Improving Inconsistency Resolution with Side-Effect Evaluation and Costs

Jochen M. Küster and Ksenia Ryndina

IBM Zurich Research Laboratory, Säumerstr. 4
8803 Rüschlikon, Switzerland
{jku,ryn}@zurich.ibm.com

Abstract. Consistency management is a major requirement in software engineering. Although this problem has attracted significant attention in the literature, support for inconsistency resolution is still not standard for modeling tools. In this paper, we introduce explicit side-effect expressions for each inconsistency resolution and costs for each inconsistency type. This allows a fine-grained evaluation of each possible inconsistency resolution for a particular inconsistent model. We further show how an inconsistency resolution module for a modeling tool can be designed and implemented based on our approach. We demonstrate the applicability of our approach for resolution of inconsistencies between object life cycles and process models.

1 Introduction

Consistency management is a major requirement in software engineering [6]. It requires one to establish consistency constraints that can be checked to identify inconsistencies in models [5,11]. For resolving a particular inconsistency, it is common practice to specify one or more suitable inconsistency resolutions that can transform the model so that the consistency constraint is no longer violated.

Although many solutions addressing various aspects of inconsistency management have been proposed [7,13,20], most modeling tools currently do not offer adequate support to the user for resolution of inconsistencies, in particular for behavioral models such as activity diagrams, statecharts and sequence diagrams. One reason for this is that inconsistency resolution for these models requires transformations that often have side-effects. Such side-effects include both, introduction of new inconsistencies and expiration of existing inconsistencies [12]. As a consequence, in the presence of numerous inconsistencies in a model, many alternative resolutions can be applicable and it is often not obvious which resolution is most appropriate to apply. One technique that has been proposed to tackle this problem is the detection of potential dependencies between inconsistency resolutions using dependency analysis [12,13,22]. This analysis is performed without taking a particular inconsistent model into account. However, only some of the discovered dependencies are usually relevant for a given inconsistent model and these must be precisely identified for comparing alternative resolutions.

G. Engels et al. (Eds.): MoDELS 2007, LNCS 4735, pp. 136–150, 2007.

In this paper, we propose an approach where inconsistency resolutions are associated with explicit side-effect expressions. Such side-effect expressions are evaluated given a concrete inconsistent model to determine whether or not a resolution will have side-effects. This leads to precise knowledge of both expired and induced inconsistencies before applying a particular resolution. Further, we introduce the concept of costs for inconsistency types that allows one to prioritize resolution of different inconsistencies and calculate the total inconsistency cost for a given model. In combination with side-effect expressions, cost reductions for a resolution can be calculated in advance. Overall, our approach leads to an improved way of inconsistency resolution, because it allows a fine-grained comparison of alternative resolutions. In addition, explicit side-effects are also used to avoid re-checking the whole model for inconsistencies after applying a resolution.

We demonstrate our approach using a case study that deals with inconsistency of object life cycles and process models in the context of IBM Insurance Application Architecture (IAA) [1]. For showing the feasibility, we briefly present a design of an inconsistency resolution module based on the proposed approach. Using this design, we have implemented a prototype extension to IBM WebSphere Business Modeler [2] for resolving inconsistencies between object life cycles and process models.

The paper is structured as follows: Section 2 presents our case study by introducing object life cycles, process models and inconsistencies that can occur between these models. In Sect.3, we introduce the concept of explicit resolution side-effects and costs for inconsistency types. Design and implementation of tool support for inconsistency resolution are discussed in Sect.4. We compare our approach with existing work in Sect.5 and conclude the paper in Sect.6.

2 Inconsistency of Object Life Cycles and Process Models

In this section, we first introduce object life cycles and process models, together with an example inspired by IAA. We then discuss inconsistencies that can occur in these models.

An object life cycle [4,9] captures all possible *states* and *state transitions* for a particular *object type* and can be represented as a state machine. Figure 1(a) uses the UML2 State Machine notation [3] for modeling a life cycle for objects of type Claim in an insurance company. The object life cycle shows that after a claim has been registered, it goes through an evaluation process and can be either granted or rejected. Rejected claims are closed directly, while granted claims are first settled and then closed. According to this model, every claim is created in state Registered and passes through state Closed to the only final state.

Our case study deals with *reference object life cycles* that are prescriptive models capturing how objects should be evolved by business processes. Consistency with such reference object life cycles may be an internal business policy or an external legal requirement.

A process model captures the coordination of individual actions in a particular process and the exchange of objects between these actions. Figure 1(b) shows

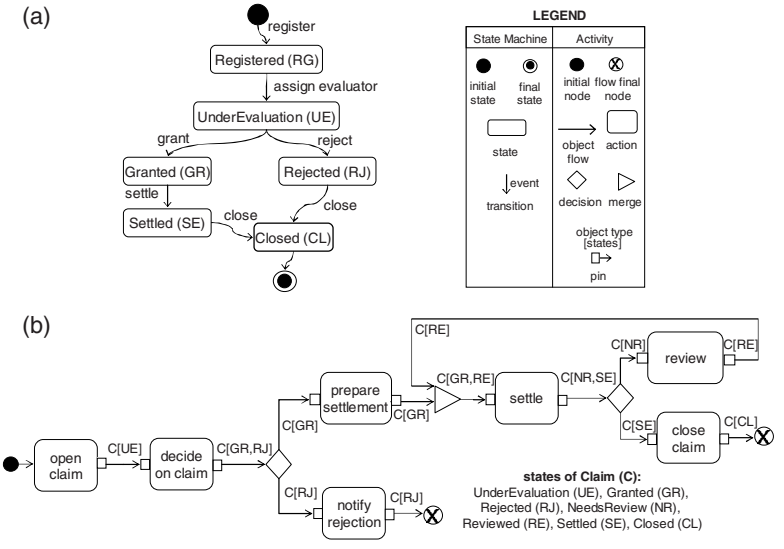

Fig. 1. (a) Reference object life cycle for claims (b) Claim handling process model

a process model for a simplified Claim handling process in the UML2 Activity Diagram notation [3]. In the beginning of this process, a claim is opened and then an evaluation decision is made. If the claim is granted, a settlement is prepared for it. During settlement of a claim, the claimant may appeal for a review of the settlement amount and conditions, in which case these are reviewed and the adjusted settlement is then carried out. Only settled claims are closed in this process.

A process model can also depict how an object changes state as it moves through the process, which is explained in detail in [17]. In our example, the open claim action creates a claim in state UE (UnderEvaluation) and passes it to action decide on claim that changes the claim's state to either GR (Granted) or RJ (Rejected). The following decision node passes claims in state GR to prepare settlement and those in state RJ to notify rejection.

In order to define consistency of a given process model with respect to an object life cycle, we first consider the relation between these models in the UML2 metamodel. Figure 2 shows an extract from the UML2 metamodel that contains relevant classes for State Machine and Activity modeling. The inState association of a Pin to a State makes it possible to specify input and output object states for ActivityNodes in an Activity. We assume that control nodes also have pins that are not explicitly shown in Fig.1(b). For a more convenient means of referring to input and output object states of ActivityNodes, we define additional inputstate() and outputstate() operations, shown as Object Constraint Language [10] definitions at the bottom of Fig.2.

We assume that when executing a process model, state transitions of objects are induced. We also identify states in which the process creates objects and states that objects can be in upon termination of the process:

Definition 1 (Induced transition, first state and last state). *Let a process model P (instance of Activity) and an object type o (instance of Class) be given.*

- *An* induced transition *of o in P is a triple (a, s_{src}, s_{tgt}) such that there is an Action a in P with $s_{src} \in a.inputstate(o)$ and $s_{tgt} \in a.outputstate(o)$;*
- *A state s_{first} is a* first state *of o in P such that $s_{first} \in a.outputstate(o)$ for some Action a that has no input pins of type o;*
- *A state s_{last} is a* last state *of o in P such that $s_{last} \in n.inputstate(o)$ for some ActivityNode n that has no output pins of type o.*

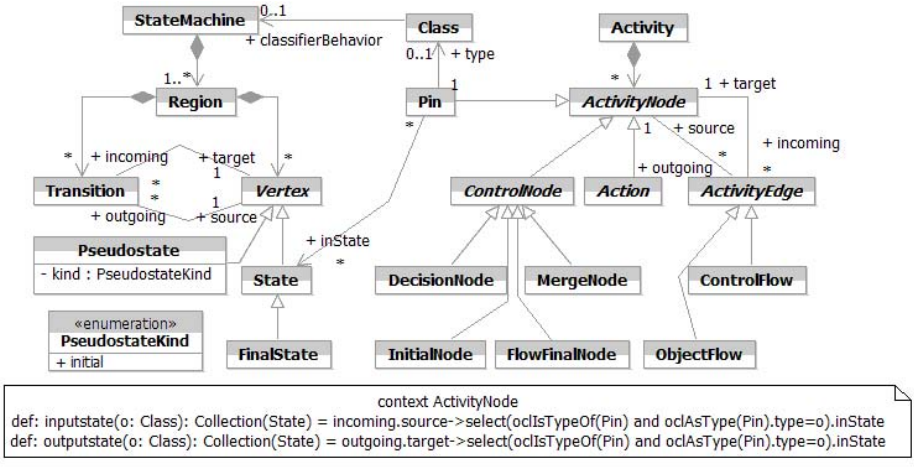

Fig. 2. UML2 metamodel extract

For consistency of process models and object life cycles, we distinguish between *conformance*[1] and *coverage* [17]: *Conformance* requires that a given process model does not manipulate objects in a way that is not defined in the given life cycle. *Coverage* requires that objects used in the process model cover the entire scope of their life cycle. In the following, we directly define inconsistency types:

Definition 2 (Inconsistency types). *Given an object life cycle OLC (instance of State Machine) for object type o (instance of Class) and a process model P (instance of Activity), we define the following inconsistency types:*

- non-conformant transition: *an induced transition (a, s_{src}, s_{tgt}) of o in P, such that OLC contains no transition from state s_{src} to state s_{tgt};*
- non-conformant first state: *a first state s_{first} of o in P, such that OLC contains no transition from the initial state to state s_{first};*
- non-conformant last state: *a last state s_{last} of o in P, such that OLC contains no transition from s_{last} to a final state;*

[1] Called compliance in [17].

- non-covered transition: *a transition* (s_{src}, s_{tgt}) *in OLC where* s_{src} *is not the initial state, such that there is no induced transition* (a, s_{src}, s_{tgt}) *of* o *in* P *for any action* a*;*
- non-covered initial state: *a state* s_i *in OLC that has an incoming transition from the initial state, such that* s_i *is not a first state of* o *in* P*;*
- non-covered final state: *a state* s_f *in OLC that has an outgoing transition to a final state, such that* s_f *is not a last state of* o *in* P*.*

We discover the following inconsistencies in the example claim handling process with respect to the reference life cycle for claims (Fig.1): non-conformant transitions (settle, GR, NR), (settle, RE, SE), (review, NR, RE), a non-conformant first state UE, a non-conformant last state RJ, a non-covered transition (RJ, CL), and a non-covered initial state RG. Two inconsistencies of the same type are distinguished by their *contexts* that comprise model elements contributing to the inconsistency. For example, the context of the non-conformant transition (settle, GR, NR) comprises action settle, and states GR and NR.

3 Inconsistency Resolution with Side-Effects and Costs

In this section we first identify requirements that a solution for inconsistency resolution needs to satisfy, motivated by the case study. We then introduce our proposed solution and explain how it addresses these requirements.

Given a set of inconsistencies such as those discovered for the claims handling example in Sect.2, it is unclear which inconsistency should be resolved first. If our goal is to achieve conformance with the reference claim life cycle, but not necessarily full coverage of it, non-conformance inconsistencies will be of higher priority for the resolution than non-coverage inconsistencies. The first requirement is therefore that **priorities of different inconsistency types must be made explicit in the resolution process (Req1)**.

In order to support the user in resolving a particular inconsistency, we define a number of alternative resolutions for each inconsistency type, following existing work [21,22]. In this paper, we selected three resolutions for resolving a non-conformant transition (an induced transition (a, s_{src}, s_{tgt}) of o in P, such that OLC contains no transition from state s_{src} to state s_{tgt}). These are informally specified in Fig.3: (a) r_1 removes state s_{src} from $a.inputstate(o)$, (b) r_2 removes s_{tgt} from $a.outputstate(o)$ and (c) r_3 removes the entire action a from the process model. We assume that removing the last input or output state also involves removing the associated pin.

Suppose that from the set of discovered inconsistencies for the claims handling process, we choose to first resolve the non-conformant transitions (settle, GR, NR) and (review, NR, RE). This gives rise to three alternative resolutions for each inconsistency, shown in Fig.4. To assist the user in choosing how to resolve a particular inconsistency, **advantages and disadvantages of each available resolution should be identified and used to rank the resolutions (Req2).**

Some resolutions may have *side-effects*, in other words an application of a resolution may not only resolve its target inconsistency, but also introduce new incon-

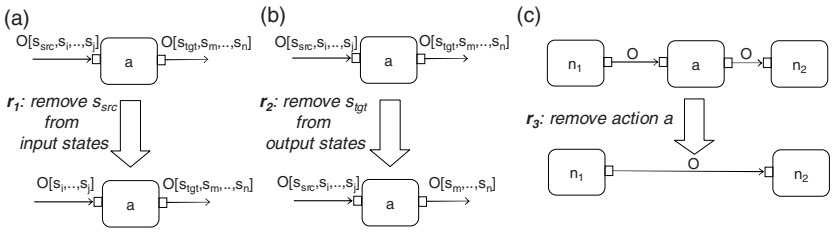

Fig. 3. Resolutions for a non-conformant transition

sistencies (*induced inconsistencies*) or remove other existing inconsistencies (*expired inconsistencies*) [12]. For example, if we apply r_1 to the non-conformant transition (settle, GR, NR) as shown in Fig.4 (b), we will introduce a new non-covered transition (GR, SE). Applying r_1 to the non-conformant transition (review, NR, RE) introduces a new non-covered first state RE, see Fig.4 (e). This shows that applying the same resolution to different inconsistencies may yield different side-effects. Generally, the user will become aware of the side-effects of a resolution only after applying it. To improve this situation, **side-effects of a resolution must be calculated before the resolution is applied (Req3)**.

In the following, we explain our approach to inconsistency resolution that satisfies the identified requirements.

Let $M = \{me_1, ..., me_n\}$ be a model comprising model elements $me_1, ..., me_n$. We denote the set of inconsistency types defined for M as T_M, where each inconsistency type $t \in T_M$ is associated with a set of resolutions R_t. In our example, the claims handling process together with the claim life cycle form the

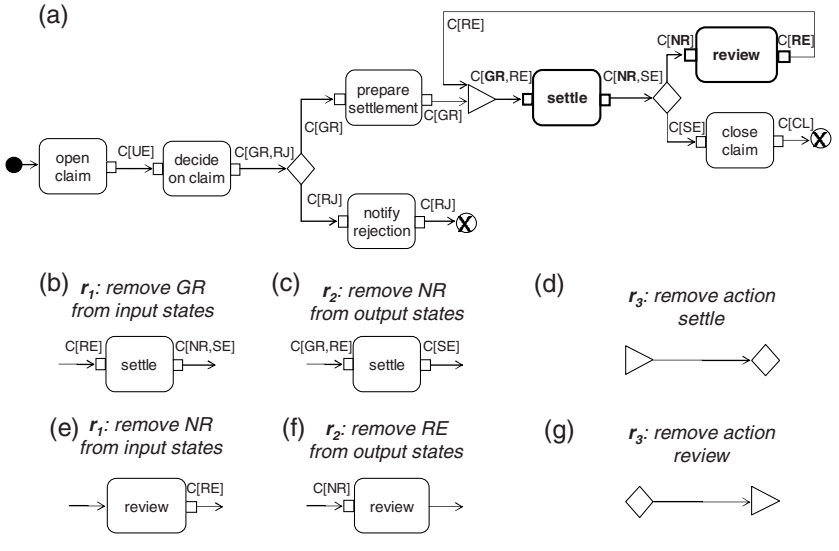

Fig. 4. Resolving non-conformant transitions in the example

model M. The set of inconsistency types is $T_M = \{$ncnf_tran, ncnf_first, ncnf_last, ncov_tran, ncov_init, ncov_fin$\}$ and comprises non-conformant transition, first state, and last state, non-covered transition, initial state and final state inconsistency types, respectively. For non-conformant transitions, the resolution set is $R_{ncnf\_tran} = \{r_1, r_2, r_3\}$.

We denote the set of inconsistencies in M as I_M. Each inconsistency $i \in I_M$ has a type $t \in T_M$ and a context $\{me_j, ...me_k\} \subseteq M$ of model elements that contribute to this inconsistency, hence we write $i = t(me_j, ..., me_k)$. In the example, the set of inconsistencies is $I_M = \{$ ncnf_tran(settle, GR, NR), ncnf_tran(review, NR, RE), ncnf_first(UE), ncnf_last(RJ), ncov_tran(RJ, CL), ncov_init(RG)$\}$.

We introduce *costs* for different inconsistency types to reflect that resolving inconsistencies of some types has a higher priority than of others (Req1).

Definition 3 (Cost of an inconsistency type). *The* cost of an inconsistency type *is defined by a function cost* $: T_M \to \mathbb{N}$ *that maps an inconsistency type to a natural number.*

Costs can either be assigned to inconsistency types once and then used for inconsistency resolution in every model, or different costs for each model can be assigned to reflect a specific resolution goal. For our example, we assume that our main goal is to achieve conformance of the claim handling process. We further consider that conformance of transitions and last states is more important than that of first states. Therefore, we assign the following costs to the different inconsistency types: cost(ncnf_tran) = 3, cost(ncnf_first) = 2, cost(ncnf_last) = 3, cost(ncov_tran) = 1, cost(ncov_init) = 1 and cost(ncov_fin) = 1.

We further associate each resolution with one or more *side-effects*, each having an explicit *side-effect expression* that can be evaluated given an inconsistency in a concrete model. Evaluating side-effect expressions allows us to calculate side-effects of each resolution before they are applied (Req3).

Definition 4 (Side-effect and side-effect expression). *A resolution* $r \in R_t$ *for a given inconsistency type* $t \in T_M$ *is associated with a set of* side-effects $E_r = {E_r}^- \cup {E_r}^+$*, where* ${E_r}^-$ *are side-effects that expire existing inconsistencies and* ${E_r}^+$ *are side-effects that induce new inconsistencies. Each side-effect affects inconsistencies of one type, defined by function type* $: E_r \to T_M$*. A side-effect* $e \in E_r$ *is associated with a* side-effect expression $exp_e : I_M \to \mathcal{P}(I_{M'})$*, where* M' *denotes the model that would be obtained by applying* r *to* M*.*

In Table 1, we define side-effect expressions for resolutions r_1, r_2 and r_3, which were identified by manually analyzing each resolution. We currently specify expressions informally, although this could also be done using first-order logic or the Object Constraint Language [10]. Applying r_1 involves removing s_{src} from $a.inputstate(o)$, which may resolve non-conformant transitions that are induced by action a other than the target non-conformant transition (a, s_{src}, s_{tgt}). This means that r_1 can expire existing non-conformant transitions, as defined in exp_{e_1}.

Furthermore, some induced transitions that provide coverage for a transition in the life cycle may no longer be induced after r_1 is applied. To capture this in

a side-effect expression, we introduce the concept of a coverage set: A *coverage set* of a transition (s_k, s_l) in the object life cycle contains all induced transitions of the form (a', s_k, s_l) in the process model. If an induced transition (a', s_k, s_l) is the only element of a coverage set for (s_k, s_l), then removing this induced transition will introduce a new non-covered transition (s_k, s_l). For r_1, induced non-covered transitions are defined in exp_{e_2}.

Table 1. Side-effect expressions, where $i = ncnf\_tran(a, s_{src}, s_{tgt})$

Res	Side-effect	Side-effect expression
r_1	$e_1 \in E_{r_1}{}^-$	$exp_{e_1}(i) = \{ncnf\_tran(a, s_{src}, s_i) \mid s_i \in a.outputstate(o)$ and $ncnf\_tran(a, s_{src}, s_i) \in I_M\}$
	$e_2 \in E_{r_1}{}^+$	$exp_{e_2}(i) = \{ncov\_tran(s_{src}, s_i) \mid (a, s_{src}, s_i)$ is an induced transition of o in P and (a, s_{src}, s_i) is the only element in the coverage set of $(s_{src}, s_i)\}$
	$e_3 \in E_{r_1}{}^+$	$exp_{e_3}(i) = \{ncnf\_first(s_i) \mid s_i \in a.outputstate(o)$ and s_{src} is the only state in $a.inputstate(o)$ and there is no transition from the initial state to s_i in $OLC\}$
r_2	$e_4 \in E_{r_1}{}^-$	$exp_{e_4}(i) = \{ncnf\_tran(a, s_i, s_{tgt}) \mid s_i \in a.inputstate(o)$ and $ncnf\_tran(a, s_i, s_{tgt}) \in I_M\}$
	$e_5 \in E_{r_2}{}^+$	$exp_{e_5}(i) = \{ncov\_tran(s_i, s_{tgt}) \mid (a, s_i, s_{tgt})$ is an induced transition of o in P and (a, s_i, s_{tgt}) is the only element in the coverage set of $(s_i, s_{tgt})\}$
	$e_6 \in E_{r_2}{}^+$	$exp_{e_6}(i) = \{ncnf\_last(s_i) \mid s_i \in a.inputstate(o)$ and s_{tgt} is the only state in $a.outputstate(o)$ and there is no transition from s_i to a final state in $OLC\}$
r_3	$e_7 \in E_{r_3}{}^-$	$exp_{e_7}(i) = \{ncnf\_tran(a, s_i, s_j) \mid s_i \in a.inputstate(o)$ and $s_j \in a.outputstate(o)$ and $ncnf\_tran(a, s_i, s_j) \in I_M\}$
	$e_8 \in E_{r_3}{}^+$	$exp_{e_8}(i) = \{ncov\_tran(s_i, s_j) \mid (a, s_i, s_j)$ is an induced transition of o in P and (a, s_i, s_j) is the only element in the coverage set of $(s_i, s_j)\}$

For resolving non-conformant transitions (settle, GR, NR) and (review, NR, RE), we get the following details about the effect of each resolution if we evaluate the defined side-effect expressions (see Fig.4(b)-(g)):

Resolutions for ncnf_tran(settle, GR, NR)

r_1 resolves ncnf_tran(settle, GR, NR) and introduces ncov_tran(GR,SE);
r_2 resolves ncnf_tran(settle, GR, NR), ncnf_tran(settle, RE, NR);
r_3 resolves ncnf_tran(settle, GR, NR), ncnf_tran(settle, RE, NR) and introduces ncov_tran(GR,SE).

Resolutions for ncnf_tran(review, NR, RE)

r_1 resolves ncnf_tran(review, NR, RE) and introduces ncnf_first(review, RE);
r_2 resolves ncnf_tran(review, NR, RE) and introduces ncnf_last(review, NR);
r_3 resolves ncnf_tran(review, NR, RE).

This detailed information about the effect of each resolution helps the user to decide which resolution to apply in each case. Generally, the most beneficial resolution would be the one that overall removes the greatest number of inconsistencies. In this example, we would choose r_2 to resolve ncnf_tran(settle, GR, NR) and r_3 to resolve ncnf_tran(review, NR, RE).

To satisfy Req2, our approach goes further to provide a more fine-grained comparison of resolutions based on *cost reduction* values calculated for each resolution.

Definition 5 (Cost reduction of a resolution). *Given a resolution $r \in R_t$ that can resolve an inconsistency $i \in I_M$ of type $t \in T_M$, with side-effects $E_r^- = \{e_{11}, ..., e_{1p}\}$ and $E_r^+ = \{e_{21}, ..., e_{2q}\}$, the* cost reduction of resolution r *is denoted by $costred_r$ and calculated as follows:*

$$costred_r = cost(t) + \sum_{j=1}^{p} \left(\mid exp_{e_{1j}}(i) \mid \times cost(type(e_{1j})) \right) - \sum_{k=1}^{q} \left(\mid exp_{e_{2k}}(i) \mid \times cost(type(e_{2k})) \right)$$

We now calculate cost reduction values for the resolutions in our example:

ncnf_tran(settle, GR, NR)	**ncnf_tran(review, NR, RE)**
$costred_{r_1} = 3 - (1 \times 1) = 2$	$costred_{r_1} = 3 - (1 \times 2) = 1$
$costred_{r_2} = 3 + (1 \times 3) = 6$	$costred_{r_2} = 3 - (1 \times 3) = 0$
$costred_{r_3} = 3 + (1 \times 3) - (1 \times 1) = 5$	$costred_{r_3} = 3$

It can be seen that based on the calculated cost reduction values, we can perform a more fine-grained comparison of the resolutions that takes into account the priorities or costs of different inconsistency types.

With our approach, we could also introduce more automation into the resolution process by applying resolutions without user intervention whenever there is one resolution that has a highest cost reduction value for a particular inconsistency. However, in our scenario, approving the choice of a resolution needs to be done by an expert who is aware of what impact the change in the process model has on the business. Provided that we are working with a model of an existing business process, removing an action from this model translates to removing a step in the process and may be difficult to implement in practice, even though the cost reduction value indicates that this is the best resolution.

4 Design and Implementation of Tool Support

In this section we present a design for an inconsistency resolution module based on our approach, which leads to an efficient implementation of inconsistency resolution.

Figure 5 shows the fundamental elements of the resolution module design, comprising several abstract classes (labeled in italics) that need to be extended to arrive at an executable implementation.

The central element of the module is the InconsistencyResolver that has references to all InconsistencyTypes that it can handle and a Model that is the subject of inconsistency handling. The checkAndResolve() method of the InconsistencyResolver is the entry-point to the inconsistency checking and resolution process.

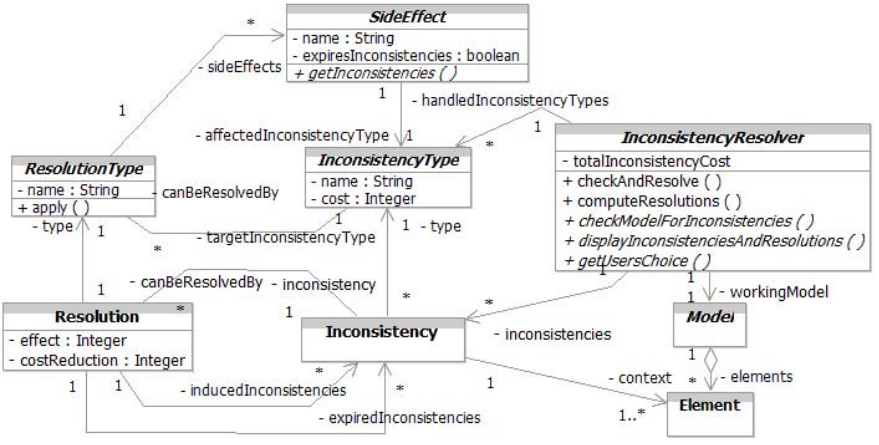

Fig. 5. Design for an inconsistency resolution module

Listing 1.1. InconsistencyResolver : checkAndResolve()

```
1  checkAndResolve()
2      inconsistencies = checkModelForInconsistencies();
3      while ( inconsistencies.size > 0) do
4          totalInconsistencyCost = 0;
5          for each ( Inconsistency i in inconsistencies ) do
6              totalInconsistencyCost += i.type.cost;
7          computeResolutions();
8          displayInconsistenciesAndResolutions ();
9          Resolution r = getUsersChoice();
10         if (r == null) then
11             return;
12         else
13             r.type.apply();
14         inconsistencies.remove(r.inconsistency );
15         inconsistencies.removeAll(r. expiredInconsistencies );
16         inconsistencies.add(r. inducedInconsistencies );
```

As shown in the method in Listing 1.1, the working model is first checked for inconsistencies and then the total inconsistency cost for the model is computed (lines 4-6). For each discovered inconsistency, possible resolutions are identified and these are communicated to the user (lines 7,8). After a resolution of user's choice is applied (lines 9-13), the model is not re-checked, but rather the inconsistency set is directly updated by removing the resolved inconsistency and those that expired and adding induced inconsistencies (lines 14-16). This added efficiency is valuable in practice, as in reality many models tend to get very large and as soon as re-checking them takes a noticeable amount of time, unnecessary interruptions are introduced into the resolution process.

As shown in Fig.5, each Resolution is associated with sets of inconsistencies that it will induce or expire if it is applied. An effect of a Resolution is the overall change in number of inconsistencies that it will inflict and its costReduction reflects how

the resolution will reduce the total inconsistency cost. These are determined in the computeResolutions() method of the InconsistencyResolver shown in Listing 1.2 as pseudocode. As each resolution resolves its target inconsistency, effect of each resolution is initialized to -1 (line 8) and initial costReduction is set to the cost of the target inconsistency type (line 9). Iterating over the resolution side-effects, values of effect and costReduction attributes are updated (lines 13-14,17-18).

Listing 1.2. InconsistencyResolver : computeResolutions()

```
1  computeResolutions()
2      for each ( Inconsistency i in inconsistencies ) do
3          i.canBeResolvedBy.clear();
4          for each (ResolutionType rt in i.type.canBeResolvedBy) do
5              Resolution r = new Resolution();
6              i.canBeResolvedBy.add(r);
7              r.inconsistency = i;
8              r.effect = -1;
9              r.costReduction = i.type.cost;
10             for each ( SideEffect se in rt.sideEffects ) do
11                 Set inconsistencies = se.getInconsistencies (i,workingModel);
12                 if (se.expiresInconsistencies ) then
13                     r.effect -= inconsistencies.size();
14                     r.costReduction += inconsistencies.size() * se.affectedInconsistencyType.cost;
15                     r.expiredInconsistencies.add( inconsistencies );
16                 else
17                     r.effect += inconsistencies.size();
18                     r.costReduction -= inconsistencies.size() * se.affectedInconsistencyType.cost;
19                     r.inducedInconsistencies.add( inconsistencies );
```

Our proposed design can be directly used to derive the implementation core of a resolution module, after which all abstract classes need to be extended to complete the implementation. For implementing support for inconsistency resolution between object life cycles and process models for example, we extend InconsistencyResolver with the concrete class ObjectLifeCycleProcessModelResolver that provides an implementation for finding inconsistencies in a model and communication with the user. For each resolution side-effect like the ones shown in Table 1, we create an extension of the SideEffect class and implement the getInconsistencies() method that evaluates the associated side-effect expression. Our approach does not place any restrictions on how transformations associated with resolutions are to be implemented, i.e. this can be done directly in a conventional programming language such as Java or using one of the existing model transformation approaches.

We have implemented a prototype for resolution of inconsistencies between object life cycles and process models, based on the presented approach. Our prototype is an extension to IBM WebSphere Business Modeler [2] that natively supports process modeling and has been extended for modeling object life cycles (Fig.6).

Detected inconsistencies are shown in the inconsistencies view below the model editors, where inconsistencies with higher costs are displayed higher in the list. The total number and cost of inconsistencies are shown at the top of this view, so that the effect on these numbers can be monitored during the resolution process.

After an inconsistency is selected, resolution side-effects and cost reduction values are calculated by the tool and can be taken into account for choosing the most appropriate resolution.

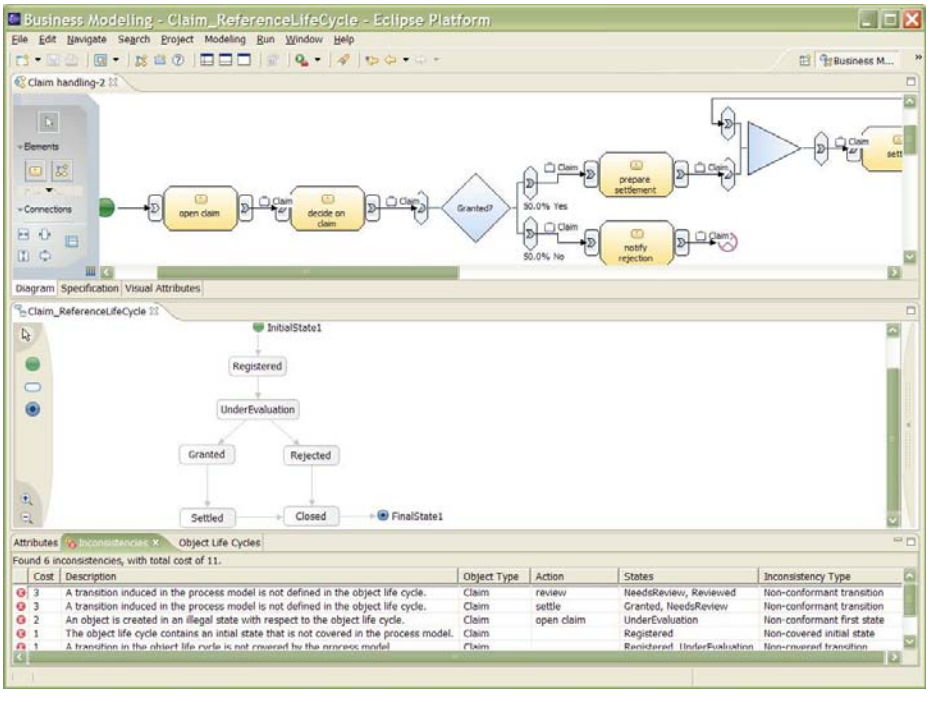

Fig. 6. Inconsistency resolution prototype in IBM WebSphere Business Modeler

We have used the prototype for resolving inconsistencies between several example models inspired by IAA [1]. This initial study has shown that the proposed approach provides powerful assistance for the user in selecting a resolution for each inconsistency.

5 Related Work

Recent work on inconsistency resolution includes work by Mens et al. [12,13] that uses the AGG [23] graph transformation tool to detect potential dependencies between different inconsistency resolutions. Inconsistency detection and resolution rules are expressed as graph transformation rules in AGG and are then analyzed using critical pair analysis. Analysis results point to potentially conflicting resolutions, resolutions that may induce or expire other types of inconsistencies and potential cycles between resolutions. In this paper, we propose explicit specification of side-effect expressions that can be evaluated given a model inconsistency and hence provide the user with a basis for comparison of

alternative resolutions. It would be interesting to investigate how automated dependency analysis can assist in the specification of side-effect expressions.

The FUJABA tool suite [15] supports both manual and automatic incremental inconsistency resolution [24]. Consistency checking rules can be configured by the user and organized into different categories in order to support domain- or project-specific consistency requirements. Consistency checking rules and inconsistency resolution rules are specified using graph grammar rules and executed by a FUJABA rule engine. Although different categories could also be used for obtaining different priorities, our approach can be seen as complementary because we focus on the evaluation of several alternative resolutions for one inconsistency based on side-effects and costs. Work on incremental transformations using triple graph grammars [19] studies the problem of keeping two models synchronized [8,18]. This is achieved by analyzing changes in one model and applying incremental updates for re-establishing consistency. Although these updates are analyzed for conflicts, a detailed evaluation of side-effects is not addressed.

Nentwich et al [14] propose to generate inconsistency resolutions (called repair actions) automatically from consistency constraints that are specified in first order logic. As opposed to our approach, generated repair actions do not take into account a concrete model violating consistency constraints and also do not consider side-effects.

Spanoudakis and Zisman [20] conducted a survey about inconsistency management and concluded that the most important open research issue in inconsistency handling is providing more guidance to the user for choosing among multiple alternative resolutions. The authors argue that resolutions should be ordered based on cost, risk and benefit. They further conclude that existing approaches do not adequately address efficiency and scalability of inconsistency detection in models that change during the resolution process. In our approach, we use side-effects and costs for evaluating alternative resolution and avoid rechecking the whole model after a resolution is applied.

Nuseibeh et al. [16] present a framework for managing inconsistency in software development. This framework comprises monitoring, identification and measuring of inconsistencies. Measuring inconsistencies includes attaching priorities to different inconsistencies. In our work, we use costs to reflect priorities of inconsistency types and also to calculate cost reduction for each resolution.

6 Conclusions and Future Work

In this paper, we introduce the concept of side-effect expressions that can be evaluated for a given inconsistent model to determine whether or not a resolution leads to new or expired inconsistencies. This allows the user to compare alternative resolutions for the same inconsistency. We attach costs to each inconsistency type, which enables us to calculate cost reduction values for each resolution and therefore provide a more fine-grained comparison of resolutions. Finally, we show how our concepts can be used to implement an efficient inconsistency resolution module for integration with an existing modeling tool.

Our case study and application of the prototype to various examples have shown that the approach adds significant value for the user during the resolution process. To enhance our solution further, we will next investigate more sophisticated cost models and study how cycles can be detected during the resolution process. Another area of future work is the application of our approach to domain-specific languages. Here, automated resolution dependency analysis and reusable side-effect specifications can be of interest in order to decrease the manual overhead of our approach.

Acknowledgements. We would like to thank Harald Gall, Jana Koehler, Cesare Pautasso, Hagen Völzer and Michael Wahler for their valuable feedback on an earlier version of this paper.

References

1. IBM Insurance Application Architecture, `http://www.ibm.com/industries/financialservices/doc/content/solution/278918103.html`
2. IBM WebSphere Business Modeler, `http://www.ibm.com/software/integration/wbimodeler/`
3. UML2.0 Superstructure, formal/05-07-04. OMG Document (2005)
4. Ebert, J., Engels, G.: Specialization of Object Life Cycle Definitions. Fachberichte Informatik 19/95, University of Koblenz-Landau (1997)
5. Engels, G., Küster, J.M., Groenewegen, L., Heckel, R.: A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models. In: ESEC'01. Proceedings of the 8th European Software Engineering Conference, pp. 186–195. ACM Press, New York (2001)
6. Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., Nuseibeh, B.: Inconsistency Handling in Multi-Perspective Specifications. IEEE Transactions on Software Engineering 20(8), 569–578 (1994)
7. Ghezzi, C., Nuseibeh, B.A.: Special Issue on Managing Inconsistency in Software Development (1). IEEE Transactions on Software Engineering 24(11) (November 1998)
8. Giese, H., Wagner, R.: Incremental Model Synchronization with Triple Graph Grammars. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 543–557. Springer, Heidelberg (2006)
9. Kappel, G., Schrefl, M.: Object/Behavior Diagrams. In: Proceedings of the 7th International Conference on Data Engineering, Washington, DC, USA, pp. 530–539. IEEE Computer Society, Los Alamitos (1991)
10. Kleppe, A., Warmer, J.: The Object Constraint Language, 2nd edn. Addison-Wesley, Reading (2003)
11. Küster, J.M.: Consistency Management of Object-Oriented Behavioral Models. PhD thesis, University of Paderborn (March 2004)
12. Mens, T., Van Der Staeten, R., Warny, J.-F.: Graph-Based Tool Support to Improve Model Quality. In: Proceedings of the 1st Workshop on Quality in Modeling co-located with MoDELS 2006, Technical report 0627, Technische Universiteit Eindhoven, pages 47–62 (2006)

13. Mens, T., Van Der Straeten, R., D'Hondt, M.: Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 200–214. Springer, Heidelberg (2006)
14. Nentwich, C., Emmerich, W., Finkelstein, A.: Consistency Management with Repair Actions. In: Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, USA, May 3-10, 2003, pp. 455–464. IEEE Computer Society, Los Alamitos (2003)
15. Nickel, U.A., Niere, J., Zündorf, A.: Tool Demonstration: The FUJABA Environment. In: ICSE. Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland, pp. 742–745. ACM Press, New York (2000)
16. Nuseibeh, B., Easterbrook, S., Russo, A.: Making Inconsistency Respectable in Software Development. Journal of Systems and Software 58(2), 171–180 (2001)
17. Ryndina, K., Küster, J.M., Gall, H.: Consistency of Business Process Models and Object Life Cycles. In: Kühne, T. (ed.) Workshops and Symposia at MoDELS 2006. LNCS, vol. 4364, pp. 80–90. Springer, Heidelberg (2007)
18. Lohmann, S., Westfechtel, B., Becker, S., Herold, S.: A Graph-Based Algorithm for Consistency Maintenance in Incremental and Interactive Integration Tools. Journal of Software and Systems Modeling (to appear, 2007)
19. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
20. Spanoudakis, G., Zisman, A.: Handbook of Software Engineering and Knowledge Engineering, chapter Inconsistency Management in Software Engineering: Survey and Open Research Issues, pp. 329–380. World Scientific Publishing Co. (2001)
21. Van Der Straeten, R.: Inconsistency Management in Model-Driven Engineering. PhD thesis, Vrije Universiteit Brussel (September 2005)
22. Van Der Straeten, R., D'Hondt, M.: Model Refactorings through Rule-Based Inconsistency Resolution. In: SAC. Proceedings of the 2006 ACM Symposium on Applied Computing, Dijon, France, April 23-27, 2006, pp. 1210–1217. ACM, New York (2006)
23. Taentzer, G.: AGG: A Graph Transformation Environment for Modeling and Validation of Software. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 446–453. Springer, Heidelberg (2004)
24. Wagner, R., Giese, H., Nickel, U.: A Plug-In for Flexible and Incremental Consistency Management. In: Proceedings Workshop on Consistency Problems in UML-based Software Development, San Francisco, USA, Technical Report. Blekinge Institute of Technology, San Francisco (October 2003)