Online Supplement

Ensuring Flexible Process Compliance with Semantic Constraints using Mixed-Integer Programming (MIP)

Akhil Kumar[1], Wen Yao[2], Chao-Hsien Chu[2]
[1] Smeal College of Business, Pennsylvania State University, University Park, PA 16802, USA
[2] College of Information Sciences and Technology, Pennsylvania State University, University Park, PA 16802.
{akhil,wxy119,chc4}@psu.edu

A1. Examples

In this section, we use several healthcare scenarios along with the running example in Figure 1 to illustrate the applicability of our approach. As shown in Figure A1, our examples cover the lifecycle of a process from design time until its completion. Scenarios are also briefly described in the figure. We use the CPLEX tool to solve a MIP formulation and check whether a process is compliant with semantic constraints. The CPLEX model has eight parts: the defined *variables*; the *decision variables* for which the model finds a solution; the *objective function* to be maximized or minimized; *process representation constraints*; *consistency* and *transitivity* constraints; and the *model- and instance-level semantic constraints*. Each part of the model has been described above. The CPLEX model for Example 1 is given in the Section A4. We describe the details for each example and analyze the results.

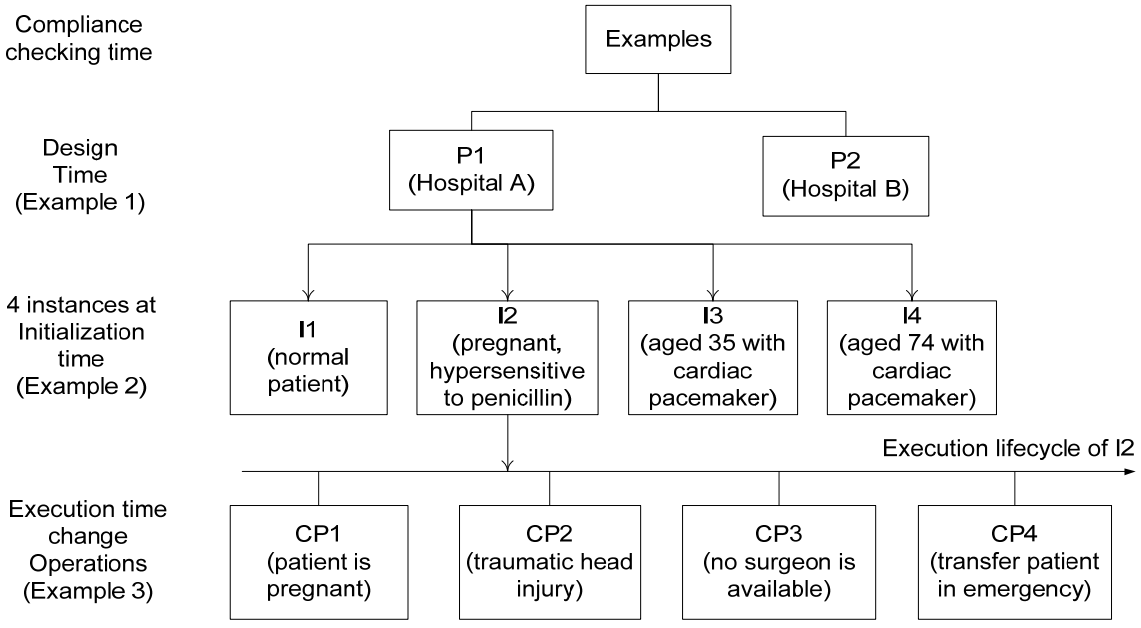


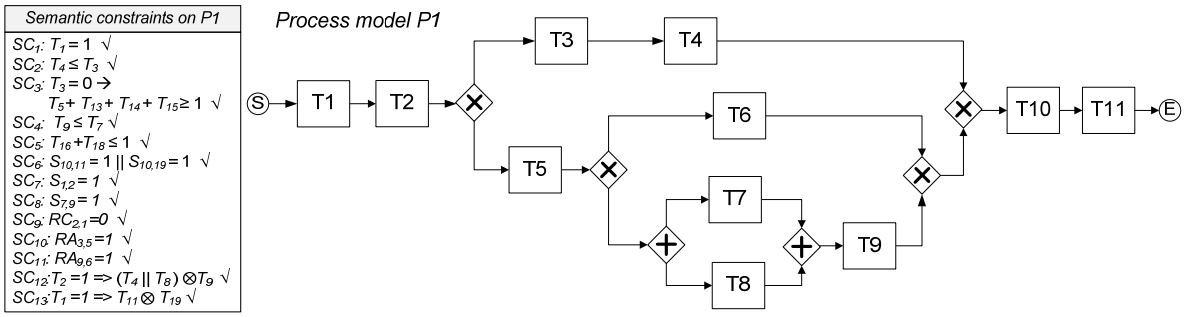
Figure A1: Overview of the Illustrated Examples

Example 1 (Design Time): Suppose the semantic constraints in Table 2 (in the main body of the paper) are enforced for any workflow model of a proximal femoral fracture process. Further, assume that Hospitals A and B have designed their own clinical workflow models P1 and P2 (see Figure A2). Using our algorithm, we can check their compliance against the semantic constraints that should be enforced. The semantic constraints SC_1-SC_{13} have been validated beforehand to ensure that the constraint set is sound and complete (i.e., there are no redundancies or conflicts). The result shows that P1 is compliant since the objective function is 0. Thus the compliance degree $\mathbb{D}=1$. In contrast, P2 is not compliant since the objective function is 3; hence $\mathbb{D}=1-3/13=10/13$. Actually the output of the objective function in this CPLEX model is 4, but the *adjusted obj* is 3 because both $SY_{9,7}=1$ and $SX_{7,9}=1$ results in double counting. Suggestions are also provided to make P2 compliant with SC: $TX_1=1$ (for SC_1), $SY_{9,7}=1$ and $SX_{7,9}=1$ (for SC_8), $TX_{11}=1$ (for SC_{13}).

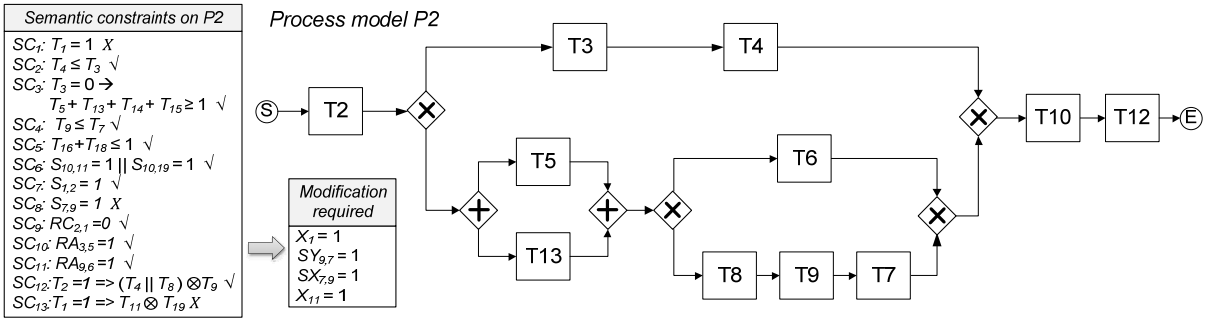
The interpretation of this output from the CPLEX model is that the process modeler should:

add task T_1 (i.e., patient admission)
execute T_7 before T_9 (i.e., surgical planning should be carried out before surgery), and
add T_{11} (i.e., a patient should be discharged)

Thus, specific feedback is given to the user in this manner. This ability to check process compliance in an automatic manner is useful for any non-trivial model. With this approach, we can verify if a process model is compliant or not. If it is not compliant, we can find an optimal solution (if one exists) and suggest compensation operations to restore it. Our algorithm calculates the compliance degrees of different process models for comparison across them.



(a) Process model P1, obj=0, $P1 \vDash SC$, $\mathbb{D}=1$



(b) Process model P2, obj=3, $P2 \not\models SC$, $\mathbb{D}$ =10/13

Figure A2: Compliance Checking at Design Time

Example 2 (Initialization Time): Suppose the four process instances I1 through I4 are initialized from P1 with different case data: I1 (a normal patient), I2 (a pregnant patient who is hypersensitive to penicillin), I3 (a patient aged 35 with cardiac pacemaker), and I4 (a patient aged 74 with cardiac pacemaker). We use Δ_{SC} to denote additional semantic constraints triggered by the applicable ECA rules at initialization time as follows:

- I1 (a normal patient): $\Delta_{SC} = \emptyset$.
- I2 (a pregnant patient hypersensitive to penicillin): $\Delta_{SC} = \{T_5 = 0, T_{17} = 0, T_{18} = 0\}$
- I3 (a patient aged 35 with cardiac pacemaker): $\Delta_{SC} = \{T_{13} = 0\}$
- I4 (a patient aged 74 with cardiac pacemaker): $\Delta_{SC} = \{T_{13} = 0, T_{12} = 1, S_{12,9} = 1\}$

Thus, no compliance checking is needed when I1 is initialized because no additional constraint is triggered, i.e., $\Delta_{SC} = \emptyset$. However, compliance checking is needed for I2, I3, and I4. We apply the *compliance_init* algorithm and obtain solutions as given in Figure A3:

Extra semantic constraints on I2 derived from P1
SC_1: $T_{17} = 0$ √
SC_2: $T_{18} = 0$ √
SC_3: $T_5 = 0$ √

(a) I2: obj =0, $\mathbb{D}$ = 1

Extra semantic constraints on I3 derived from P1
SC_1: $T_{13} = 0$ √

(b) I3: obj=0, $\mathbb{D}$ =1

Extra semantic constraints on I4 derived from P1
SC_1: $T_{13} = 0$ √
SC_2: $T_{12} = 1$ X
SC_3: $S_{12,9} = 1$ X

(c) I4: obj=2, $\mathbb{D}$ =14/16, suggested action: TX_{12}=1, $SX_{12,9}$=1

Figure A3: Compliance Checking at Initialization Time

From the results, I2 and I3 are compliant despite adding the new semantic constraints. Thus, no further action is required. The objective function value for I4 is 2 and suggested actions to make it compliant are provided. However, these actions, before being applied, should be checked

against a combination of the original and new constraints; otherwise, the original constraints might be violated (refer to algorithm *Compliance_init*). The solution for I4 requires that a new task "tolerance test (T_{12})" be inserted and executed before "surgery (T_9)" (see Figure A3 (c)). After modifications are made to I4, we obtain the revised process instances shown in Figure A4.

This example shows the difference between model- and instance-level semantic constraints. The former ones are complied with by all instances derived from the process model; hence, they are checked at design time. In contrast, the latter ones are only applicable to cases that satisfy specific conditions, thus they are only verified at initialization time when case data is available. In general, different cases can have various constraints and they are treated differently.

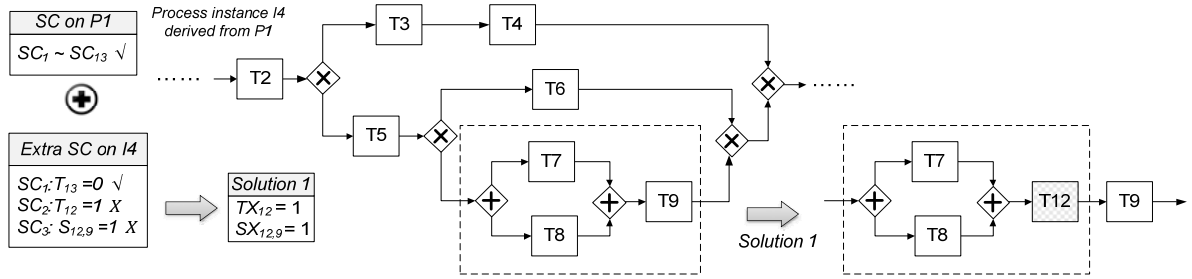


Figure A4: Solution and Modification to I4

Example 3 (Execution time): An instance may go through several changes during its execution lifecycle. After each change set is applied or a task execution occurs, the current context for this process instance will be changed. Thus, each time before a change is issued it should be verified. Consider the execution of instance I2 derived from process model P1. Note that the initial status of I2 is compliant as shown in Example 2. We apply four sets of changes CP1 through CP4 to I2 as shown in Table A1. This table describes the meaning of each change set and the resulting actions generated by our algorithm. The final resulting process is shown in Figure A5.

For change operations that are invalid or partially valid, the user will have the option to adopt the suggested actions or not. In other words, she can use the invalid change operations but the system will log this violation for future analysis. For example, for CP1, the doctor is advised to perform MRI (no radiation) instead of CT scan due to the patient's pregnancy. However, since a MRI test takes much longer, the doctor may want to go ahead with treatment based on symptoms and then make a decision depending upon how the patient responds. The doctor can order a MRI test or a CT scan after medication (i.e., the change operation is: insert an imaging test after T8).

Table A1: Change Sets Occurring during the Execution Lifecycle of Instance I2

Change set	**Explanation**	**Results**
CP1={OP_1}, OP_1 = remove (T_5)	This change set is automatically triggered when I_2 takes the path to T_5 (i.e., this patient is under suspicion for a proximal femoral fracture). This could happen because a pregnant patient must NOT have CT scan.	Solution: obj=1, CP1 is valid with compensation that suggests T_{13} should be inserted (TY_{13} =1). Semantically, it means at least one imaging diagnosis test should be carried out if the patient is suspicious of proximal femoral fracture.
CP2={OP_1, OP_2}, OP_1 =insert (T_{16} before T_8) OP_2 =remove (T_8)	It is issued before T_8 is triggered. This could happen due to a traumatic head injury event. OP_1 aims to insert a task "Administration of Marcumar" to decrease the clotting ability of the blood so that thrombosis is prevented. OP_2 aims to forbid "Administration of Aspirin".	Solution: obj=0, CP2 is valid. Semantically, Marcumar can interact with Aspirin, so they are exclusive to avoid drug interaction (i.e., $T_8 + T_{16} \leq 1$). However, administration of Aspirin is moved (i.e., T_8 =0) due to the head injury event, thus "Administration of Marcumar" is allowed.
CP3={OP_1, OP_2}, OP_1 = removeResource(T_9, R_6); OP_2 = setResource (T_9, R_3)	It is issued when T_7 is completed. It tries to assign a therapist to the surgery instead of a surgeon since no surgeon is available.	Solution: no obj, CP3 is invalid. Semantically, it means performing a surgery requires a surgeon.
CP4={OP_1, OP_2}, OP_1=delete (T_{11}), OP_2=insert (T_{19})	Issued when task T_{11} is running. It replaces the discharge task with transfer task because the patient will be transported to another hospital due to emergency.	Solution: obj=2, CP4 is allowed with compensation that suggests T_{11} be removed and T_{19} be inserted as a secondary obligation. See Figure A5.

This example shows that a process instance may encounter a number of changes during its execution lifecycle. After each change, the structure of this instance may change and thus impact subsequent changes. Again it is difficult, if not impossible, for human beings to memorize such frequent changes. Hence, medical errors can occur easily in the absence of automatic checking.

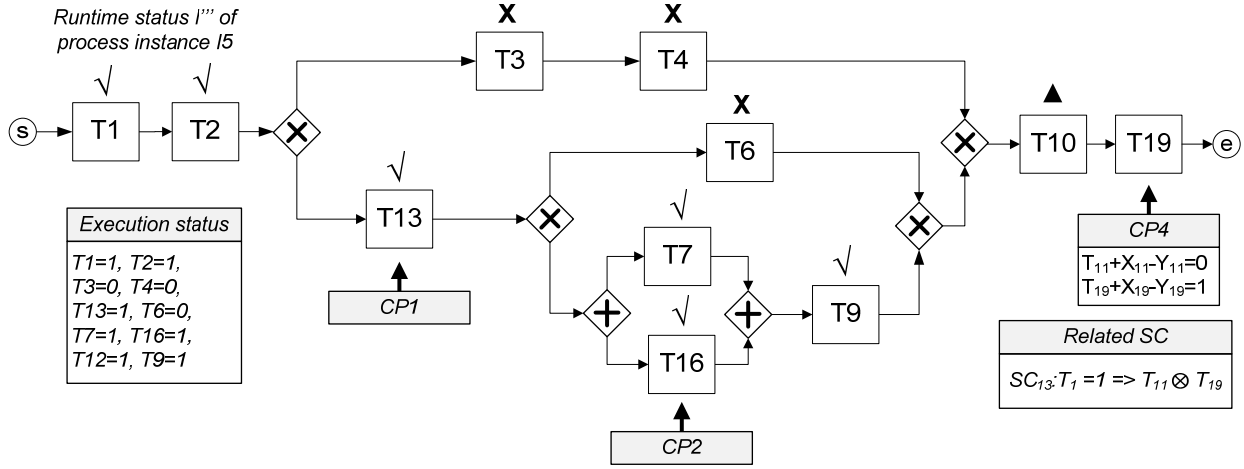


Figure A5: Compliance Checking during the Lifecycle - After CP4 is applied

A.2. Comparison of MIP- vs. Logic-based Approach

The MIP-based approach for expressing constraints for process compliance has considerable appeal in comparison with a logic based one, such as first order predicate logic (Huth and Ryan, 2004). We compare these two approaches according to a variety of criteria in Table A2. First, logic does not allow us to formulate a problem in terms of a penalty function as an objective, and find an optimal solution to ensure weak compliance as one can do with the MIP method. Second, it is very difficult to express m-of-n constraints in logic but not so in our approach as we showed in Section 3. Thirdly, commutative and transitive relationships can be expressed more simply and elegantly in the proposed formalism as compared with logic. Moreover, MIP formulations have standard representation formats, and can be solved very efficiently using a variety of proprietary and open source tools. Logic rules can be represented in various formats that are not so easily interchangeable. Of course, MIP formulations have limitations in terms of expressive power that can be obtained in higher-order logics.

This comparison simply shows that for our problem at hand, the MIP based approach is particularly appealing. It is not our intention to argue that one approach is superior than the other because it is not possible to generalize. More comparison and discussion of these approaches and hybrid approaches that combine logic constraints with IP style ones can be found in (Hähnle, 1994; Hooker and Osorio, 1999).

Table A2: Comparison of MIP-based and Logic-based Compliance Checking

Criteria	**MIP-based Approach**	**Logic-based Approach**
Penalty function as an objective	Easy to formulate	Does not lend itself easily
Optimal solution	Determines minimum changes needed to restore compliance	Cannot optimize easily
Expressing m-of-n relationships	Simple	Messy
Expressing constraint properties (e.g., commutativity, transitivity)	As an algebraic expression	As logic predicates
Support of mature tools	Efficient tools like CPLEX based on standards	Large number of non-standard tools
Language issues	MIP formulations are quite standard	Many languages for expressing rules
Higher order logics	Not supported	More expressive, but harder to implement

Specifically, in contrast to a LTL-based approach (Awad et al., 2009; Awad et al., 2011), we do not have the notion of states. From a complexity point of view the number of states is exponential in the number of tasks and the complexity of the solution is exponential in the

number of states. Thus, roughly speaking there is a double exponential complexity (in the number of tasks) in such an approach versus the single exponential complexity in our approach. Hence there is a natural tradeoff between expressive power and complexity. We use an existing well known solution technique to provide reasonable expressive power, while the other approach proposes a new framework for which a solution engine is not readily available.

A3. Implementation Architecture

Figure A6 presents an implementation architecture that integrates the specification of semantic constraints and compliance checking of processes into an APMS. Generally, pre-designed process models are stored in the process repository and tasks are stored in the task repository (omitted in the figure). The MIP adapter analyzes the process models and converts them into MIP models in CPLEX. As shown in the paper, the control-flow structure can be converted into MIP formulas easily. Resource assignments pertaining to each task should be converted into MIP formulas as well. The MIP adapter is responsible for synchronizing any changes made to process models. Each process model represented in a MIP formulation also includes a complete and sound set of semantic constraints. The semantic constraint designer is a GUI-based tool that allows knowledge engineers to design semantic constraints using our specification language in Table 1 (in the main body of the paper). We propose that a visual representation of semantic constraints be available to knowledge engineers to assist their design. The validated semantic constraints are stored in the repository that can be shared by different users, departments, or even organizations. As domain knowledge changes, the repository is updated.

The MIP solver is responsible for checking the compliance of process models and instances by solving MIP formulations. Once a process model is inserted (or updated) into the repository, the MIP model is added or updated as well. Then the MIP solver will look for its associated semantic constraints in the repository. If such semantic constraints are found, they are added to the MIP model and trigger the algorithm *Compliance_design*. Depending on the resulting objective function, the user is notified whether the process model is compliant or not (with suggested compensation operations). If the semantic constraint set pertaining to any process model is updated, the design time compliance checking will be repeated. The CPLEX tool is the core module used to solve MIP models by the MIP solver.

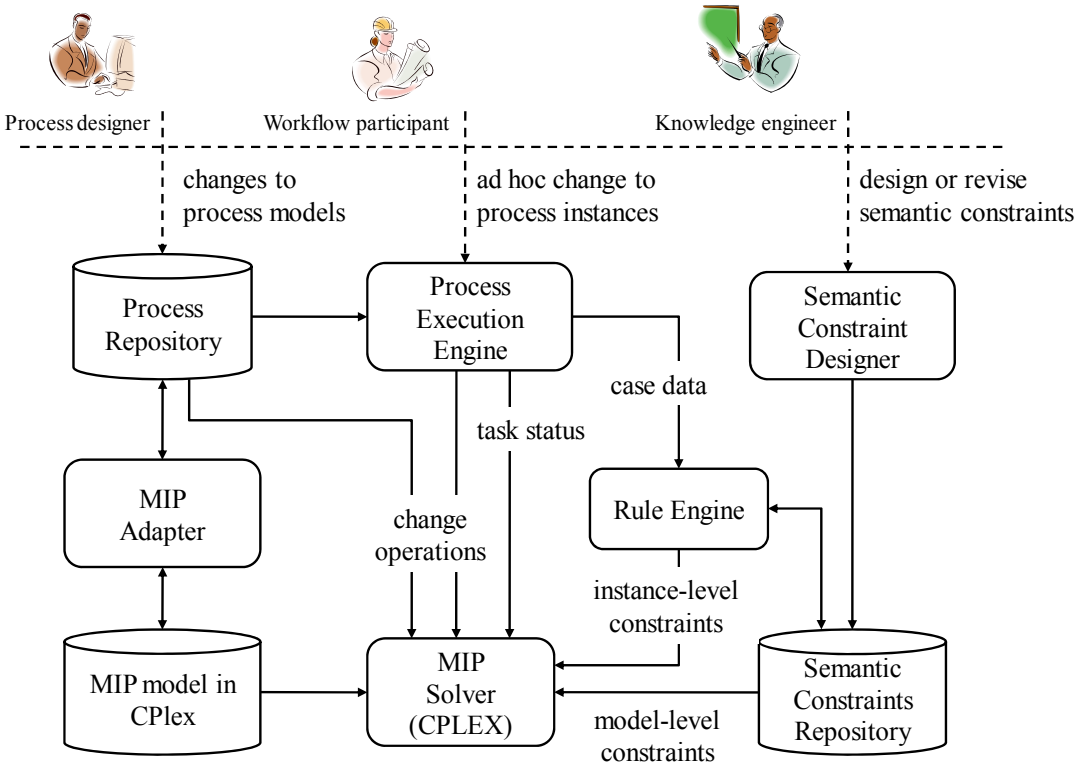


Figure A6: An Overall Architecture for Implementation

When a process model is initialized, case data is entered by workflow participants and passed from the process execution engine to the rule engine. If their conditions are met, then ECA rules are triggered and they may produce instance-level semantic constraints, e.g. if the patient is more than 70 years old, then *a tolerance test must be done prior to the surgery*. The MIP solver checks the compliance of the initialized instance by running the *Compliance_init* algorithm that takes into consideration both model-level and instance-level constraints. Compensation operations are suggested to users if the instance is not compliant. During execution, the process execution engine maintains the runtime status of all running instances and sends their task states back to the MIP solver. This information is maintained and used for checking the validity of ad hoc changes during instance execution, according to the *Compliance_runtime* algorithm. This is helpful for workflow models that involve a variety of participants and are prone to exceptional events. This system should give users the right to decide whether to apply non-compliant changes. If so, such exceptions should be logged in the system and analyzed for future use. For example, if certain process violation occurs frequently, the process designer should consider updating the corresponding process model by incorporating the repeated violation.

A.4 Full CPLEX Model for Example 1

```
/*********************************************
 * OPL 12.3 Model
 * Author: axk41_admin
 * Creation Date: Aug 27, 2012 at 4:24:31 PM
 *********************************************/
/*********************************************
 * OPL 12.2 Model
 * Author: Wen Yao
 * Description: CPLEX model for Process P1
 * Creation Date: Nov 16, 2010 at 11:15:25 AM
 *********************************************/

 //***************data************************
int NbTask = 19; //total number of task in this process model
range taskID = 1..NbTask; //task ID
int start = 1; //starting task
int end = 1; //ending task
int NbResource = 7; //total number of resources in this process model
range resourceID = 1..NbResource; //resource ID

  //***************Define variables*******************
dvar int Task[taskID] in 0..1; //variable for defining task presence
dvar int T_new[taskID] in 0..1; //variable for defining task presence
dvar int S_new[taskID][taskID] in 0..1; //variable for defining sequence
dvar int RA_new[taskID][resourceID] in 0..1; //variable for defining task
assignment

 //***************Define decision variables*******************
dvar int TX[taskID] in 0..1; //variable for task presence relaxation
dvar int TY[taskID] in 0..1; // variable for task presence relaxation
dvar int SX[taskID][taskID] in 0..1; //decision variable for task sequence
dvar int SY[taskID][taskID] in 0..1; //decision variable for task sequence
dvar int RX[taskID][resourceID] in 0..1; //decision variable for task
assignment
dvar int RY[taskID][resourceID] in 0..1; //decision variable for task
assignment

{int} TAbsenceIndex = {12,13,14,15,16,17,18,19}; //absent tasks

//define matrix indicating current sequence relationships among tasks
int S_initial [taskID][taskID] =
                [[0,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0], //S(1,2)=1, ...
                  [0,0,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0], //S(2,3)=1, ...
                  [0,0,0,1,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0], //S(3,4)=1, ...
                  [0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0], //S(4,10)=1, ...
                  [0,0,0,0,0,1,1,1,1,1,1,0,0,0,0,0,0,0,0], //S(5,6)=1, ...
                  [0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0], //S(6,10)=1, ...
                  [0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0], //S(7,9)=1,...
                  [0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0], //S(8,9)=1,...
                  [0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0], //S(9,10)=1,...
                  [0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0], //S(10,11)=1,...
                  [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                  [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
```

```
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]];

 int RA_initial [taskID][resourceID] =
                    [[1,0,0,0,0,0,0],
                     [0,1,0,0,0,0,0],
                     [0,1,0,0,0,0,0],
                     [0,0,1,0,0,0,0],
                     [0,0,0,1,0,0,0],
                     [0,0,1,0,0,0,0],
                     [0,0,0,0,1,1,0],
                     [0,0,0,0,1,0,0],
                     [0,0,0,0,0,1,1],
                     [1,0,0,0,0,0,0],
                     [1,0,0,0,0,0,0],
                     [0,0,0,0,0,0,0],
                     [0,0,0,0,0,0,0],
                     [0,0,0,0,0,0,0],
                     [0,0,0,0,0,0,0],
                     [0,0,0,0,0,0,0],
                     [0,0,0,0,0,0,0],
                     [0,0,0,0,0,0,0],
                     [0,0,0,0,0,0,0]];


 //***************Define objective function************************
minimize sum(i in taskID) (TX[i]+TY[i])
       + sum(i, j in taskID) (SX[i][j]+SY[i][j])
       + sum(i in taskID, j in resourceID) (RX[i][j]+RY[i][j]);

subject to {

 //***************Define process presentation constraints
      //***************task presence and dependency constraints**********

   Task[7]+TX[7]-TY[7]==Task[8]+TX[8]-TY[8];
   Task[8]+TX[8]-TY[8]==Task[9]+TX[9]-TY[9];
   Task[5]+TX[5]-TY[5]==Task[6]+TX[6]-TY[6]+Task[7]+TX[7]-TY[7];
   Task[3]+TX[3]-TY[3]==Task[4]+TX[4]-TY[4];
   Task[2]+TX[2]-TY[2]==Task[3]+TX[3]-TY[3]+Task[5]+TX[5]-TY[5];
   Task[1]+TX[1]-TY[1]==Task[2]+TX[2]-TY[2];
   Task[10]+TX[10]-TY[10]==Task[11]+TX[11]-TY[11];
   Task[1]+TX[1]-TY[1]==start;
   Task[11]+TX[11]-TY[11]==end;

   forall (i in taskID) //new tasks
            T_new[i] == Task[i] +TX[i]-TY[i];


   T_new[7]==T_new[8];
   T_new[8]==T_new[9];
   T_new[5]==T_new[6] + T_new[7];
```

```
   T_new[3]==T_new[4];
   T_new[2]==T_new[3]+T_new[5];
   T_new[1]==T_new[2];
   T_new[10]==T_new[11];
   T_new[1]==start;
   T_new[11]==end;

   forall (i in TAbsenceIndex) //tasks that will not executed
            Task[i]+TX[i]-TY[i]==0;

   //***************sequence constraints**************
   forall (i, j in taskID)
          S_new[i][j]== S_initial[i][j]- SX[i][j] + SY[i][j] ;

   //***************resource assignment constraints**************
   forall (i in taskID, j in resourceID)
            RA_new[i][j]==RA_initial[i][j]- RX[i][j]+ RY[i][j];

   //***************define consistency constraints**************
   forall (i in taskID)
            0<= Task[i]- TX[i]+TY[i] <=1;

      forall (i in taskID, j in taskID)
      {
            0<= S_new[i][j] <=1;
            S_initial[i][j]+S_initial[j][i]<=1;
            S_new[i][j]+S_new[j][i]<=1;
      }

      forall (i in taskID, j in resourceID)
            0<= RA_new[i][j] <=1;


 //***************Define transitivity constraints**************
      forall (i, j, k in taskID)
            S_new[i][j]==1 && S_new[j][k]==1 => S_new[i][k]==1;

 //************Define semantic constraints (not relaxable)************
    T_new[1]==1; //SC1
    T_new[4] <= T_new[3]; //SC2
    T_new[3]==0 => T_new[5]+T_new[13]+T_new[14]+T_new[15]>=1; //SC3
    T_new[9] <= T_new[7];  //SC4
    T_new[16] + T_new[18] <= 1;  //SC5
    (S_new[11][10] ==1 => S_new[10][11]==1) || (S_new[19][10]==1 =>
S_new[10][19]==1); //SC6

    S_new[2][1]==1 => S_new[1][2]==1;  //SC7
    S_new[9][7]==1 => S_new[7][9]==1;  //SC8

    RA_new[2][1]!=1; //SC9
    RA_new [5][4]==1; //SC10
    RA_new [9][6]==1; //SC11

    T_new[2]==1 && (T_new[4]==0 && T_new[6]==0)=> T_new[9]==1; //SC12
    T_new[1]==1 => T_new[11]==1;   //SC13
    (T_new[1]==1)&&(T_new[11]==0) => T_new[19]==1;  //SC13
   }
```

References

Awad, A., Weidlich, M., Weske, M. 2009. Specification, verification and explanation of violation for data aware compliance rules. eds. *ICSOC-ServiceWave*.

Awad, A., Weidlich, M., Weske, M. 2011. Visually specifying compliance rules and explaining their violations for business processes. *Journal of Visual Languages & Computing*. **22**(1) 30-55.

Hähnle, R. 1994. Many-valued logic and mixed integer programming. *Annals of Mathematics and Artificial Intelligence*. **12**(3) 231–263.

Hooker, J.N., Osorio, M.A. 1999. Mixed logical-linear programming. *Discrete Applied Mathematics*. **96-97**(1) 395–442.

Huth, M., Ryan, M. 2004. *Logic in computer science*. Cambridge University Press Cambridge, New York.