b, A) = \text{true} \), \( (w, v) \in \Delta(b, d) \) \} 
5. define \( p^e = ( \& \& \text{a} = 2 \) & ( \& \& \text{a} = 1 \) & ( \& \& \text{v} = \perp \) \) & ( \& \& \text{v} = \perp \) 
6. define \( p^o = ( \& \& \text{a} = 1 \) & ( \& \& \text{a} = 0 \) & ( \& \& \text{v} = \perp \) & ( \& \& \text{v} = \perp \) 
7. return \( p^e \) and \( p^o \).

Influence on correctness conditions:
External memory variables: None.
Process semantics: None.
Precondition, Postcondition: None.

Operation 4: construct TransitionAndDataCond

Input:
- Cyclic business process \( P_1 \) with graph \( G_1(N_1,E_1) \) (before the interval replacement) 
- Cyclic business process \( P_2 \) with graph \( G_2(N_2,E_2) \) (after the interval replacement) 
- \( I(h) \) interval 
- \( BR_{ab} \) set of interval branching activities 
- \( R_{ab} \) set of interval state registers

Output: Cyclic business process \( P_2 \) with graph process \( G_2(N_2,E_2) \)

Method:
1. \( P_2 \leftarrow P_1, A_{out} \leftarrow \emptyset ; \)
2. for all \( a \in \text{In}(I(h)), b \in (N \setminus \text{In}(I(h))) \cup C, e = a,b, p \Rightarrow E_1, \) do 
   - \( E_2 \leftarrow E_2 \setminus \{ e \} \cup \{ \text{out}_{\text{app}}(b, p) \} \)
   - \( E'_{\text{out}} \leftarrow E'_{\text{out}} \cup \{ e \} \)
3. for all \( a \in E_{\text{in}}, b \in E_{\text{in}} \) do 
   - \( \{ p', p'' \} \leftarrow \text{construct_BranchActCond} \) \( P_{2_a}, BR_{ab}, R_{ab} \)
   - define \( v^* = \{ v' \mid \text{Dom}(v') = \text{Dom}(v), (w, v) \in \text{out}_{\text{app}}(b, p) \} \)
   - define \( p'' = \pi_{\text{e}}(e) \) \( \cup (p' \cup p'') \) with \( \text{if}(p'') = \text{if}(p') \cup v^* \)
   - if \( \pi_{\text{e}}(e) \notin A_{\text{out}} \) then \( A_{\text{out}} = A_{\text{out}} \cup \{ \pi_{\text{e}}(e) \} ; P_{2_a} = p' \) else \( E_2 \leftarrow \text{out}_{\text{app}}(b, p) \)
   - \( C_2 = C_1 \setminus \{ \pi_{\text{e}}(e) \} , E_2 = E_2 \setminus \{ e \} \)
4. for all \( a \in A_{\text{out}}, \) do 
   - \( C_2 = C_2 \cup \{ P_a \} ; E_2 = E_2 \cup \{ \text{out}_{\text{app}}(a, p) \} \)
   - define \( A_{\text{out}}(P_a) = \{ (w, v) \mid (w, v) \in \Delta(A_{\text{out}}), (w, v) \in \text{Dom}(I(h)) \}
   - return \( P_2 \).

Influence on correctness conditions:
External memory variables: None.
Process semantics: None.

All control connectors are replaced by equivalent connectors based on removed connectors and interval conditions of branching activities (operation 3). Interval conditions have to transfer two possible cases of branching activity cycle terminating execution (state register value is 0 or greater then 1).

Precondition, Postcondition: None.

4.2. Transformation Conditions

Conditions are specified below for the transformation of a cyclic BP verification to an acyclic BP verification. A cyclic BP for which the specified conditions are satisfied is called a restricted cyclic BP and is denoted by E-process.

Assume \( A = \{ D \in N \mid \Delta(A,D) \neq \emptyset \} \).

**Definition (E-process)**

E-process is a cyclic BP, which satisfies the following restrictions:

1. The control graph is a first order factor graph [9].
2. Data flow connectors can connect an activity with another activity if they are connected in the control flow, too.
   \( \forall A \in N, \exists \Delta(A,D) \subseteq \{ D \in N \wedge D \notin N \} \wedge \exists \langle A, D, t \rangle > e \} \cup M \)
3. Branching activities, contained in a cycle, can have only two outgoing mutually exclusive transition conditions.
   \( \forall A \in N, G \in K, \langle G, D, t \rangle > e \} \wedge D \notin K \Rightarrow t_1 = 1 \)
4. The memory elements can have single data connectors pointing onto them.
   \[ \forall A \in N, D \in N, <v_1, v'> \in \Delta(A, M), <v_2, v'> \in \Delta(D, M) \]
   \( \Rightarrow A = D. \)

4.3. Transformation algorithm of E-processes

Let us define the transformation algorithm of E-processes by use of busing operations.

**Algorithm R. Cyclic process transformation.**

**Input** E-process \( P_k = <T, N, C, E, \Phi, \Phi_C, \Phi, \Psi, \Delta, M > \) with graph \( G(N, E) \)

**Output** W-process \( P_w = <T', N', C', E', \Phi', \Psi', \Delta', M' > \) with graph \( G(N', E') \)

**Method:**

1. \( N' \leftarrow N; C' \leftarrow C; E' \leftarrow E; M' \leftarrow M; C' \leftarrow C; \Phi' \leftarrow \Phi; P_1 \leftarrow P_k \)
2. \( Sc \leftarrow \text{construct IntervalSet} (P_k) \).
3. for all \( I(h) \in Sc \), do
   a. construct the set of branching activities :
      \( BR_{h(b)} \leftarrow \{ a \in I(h) \mid \exists b \in I(h) \wedge \exists e \in E \wedge s, a, b, t > e \} \).
   b. construct the set of state registers:
      \( R_{h(b)} \leftarrow \{ r_1 \mid r_1 \text{ is the state register of } a \in BR_{h(b)} \}
   c. \( P_2 \leftarrow \text{remove Interval} (P_1, I(h), R_{h(b)}) \).
   d. \( P_3 \leftarrow \text{construct TransitionAndDataCon} (P_1, P_2, I(h), BR_{h(b)} R_{h(b)}) \).
4. Return \( P_3 \).

The following statement holds.

**Statement**

Let \( P \) be a cyclic E-process. It is transformable to the acyclic W-process, called \( Q \), such that
\( \forall P \) ( \( P \) is a E-process), \( \exists Q \) ( \( Q \) is a W-process); \( |Q| \) is correct/incorrect \( \Rightarrow P \) is correct/incorrect.

**Proof**

Let \( P \) be an E-process. Let -\( \alpha \), \( \beta \) be precondition and a postcondition of \( P \), \( M \) be the external memory of \( P \). In this case, \( P \) is correct with respect to \( \alpha \) and \( \beta \), if \( \alpha(M)=1 \Rightarrow \beta(MP)=1 \)

Let us apply the cyclic process transformation algorithm to transform the \( P \) to the acyclic W-process called \( Q \). Seeing that if all transformation operations would not change the logic of process correctness conditions and process semantics then \( Q \) will be equivalent to the initial \( P \) process from a verification standpoint. Let -\( \alpha_i \), \( \beta_i \) be precondition and postcondition of \( Q \), \( M_Q \) be the external memory of \( Q \). The precondition \( (\alpha_i) \) and postcondition \( (\beta_i) \) of the \( Q \) process would be logically equivalent to \( \alpha \) and \( \beta \) correspondingly, such that \( \alpha(M)=1 \Rightarrow \alpha_i(M_Q)=1 \Rightarrow \beta_i(M_Q)=1 \Rightarrow \beta(MP)=1 \)

The statement is proved.

5. THE ALGORITHM APPLICATION

The application of the suggested approach is demonstrated on a special class of business processes – Information Technology Infrastructure Library (ITIL) [10-12] processes. ITIL is a set of process-based best practices for the management of IT services, and serves as an international de facto standard guidance for creating IT management processes. IBM provides Process Reference Model for IT (PRM-IT) [12] that supplements the content of ITIL based on IBM’s extensive IT management experience [10-12] IBM has developed a library with detailed (task-level) process workflows based on PRM-IT. It is also possible to build new processes based on the given processes.

**Incident Management** process is one of the processes from ITIL, which will be used to demonstrate application of the verification algorithm. The accuracy of the algorithm is demonstrated on the example where the algorithm finds an incorrect condition. Because of the limited space, only a fragment of the process is presented in Figure 4.

For the verification of the given process, precondition and postcondition should be specified for the process. The specific conditions are created based on the needs of verification against definite aspects of process behavior.

**PreC = i(1).Incident \# \_ AND**

\( i(1).Incident\_Communication\_to\_User \# \_ AND \)

\( i(2).Problem\_and\_Known\_Errors \# \_ AND \)
This PostC states that for all executions of the process either a workaround or a fix should be provided or it should be created (if not available). $b\text{WorkaroundOrFixProvided}$, $b\text{WorkaroundOrFixCreated}$, $b\text{IncidentResolutionCreated}$ are predicates indicating successful execution of corresponding tasks.

**Operation 1: Construct IntervalSet**
Result: $S_c = \{1, 2, 3, 4, 5\}$.

**Operation 2: Remove Interval**
The found interval is replaced by an equivalent activity (activity 10).

**Operation 3: Construct BranchActCond**
This operation constructs input and output containers for the newly created activity (activity 10).

Input container:

$$i(2).\text{Incident_info} \neq \L AND$$
$$i(6).\text{Problem_and_Known_Errors} \neq \L,$$
where $\L$ denotes unknown value of the variable.

$$PostC = (b\text{WorkaroundOrFixProvided} = \text{TRUE OR}$$
$$b\text{WorkaroundOrFixCreated} = \text{TRUE}) AND$$
$$b\text{IncidentResolutionCreated} = \text{TRUE}$$

Output container:

$$o(10) = <i(1).\text{Incident},$$
$$i(1).\text{Incident_Communication_to_User},$$
$$i(2).\text{Problem_and_Known_Errors},$$
$$i(2).\text{Incident_info} >.$$

where $br(3)$ and $br(4)$ are state registers of branching activities 3 and 4.
To improve the overall quality of the Incident Management process shown in the example in Figure 3, additional task “workaround testing” (activity 7.1) is added to the process (Figure 6). This activity tests if everything is normal after creating a workaround or a fix. In case of an abnormally created workaround or a fix, the activity “notify” (activity 7.2) would notify about it.

![Figure 6: Modified section of the process](image)

New postcondition:

\[
\text{Post}_C = (b\text{WorkaroundOrFixProvided} = \text{TRUE} \; \text{OR} \; b\text{WorkaroundOrFixCreated} = \text{TRUE}) \; \text{AND} \; \text{Tested} = \text{TRUE} \; \text{AND} \; b\text{Passed} = \text{YES} \; \text{AND} \; b\text{IncidentResolutionCreated} = \text{TRUE}
\]

As a result of applying the similar steps of the process transformation and verification, the condition of incorrect processes would be satisfied [6]. The control connector between activities 7.1 and 8 has to be removed to correct the modified template logic. The postcondition has to be also changed to:

\[
\text{Post}_C = (b\text{WorkaroundOrFixProvided} = \text{TRUE} \; \text{OR} \; b\text{WorkaroundOrFixCreated} = \text{TRUE}) \; \text{AND} \; \text{Tested} = \text{TRUE} \; \text{AND} \; ((b\text{Passed} = \text{YES} \; \text{AND} \; b\text{IncidentResolutionCreated} = \text{TRUE}) \; \text{OR} \; (b\text{Passed} = \text{YES} \; \text{AND} \; b\text{Informed} = \text{TRUE})).
\]

Applying the algorithm to the corrected process will result in the satisfaction of the correct process condition [6].

6. CONCLUSION AND FUTURE DIRECTIONS

In future it is planned to continue this work in two directions. One direction is to determine the range of the application of the suggested approach. The first step is its application to all processes from the ITIL library. A tool named IT Management Process Builder is being developed which will allow application of both formal and simulation methods of process verification. At first process is verified by the formal algorithm. If the result is ‘unknown’ or ‘unverifiable’ simulation algorithm is applied. The other direction is to improve the suggested verification algorithm, specifically, to increase the accuracy of the algorithm, to consider processes with fewer restrictions. The final goal is to find out limitations of the proposed approach.

References