Appl Intell
DOI 10.1007/s10489-007-0105-8

Automated debugging of recommender user interface descriptions

Alexander Felfernig · Gerhard Friedrich · Klaus Isak · Kostyantyn Shchekotykhin · Erich Teppan · Dietmar Jannach

Abstract Customers interacting with online selling platforms require the assistance of sales support systems in the product and service selection process. Knowledge-based recommenders are specific sales support systems which involve online customers in dialogs with the goal to support preference forming processes. These systems have been successfully deployed in commercial environments supporting the recommendation of, e.g., financial services, e-tourism services, or consumer goods. However, the development of user interface descriptions and knowledge bases underlying knowledge-based recommenders is often an error-prone and frustrating business. In this paper we focus on the first aspect and present an approach which supports knowledge engineers in the identification of faults in user interface descriptions. These descriptions are the input for a model-based diagnosis algorithm which automatically identifies faulty elements and indicates those elements to the knowledge engineer. In addition, we present results of an empirical study which demonstrates the applicability of our approach.

1 Introduction

Recommender technologies [1–3, 9, 13, 19, 20, 26, 30] are of great importance for making product and service assortments more accessible. Basically, there are three technological approaches to the implementation of a recommender application:

1. *Content-based filtering* [20] derives product and service recommendations by detecting similarities between the preferences of a customer and existing product and service (item) descriptions. The recommender proposes items which are similar to those the customer has liked in the past. For instance, if a customer has bought books about the *SAP system*, similar books will be recommended in future advisory sessions. Consequently content-based approaches do not support the exploitation of serendipity effects which are in many cases required and welcome in recommendation contexts. A typical application of content-based filtering is the recommendation of Web sites.
2. *Collaborative filtering* algorithms [13, 22, 26] exploit information about preferences of a large group of customers. Item recommendations are derived by taking into account preferences of customers with similar purchasing patterns, for instance, movies not yet bought by the current customer but positively rated by customers with similar purchasing behavior will be recommended to the current customer. These algorithms are used in many cases for the recommendation of simple products such as books, movies, or compact disks.
3. *Knowledge-based recommender systems* [1, 3, 15, 23, 27, 30] exploit deep knowledge about products and services. Customers purchasing complex products such as computers or financial services are much more in the need of information and intelligent decision support in order to retrieve the best-suiting solution. In order to provide such an intelligent decision support, we need an explicit representation of product, marketing, and sales knowledge

A. Felfernig (✉) · G. Friedrich · K. Shchekotykhin · E. Teppan · D. Jannach
Department of Intelligent Systems and Business Informatics, University Klagenfurt, Universitätsstrasse 65-67, 9020 Klagenfurt, Austria
e-mail: felfernig@ifit.uni-klu.ac.at

A. Felfernig · K. Isak
ConfigWorks, Lakeside, B01, 9020 Klagenfurt, Austria

[9] which makes it possible (a) to derive recommendations which comply with the strategies of the company and suit the wishes of a customer, (b) to explain those recommendations, and (c) to support customers when no solution could be found. Particularly knowledge-based recommenders provide these functionalities.

When developing a knowledge-based recommender application (e.g., an investment recommender or a digital camera recommender), two basic aspects have to be taken into account by knowledge engineers:

1. A *recommender knowledge base* [3, 6] has to be defined which consists of three major parts. (a) A description of the provided items, e.g., financial services can be described by their *name*, *recommended investment period*, and *expected return rate*. (b) A description of the possible customer properties (e.g., *name*, *age*, *family status*, *available guarantees*) and customer requirements (e.g., *personal goals*, *expected return rate*, *intended duration of investment*, *degree of preparedness to take risks*). (c) Constraints restricting the allowed combinations of customer requirements and item recommendations. For instance, *required short-term investment periods are incompatible with high return rates* in the case that *speculation* is not the personal goal.
2. A *process* has to be defined which describes the intended behavior of the recommender user interface [3, 10], i.e., which questions regarding his/her personal properties and requirements have to be posed to a customer in which context and in which order? For instance, if the answer of the customer regarding the question on *personal expertise with financial services* is *expert*, the recommender application will select subsequent questions formulated on a more technical level. Otherwise—for customers defining themselves as *beginners*—a strictly goal-oriented dialog will be chosen.

Knowledge bases and related process definitions are the major parts of a recommender application. Having completed the definition of both, a corresponding executable application can be automatically generated [3, 5, 9]. An example for such an application is given in Fig. 1. This application supports the interactive recommendation of loans. The application poses a number of questions to a customer, e.g., *what are your personal goals* (*modification of building*, ..., *new house*, *extension of building*)? or *do you have already existing real estates* (*yes*, *yes but already burdened with mortgage*, *no*)? Each of these questions is assigned to a certain state (e.g., the question regarding *existing real estates* is assigned to the state *creditworthiness check*). In our example application, questions related to the *personal properties and goals* have been posed before questions regarding *creditworthiness*. Consequently, the order in which questions are posed to the customer strictly depends on the state those questions are assigned to. After all relevant questions have been answered by the customer, the recommender application proposes a corresponding set of recommendations. For each recommendation, a corresponding set of explanations can be shown which clearly demonstrate why a certain item has been chosen by the recommender.

In the remainder of this paper we focus on a situation where a knowledge engineer develops a model of the intended behavior of a recommender user interface. In this context we introduce automated debugging techniques which effectively support the identification of faults in those descriptions. In Sect. 2 we introduce a simple example user interface description which will be used throughout this paper for demonstration purposes. In Sect. 3 we introduce a finite state representation formalism for modeling the intended navigational behavior of recommender user interfaces. Using the concepts of Model-Based Diagnosis (MBD) [21], we present an approach to the automated identification (debugging) of minimal sets of faulty transition conditions in user interface descriptions (see Sect. 4). In Sect. 5 we evaluate the performance of the presented debugging algorithm and present results of an empirical study. Finally, Sect. 6 contains a discussion on related work.

2 Recommender user interface descriptions

The intended behavior of a recommender user interface can be described by a finite state model [10, 14, 32]. Each state of such a model represents an input unit of the application where a user can articulate his/her preferences by answering questions posed by the recommender application (see Fig. 1). Figure 2 depicts a simple example for the model of the intended behavior of a financial services recommender application. Such models are developed on the basis of our commercially available recommender modeling environment (Process Designer) [3, 8, 9] which supports the interactive design of recommender user interface descriptions (see Sect. 5). This environment allows the specification of a finite state model where the states represent input units allowing customers to articulate their requirements, e.g., in state q_2 the customer is asked about the preferred duration of investment.[1] Having defined such a model, our development environment automatically generates a corresponding application (see, e.g., Fig. 1).

Depending on the preferences articulated by the customer, the automaton of Fig. 2 changes its state, e.g., an expert (kl = expert) who is not interested in financial services advisory (aw = no), is forwarded to the state q_3, where a

[1] Note that *duration_of _investment* (*id*) is the identifier for the corresponding question posed by the recommender application (*what is your required duration of investment*?).

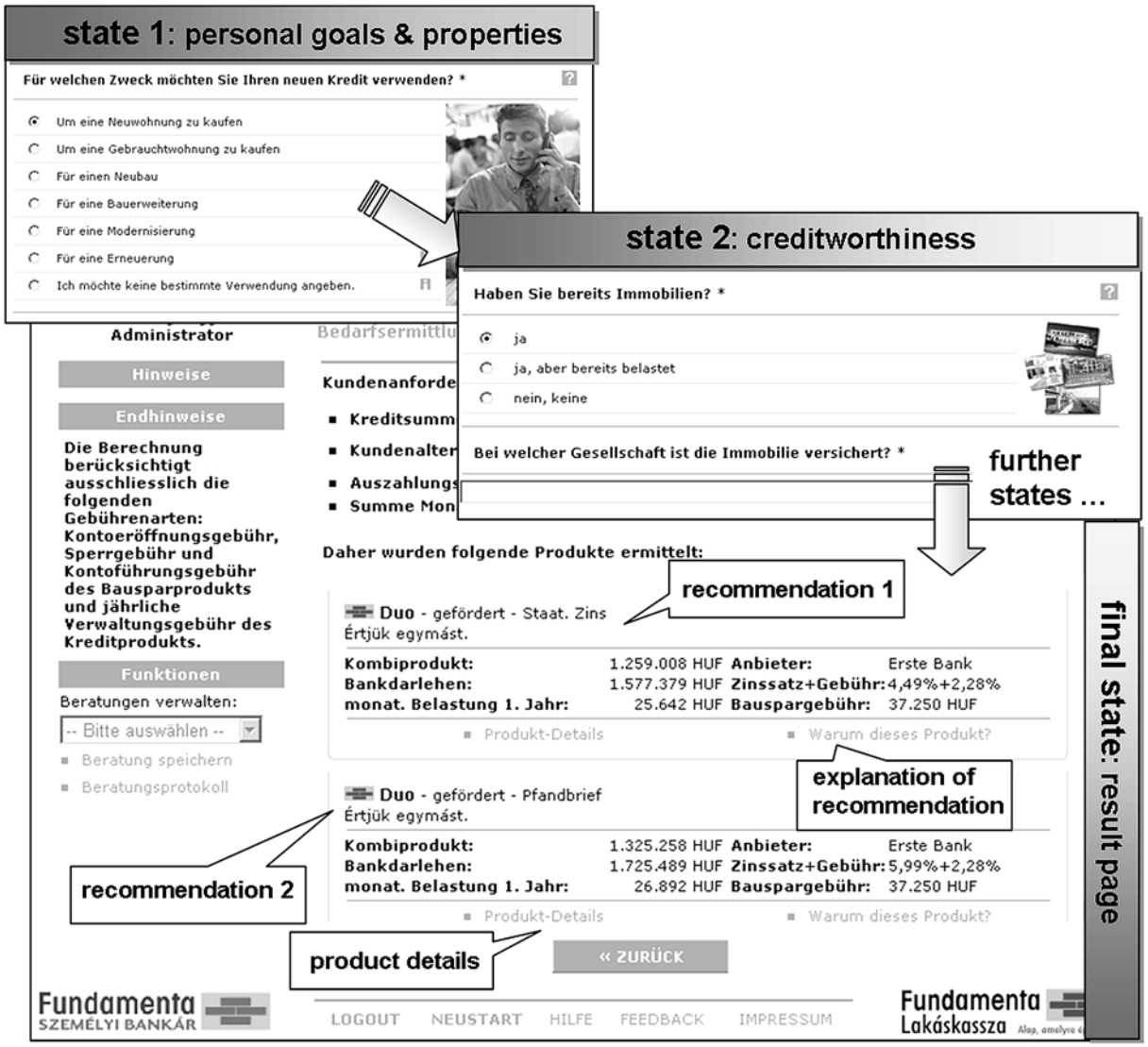

Fig. 1 Example user interface of financial service recommender

direct product search can be performed. Consequently, different navigation paths determine different subsets of input variables (questions) relevant for the preference elicitation process. After the completion of the preference elicitation process (a final state of the process definition has been reached), the recommender application can calculate and present a corresponding set of solutions [3, 9].

Note that the specification of our financial services recommender interface in Fig. 2 is faulty. A financial services expert (kl = expert) who wants to be advised by a financial services recommender (aw = yes) and is interested in long-term investments (id $\neq$ shortterm) and doesn't have any available funds (av = no) comes to a standstill at the input of availability (the transition condition sets $\{c_2, c_9\}$ and $\{c_2, c_{11}\}$ are contradictory). Furthermore, the transition between the states q_4 and q_5 will never be accessed since the transition condition c_{11} is not satisfiable, i.e., there is no possible navigation path which includes the mentioned transition condition. We denote such basic properties which should be fulfilled by recommender user interfaces as *well-formedness rules*. Such rules will be discussed in detail in Sect. 3. Being confronted with faulty interface descriptions, knowledge engineers can be supported by debugging functionalities which immediately detect the sources of inconsistency. Such concepts will be presented in the following sections.

3 Finite state models of recommender user interfaces

For the definition of the intended behavior of a recommender user interface, we introduce the concept of Predicate-based

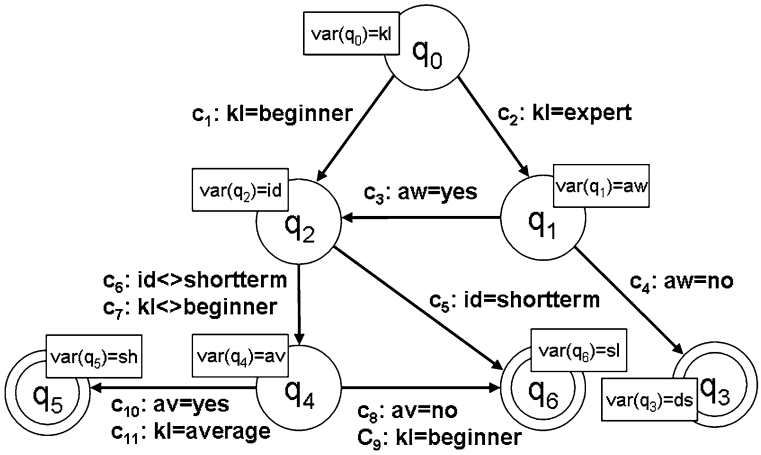

Fig. 2 Example recommender user interface description

Finite State Automata (PFSA) [10, 32] (see Fig. 2) which are a specific variant of finite state automata [14]. This type of automaton is more compact in the way state transitions can be defined (domain restrictions of finite domain variables), which makes it an excellent formalism for the graphical design and maintenance of recommender user interfaces. This representation of recommender user interfaces is integrated into the recommender development environment presented in [3, 9]. Note that in the context of building knowledge-based recommender applications, we are primarily interested in *acyclic* automata.

Definition 1 (PFSA) A Predicate-based Finite State Automaton (recognizer) (PFSA) is defined as a 6-tuple $(Q, \Sigma, \Pi, E, S, F)$, where

- $Q = \{q_1, q_2, \ldots, q_j\}$ is a finite set of states, where $\text{var}(q_i) = \{x_i\}$ is a finite domain variable assigned to q_i, $\text{prec}(q_i) = \{\phi_1, \phi_2, \ldots, \phi_m\}$ is the set of preconditions of q_i ($\phi_\alpha = \{c_r, c_s, \ldots, c_t\} \subseteq \Pi$), $\text{postc}(q_i) = \{\psi_1, \psi_2, \ldots, \psi_n\}$ is the set of postconditions of q_i ($\psi_\beta = \{c_u, c_v, \ldots, c_w\} \subseteq \Pi$), and $\text{dom}(x_i) = \{x_i = d_{i1}, x_i = d_{i2}, \ldots, x_i = d_{ip}\}$ denotes the set of possible assignments of x_i, i.e., the domain of x_i.
- $\Sigma = \{x_i = d_{ij} \mid x_i \in \text{var}(q_i), x_i = d_{ij} \in \text{dom}(x_i)\}$ is a finite set of variable assignments, the input alphabet.
- $\Pi = \{c_1, c_2, \ldots, c_q\}$ is a set of constraints (transition conditions) restricting the set of words accepted by the PFSA.
- E is a finite set of transitions $\subseteq Q \times \Pi \times Q$.
- $S \subseteq Q$ is a finite set of start states.
- $F \subseteq Q$ is a finite set of final states.

Preconditions of a state q_i ($\text{prec}(q_i) = \{\phi_1, \phi_2, \ldots, \phi_m\}$) can be automatically derived from the reachability tree of a PFSA. Figure 3 depicts the reachability tree for the PFSA of Fig. 2. The state q_2 is accessed twice in the reachability tree. Consequently, we can derive two preconditions for the state q_2 which directly correspond to the transition conditions of paths in the reachability tree leading to q_2, i.e., $\text{prec}(q_2) = \{\{c_1\}, \{c_2, c_3\}\}$ where the different subsets are interpreted as being part of a disjunction (not every precondition has to be fulfilled). Similarly, $\text{postc}(q_i)$ represents the set of possible postconditions of the state q_i which are also derived from the reachability tree, e.g., the state q_4 has two postconditions, namely $\{\{c_8, c_9\}, \{c_{10}, c_{11}\}\}$. Figure 4 depicts the textual representation of the PFSA of Fig. 2.

The set of input sequences leading to a final state is also denoted as the language accepted by the PFSA. A word $w \in \Sigma^*$ (i.e., a sequence of user inputs) is accepted by a PFSA if there is an accepting run of w in the PFSA (see [10]).

When developing user interfaces, mechanisms have to be provided which support the effective identification of violations of *well-formedness properties*, e.g., if a path in the process definition reaches a state q_i, there must be at least one extension of this path to a final state. Regarding our example of Fig. 2 there exist accepted input sequences visiting the states $[q_0, q_1, q_2, q_4]$, but none of those sequences can be propagated to any of the following states $\{q_5, q_6\}$. Path expressions form the basis for expressing well-formedness properties on a PFSA (see Definitions 2a, 2b).

Definition 2a (Path) We define a sequence (of transitions) $p = [(q_1, C_1, q_2), (q_2, C_2, q_3), \ldots, (q_{i-1}, C_{i-1}, q_i)]((q_\alpha, C_\alpha, q_\beta) \in E)$ as *path* of a given PFSA.

Definition 2b (Consistent path) Let $p = [(q_1, C_1, q_2), (q_2, C_2, q_3), \ldots, (q_{i-1}, C_{i-1}, q_i)]((q_\alpha, C_\alpha, q_\beta) \in E)$ be a path from a state $q_1 \in S$ to a state $q_i \in Q$. p is *consistent* (consistent(p)) iff $\cup C_\alpha$ is consistent.

Note that we interpret consistency in the sense of logical satisfiability, i.e., are all the constraints (logical sentences) in $\cup C_\alpha$ satisfiable. Following this definition of a consistent path we introduce a set of well-formedness rules which specify important structural properties of a PFSA (counter examples for these properties are depicted in Fig 5). These

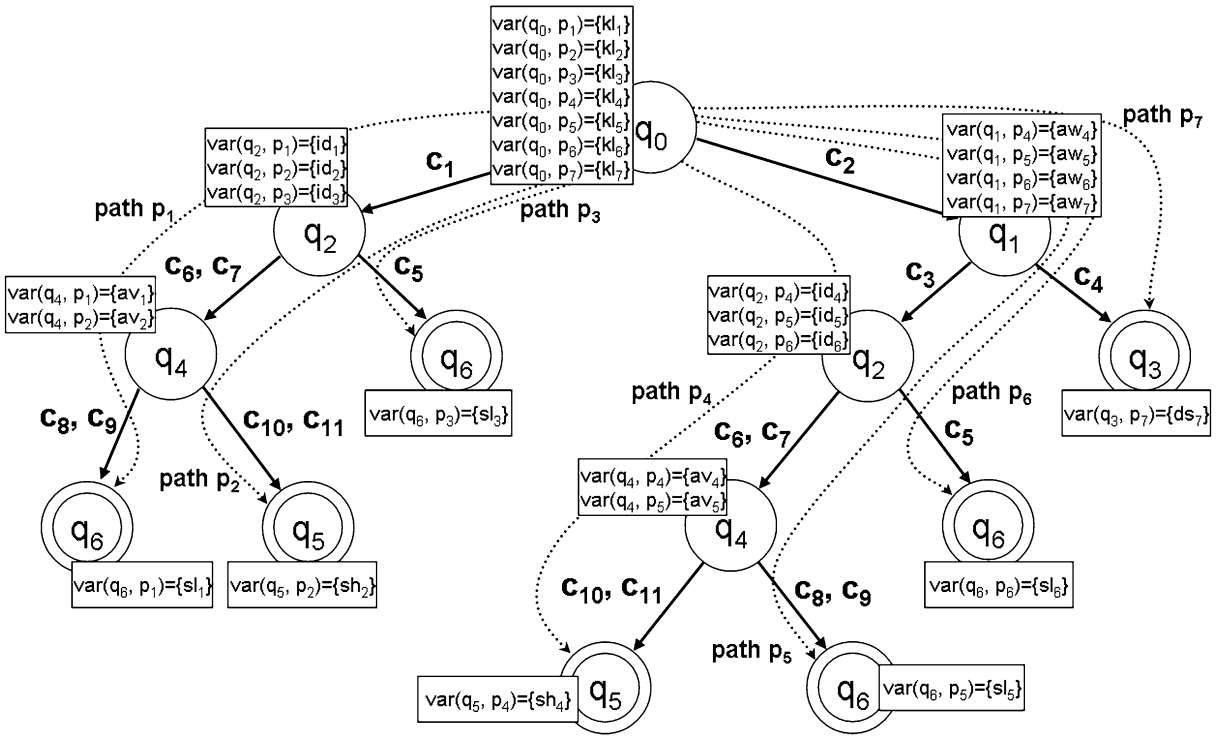

Fig. 3 Reachability tree of PFSA (depicted in Fig. 2)

```
Q = {q0, q1, q2, q3, q4, q5, q6}.

/* knowledge level */
var(q0) = {kl}.
/* advisory wanted */
var(q1) = {aw}.
/* duration of investment */
var(q2) = {id}.
/* direct product search */
var(q3) = {ds}.
/* availability of
 financial resources*/
var(q4) = {av}.
/* high risk products */
var(q5) = {sh}.
/* low risk products */
var(q6) = {sl}.

dom(kl) = {kl=beginner,kl=average,
      kl=expert}.
dom(aw) = {aw=yes,aw=no}.
dom(id) = {id=shortterm,id=mediumterm,
      id=longterm}.
dom(ds) = {ds=savings,ds=bonds,
           ds=stockfunds,
           ds=singleshares}
dom(av) = {av=yes,av=no}.
dom(sh) = {sh=stockfunds,sh=singleshares}.
dom(sl) = {sl=savings,sl=bonds}.
prec(q0) = {{true}}.
prec(q1) = {{c2}}.
prec(q2) = {{c1}, {c2, c3}}.
prec(q3) = {{c2, c4}}.
/* ... */
```

```
postc(q0) = {{c2, c4}, {c2, c3, c5},
             {c2, c3, c6, c7, c8, c9},
             {c2, c3, c6, c7, c10, c11},
             {c1, c5},
             {c1, c6, c7, c8, c9},
             {c1, c6, c7, c10, c11}}.
postc(q1) = {{c4}, {c3, c5},
             {c3, c6, c7, c8, c9},
             {c3, c6, c7, c10, c11}}.

/* ... */

postc(q4) = {{c8, c9}, {c10, c11}.

postc(q3) = {{true}}.
postc(q5) = {{true}}.
postc(q6) = {{true}}.

Σ = {kl=beginner, kl=average,
    kl=expert,
    aw=yes, aw=no, ...,
    sl=savings, sl=bonds}.

Π = {c1, c2, ..., c11}.
E= {(q0, {c2}, q1), (q0, {c1}, q2),
   (q1, {c4}, q3),
   (q1, {c3}, q2), (q2, {c6, c7}, q4),
   (q2, {c5}, q6),
   (q4,{c8, c9}, q6),
   (q4, {c10, c11}, q5)}.
S= {q0}.
F= {q3, q5, q6}.
```

Fig. 4 Textual version of PFSA (depicted in Fig. 2)

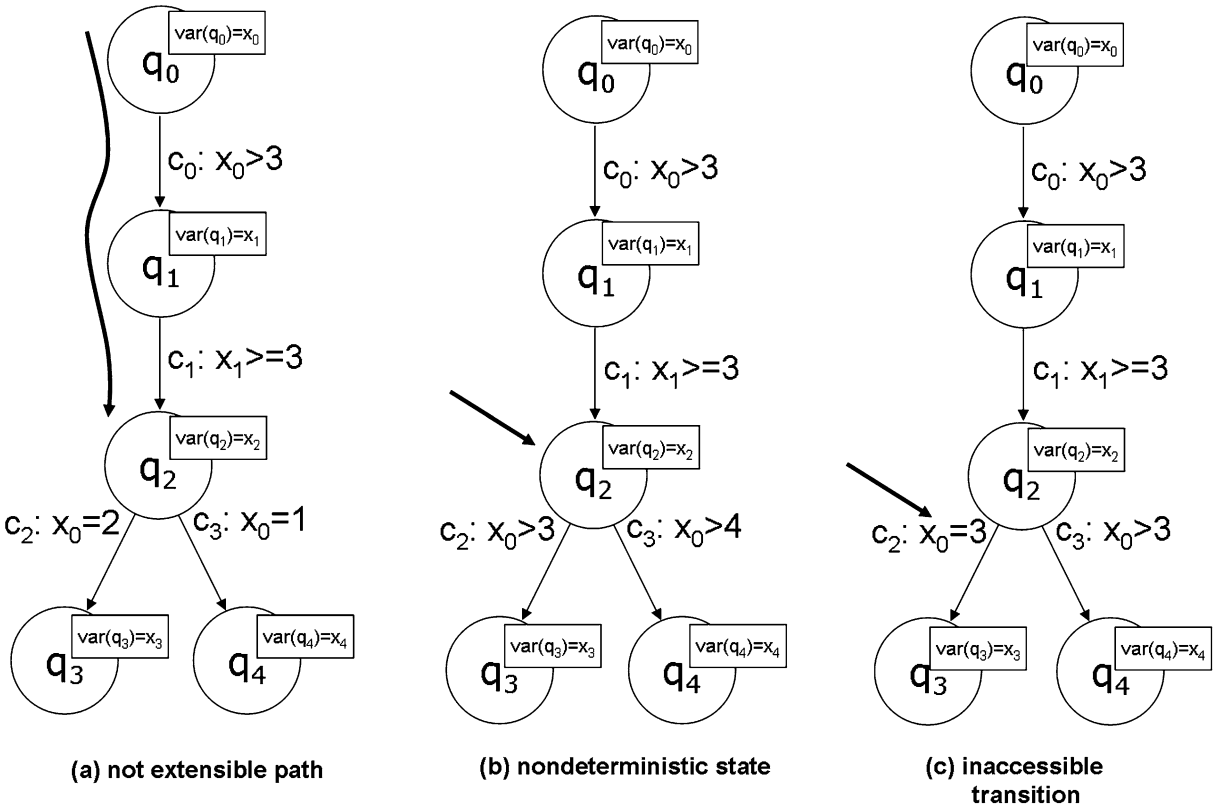

Fig. 5 Counter examples for well-formedness rules

rules have shown to be relevant for the implementation of knowledge-based recommender applications. Note that if additional well-formedness rules are needed, our framework allows the introduction of further domain-specific properties.

Extensibility. For each consistent path in a PFSA leading to a state q_i there must exist a corresponding direct postcondition, i.e., (q_i, C_i, q_{i+1}) propagating the path (i.e., each consistent path must be extensible) (see Definition 3).

Definition 3 (Extensible path) Let $p = [(q_1, C_1, q_2), (q_2, C_2, q_3), \ldots, (q_{i-1}, C_{i-1}, q_i)]$ be a consistent path from a state $q_1 \in S$ to a state $q_i \in Q - F$. p is *extensible* (extensible(p)) iff $\exists(q_i, C_i, q_{i+1})$: $C_1 \cup C_2 \cup \cdots \cup C_{i-1} \cup C_i$ is consistent.

Figure 5(a) depicts a *non-extensible path*, since the conditions $\{c_0, c_1\}$ of $p = [(q_0, \{c_0 : x_0 > 3\}, q_1), (q_1, \{c_1 : x_1 >= 3\}, q_2)]$ are inconsistent with both conditions of postc(q_2) = {{c_2},{c_3}}. Similarly, Fig. 2 includes a non-extensible path: $[(q_0, \{c_2 : \text{kl} = \text{expert}\}, q_1), (q_1, \{c_3 : \text{aw} = \text{yes}\}, q_2), (q_2, \{c_6 : \text{id} \neq \text{shortterm}, c_7 : \text{kl} \neq \text{beginner}\}, q_4)]$ is inconsistent with the conditions of postc(q_4) = {{c_{10}, c_{11}}, {c_8, c_9}}.

Determinism. Each state q_i is a decision point for the determination of the next state. This selection strictly depends on the definition of the direct postconditions for q_i, where each postcondition has to be unique for determining the subsequent state. A state q_i is deterministic if each of its postconditions is unique for determining subsequent states (see Definition 4).

Definition 4 (Deterministic state) Let $p = [(q_1, C_1, q_2), (q_2, C_2, q_3), \ldots, (q_{i-1}, C_{i-1}, q_i)]$ be a path from a state $q_1 \in S$ to a state $q_i \in Q - F$. A state (q_i) is *deterministic* iff $\forall(q_i, C_{i1}, q_j), (q_i, C_{i2}, q_k) \in E : C_1 \cup C_2 \cup \cdots \cup C_{i-1} \cup C_{i1} \cup C_{i2}$ is contradictory ($C_{i1} \neq C_{i2}$).

Well-formedness rules related to deterministic states require that each input sequence consistent with the transition conditions $\{C_1, C_2, \ldots, C_{i-1}\}$ is consistent with at most one of the following conditions $\{C_{i1}, C_{i2}\}$. Figure 5(b) depicts a *nondeterministic state*, since the conditions $\{c_0, c_1\}$ are consistent with $\{c_2, c_3\}$. By introducing negated neighbor conditions (C_{i1} is the neighbor of C_{i2}), we can achieve the inconsistency required by Definition 4. Exchanging neighbor conditions is basically realized by a pairwise exchange of the negations of existing transition conditions.[2] This ap-

[2]Note that this approach to a pairwise exchange of negated conditions is as well applicable in situations with more that two post-conditions of a given state.

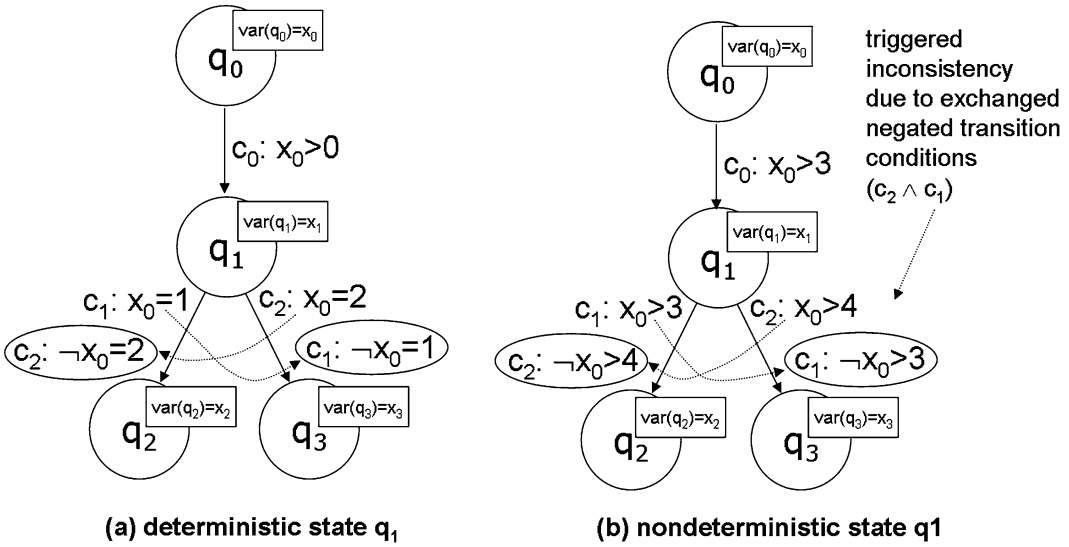

Fig. 6 Handling of nondeterministic transition conditions: state q_1 is nondeterministic (**b**); on the basis of a pairwise exchange of negated neighbor conditions, we can trigger an inconsistency which has to be handled by the diagnosis process

proach is exemplified in Fig. 6 where in case (a) the original semantics is preserved and in case (b) an inconsistency ($c_2 : x_0 > 4 \wedge c_1 : \neg x_0 > 3$) is triggered which has to be resolved by the diagnosis process (see Sect. 4).

Accessibility. Each transition should be accessible, i.e., for each transition there exists at least one corresponding path (see Definition 5).

Definition 5 (Accessible transition) A transition $t = (q_i, C_i, q_{i+1})$ (postcondition of state q_i) is *accessible* (accessible(t)) iff there exists a path $p = [(q_1, C_1, q_2), (q_2, C_2, q_3), \ldots, (q_{i-1}, C_{i-1}, q_i)](q_1 \in S)$: $C_1 \cup C_2 \cup \cdots \cup C_{i-1} \cup C_i$ is consistent.

Figure 5(c) depicts an *inaccessible transition*, since c_2 (transition $(q_2, \{c_2\}, q_3)$) is inconsistent with $\{c_0, c_1\}$, the conditions of the only path leading to q_2. Similarly, Fig. 1 contains an inaccessible transition: none of the possible paths are consistent with the transition conditions of $(q_4, \{c_{10}{:}\text{av} = \text{yes}, c_{11} : \text{kl} = \text{average}\}, q_5)$.

Well-formed PFSA. A PFSA is well-formed, if the defined set of well-formedness rules is fulfilled (see Definition 6).

Definition 6 (Well-formed PFSA) A PFSA is well-formed iff

- each consistent path $p = [(q_1, C_1, q_2), (q_2, C_2, q_3), \ldots, (q_{i-1}, C_{i-1}, q_i)]$ $(q_1 \in S,\ q_i \in Q - F)$ is extensible to a consistent path $p = [(q_1, C_1, q_2), (q_2, C_2, q_3), \ldots, (q_{i-1}, C_{i-1}, q_i), (q_i, C_i, q_j)]$.
- $\forall q_k \in$ Q: deterministic(q_k).
- $\forall t = (q_k, C_k, q_l) \in E$: accessible($t$).

Having specified necessary properties of a PFSA, we now present our approach to the calculation of minimal sets of faulty transition conditions in a PFSA.

4 Debugging finite state models

Given a faulty (not well-formed) PFSA, we want to automatically identify a minimal set of faulty transition conditions. In order to solve this task, we define a PFSA diagnosis problem which is solved using the concepts provided by model-based diagnosis (MBD) [21]. Model-based diagnosis starts with the description of a system (SD) which is in our case the structural description of the intended behavior of a PFSA (Definitions 3–5). If the actual behavior of the system conflicts with its intended behavior, the diagnosis task is to determine those components (transitions) which, when assumed to be functioning abnormally, explain the discrepancy between the actual and the intended system behavior.

In order to apply MBD concepts, we transform a PFSA into a corresponding constraint-based representation. This transformation is based on the analysis of the reachability tree generated from a given PFSA (see, e.g., Fig. 3). This tree contains for each consistent path of the PSFA a corresponding set of input variables which allow us to reason about potential behaviors of the PFSA on the basis of a simple and wide-spread constraint-based representation. In addition to the variables the corresponding transition conditions as well are transformed into a corresponding constraint-based representation. Both of these translations will be exemplified in the following subsections.

1. STAT: finite domain variables representing possible user inputs.

2. WF: constraints representing well-formedness rules.
3. TRANS: constraints representing the transition conditions of the PFSA.

The goal of a diagnosis task is to identify a minimal set of transition conditions ($\subseteq$TRANS) which are responsible for the faulty behavior of the PFSA, i.e., are inconsistent with the given set of well-formedness rules. Note that diagnoses do not need to be unique, i.e., there can be different explanations for faulty transition conditions in the PFSA. We define a PFSA Diagnosis Problem as follows.

Definition 7 (PFSA diagnosis problem) A PFSA Diagnosis Problem is represented by a tuple (SD, TRANS), where SD = STAT $\cup$ WF. STAT is the structural description of a PFSA represented by a set of finite domain variables. WF is the intended behavior of a PFSA (set of well-formedness rules) which is represented by a set of constraints on STAT. Finally, TRANS represents a set of transition conditions (as well represented by constraints on STAT).

Note that (STAT, WF, TRANS) defines a Constraint Satisfaction Problem (CSP) [31]. STAT is a set of finite domain variables related to paths of the reachability tree, e.g., $\{kl_3, id_3, sl_3\}$ are variables related to the path p_3 (Fig. 3 depicts the relationship between paths and variables $\in$ STAT, e.g., $var(q_0, p_3) = \{kl_3\}$). The complete set of variables related to paths of the reachability tree is included in Example 1. Note that each of the variables $\in$ STAT is either active ($\mathcal{ACT}$) or inactive ($\mathcal{IACT}$). This notion of variable activity has been introduced by [25]. Although we apply a simplified version of this approach (every variable is either active or inactive without any additional activation constraints), this representation perfectly supports our goal of testing well-formedness properties of process definitions. The set of solutions to the CSP defined by (STAT, WF, TRANS) represents all possible interaction sequences (accepted runs). The projection of those solutions to, e.g., the variables $\{kl_3, id_3, sl_3\}$ represents those input sequences accepted by path p_3, i.e., [kl = beginner, id = shortterm, sl = savings], [kl = beginner, id = shortterm, sl = bonds]. For our example PFSA, STAT is defined as follows.[3]

Example 1 (STAT): STAT = {
kl_1, id_1, av_1, sl_1, /* path p_1 */
kl_2, id_2, av_2, sh_2, /* path p_2 */
kl_3, id_3, sl_3, /* path p_3 */
$kl_4, aw_4, id_4, av_4, sh_4$, /* path p_4 */
$kl_5, aw_5, id_5, av_5, sl_5$, /* path p_5 */
kl_6, aw_6, id_6, sl_6, /* path p_6 */
kl_7, aw_7, ds_7 /* path p_7 */}.

[3]The corresponding variable domains are depicted in Fig. 4.

Since in our case a reachability tree (see, e.g., Fig. 3) represents the complete expansion of a corresponding PFSA, not all the paths are necessarily consistent. If a path of the reachability tree represents such an illegal trajectory, i.e., no consistent value assignment exists for the corresponding variables, all variables of this path have to be inactive. In order to assure that all variables of a path are either active or inactive, we introduce meta-constraints defined for each path in the reachability tree, e.g., for path p_3: $\mathcal{ACT}(kl_3) \wedge \mathcal{ACT}(id_3) \wedge \mathcal{ACT}(sl_3) \vee \mathcal{IACT}(kl_3) \wedge \mathcal{IACT}(id_3) \wedge \mathcal{IACT}(sl_3)$. This constraint denotes the fact that either all variables of path p_3 must be active or all variables are inactive. In order to simplify the construction of well-formedness rules, we introduce additional meta-constraints which define the *activity state* of a path variable, e.g., for path p_3: $\mathcal{ACT}(kl_3) \wedge \mathcal{ACT}(id_3) \wedge \mathcal{ACT}(sl_3) \leftrightarrow \mathcal{ACT}(p_3)$, and $\mathcal{IACT}(kl_3) \wedge \mathcal{IACT}(id_3) \wedge \mathcal{IACT}(sl_3) \leftrightarrow \mathcal{IACT}(p_3)$. In order to introduce such meta-constraints, we have to introduce $\{p_1, p_2, \ldots, p_k\} \subset$ TRANS.

In the following we give examples for the construction of well-formedness rules (WF) needed for the identification of minimal sets of faulty transition conditions in the given example PFSA.

First, we give an example for the construction of an *accessibility* rule related to the transition $(q_2, \{c_6, c_7\}, q_4)$ of our example PFSA (see Example 2).

Example 2 (Well-formedness rules for accessibility) WFaccessibility $((q_2, \{c_6, c_7\}, q_4)) = \{\mathcal{ACT}(p_1) \vee \mathcal{ACT}(p_2) \vee \mathcal{ACT}(p_4) \vee \mathcal{ACT}(p_5)\}$.

This rule denotes the fact that the transition $(q_2, \{c_6, c_7\}, q_4)$ must be accessible for at least one of the paths p_1, p_2, p_4, p_5 (see Fig. 3), i.e., at least one of the variable sets $\{kl_1, id_1, av_1, sl_1\}$, $\{kl_2, id_2, av_2, sh_2\}$, $\{kl_4, id_4, av_4, sh_4\}$, $\{kl_5, id_5, av_5, sl_5\}$ must be active in a solution for the CSP defined by (STAT, WF, TRANS).

Second, we give an example for the construction of an *extensibility* rule related to the consistent path $p = [(q_0, \{c_2\}, q_1)]$.

Example 3 (Well-formedness rules for extensibility) WFextensibility $([(q_0, \{c_2\}, q_1)]) = \{\mathcal{ACT}(p_4) \vee \mathcal{ACT}(p_5) \vee \mathcal{ACT}(p_6) \vee \mathcal{ACT}(p_7)\}$.

This rule denotes that fact that at least one of the paths p_4, p_5, p_6, p_7 must be extensible in the state q_1, i.e., the corresponding variables must be active.

Third, we give an example for the construction of a well-formedness rule related to the *determinism* of the state q_1.

Example 4 (Well-formedness rules for determinism) WFdeterminism $(q_1) = \{(kl_7 \neq kl_4 \vee aw_7 \neq aw_4 \vee \mathcal{IACT}(p_4) \vee$

$\mathcal{IACT}(p_7)) \wedge (\mathrm{kl}_7 \neq \mathrm{kl}_5 \vee \mathrm{aw}_7 \neq \mathrm{aw}_5 \vee \mathcal{IACT}(p_5) \vee \mathcal{IACT}(p_7)) \wedge (\mathrm{kl}_7 \neq \mathrm{kl}_6 \vee \mathrm{aw}_7 \neq \mathrm{aw}_6 \vee \mathcal{IACT}(p_6) \vee \mathcal{IACT}(p_7))\}$.

This rule denotes the fact that there must exist an instantiation of the variables $\{\mathrm{kl}_7, \mathrm{kl}_6, \mathrm{kl}_5, \mathrm{kl}_4, \mathrm{aw}_7, \mathrm{aw}_6, \mathrm{aw}_5, \mathrm{aw}_4\}$ such that the instantiation of $\{\mathrm{kl}_7, \mathrm{aw}_7\}$ differs in at least one value from $\{\mathrm{kl}_6, \mathrm{aw}_6\}$, in at least one value from $\{\mathrm{kl}_5, \mathrm{aw}_5\}$, and in at least one value from $\{\mathrm{kl}_4, \mathrm{aw}_4\}$. Figure 7 depicts a simple example which demonstrates the application of this type of well-formedness rule. This rule allows to determine whether there exists an extension of a given PFSA which is well-formed regarding the determinism property. This is useful in situations where the diagnosis process assumes that a certain transition is faulty (in Fig. 7: $\{c_1, c_2\}$ are assumed to be faulty). If this is the case, we have to identify a substitution of the faulty transition conditions (constructive approach) which fulfills the given set of well-formedness rules. For our simple example of Fig. 7 we have to define the following determinism well-formedness rule: $(x_1 \neq x_2 \vee y_1 \neq y_2 \vee \mathcal{IACT}(p_1) \vee \mathcal{IACT}(p_2))$. In the case of Fig. 7(a) such an extension exists for the transition conditions $\{c_1, c_2\}$ (e.g., $\{c_1 : x = 1 \wedge y = 1, c_2 : x = 2 \wedge y = 2\}$), the case of Fig. 7(b) does not allow such an extension (the cardinality of the variable domains is too low) which means that no diagnosis can be found in this case.

An example for the definition of a transition condition (TRANS) is the following. We represent the transition condition c_1 of our example PFSA.

Example 5 (TRANS for PFSA) $\{c_1$: $(\mathrm{kl}_1 =$ beginner $\vee \mathcal{IACT}(\mathrm{kl}_1)) \wedge (\mathrm{kl}_2 =$ beginner$\vee \mathcal{IACT}(\mathrm{kl}_2)) \wedge (\mathrm{kl}_3 =$ beginner$\vee \mathcal{IACT}(\mathrm{kl}_3))\} \subseteq$ TRANS.

This generated condition for c_1 contains those variables which belong to paths including the transition $(q_0, \{c_1\}, q_2)$, i.e., the variables $\{\mathrm{kl}_1, \mathrm{kl}_2, \mathrm{kl}_3\}$ which belong to the paths $\{p_1, p_2, p_3\}$. For the CSP defined by (STAT, WF, TRANS) the possible values of the variables $\{\mathrm{kl}_1, \mathrm{kl}_2, \mathrm{kl}_3\}$ are defined by condition c_1, i.e., the value of $\{\mathrm{kl}_1, \mathrm{kl}_2, \mathrm{kl}_3\}$ must be *beginner* if the corresponding variable is active.

Given a specification of (SD, TRANS), a PFSA Diagnosis is defined as follows.

Definition 8 (PFSA diagnosis) A PFSA Diagnosis for a PFSA Diagnosis Problem (SD, TRANS) is a set $S \subseteq$ TRANS s.t. SD $\cup$ TRANS $-S$ is consistent.

Given a PFSA Diagnosis Problem (SD, TRANS), a Diagnosis S for (SD = STAT $\cup$ WF, TRANS) exists under the reasonable assumption that STAT $\cup$ WF is consistent. Assuming that SD = STAT $\cup$ WF is inconsistent, it follows from the definition of a diagnosis S that SD $\cup$ TRANS-S is inconsistent $\forall S \subseteq$ TRANS. Assuming that SD $\cup$ TRANS–S is consistent, it follows that STAT $\cup$ WF is consistent.

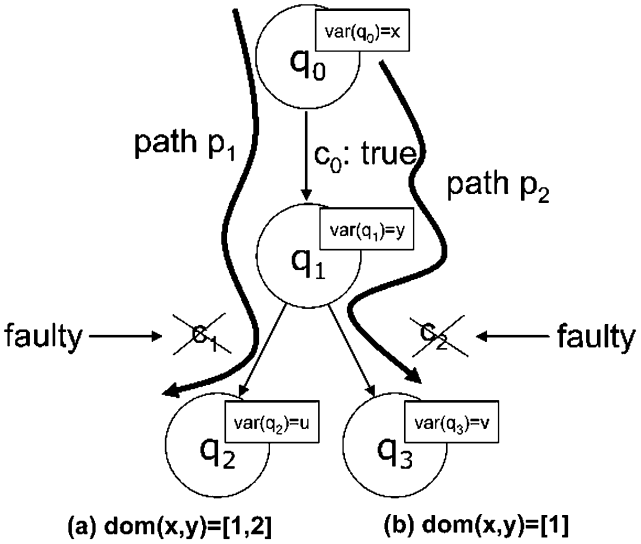

Fig. 7 Calculation of extensions for a PFSA (the transition conditions c_1, c_2 are assumed to be faulty): in case (**a**) an extension is possible (e.g., $\{c_1 : x = 1 \wedge y = 1, c_2 : x = 2 \wedge y = 2\}$), case (**b**) does not allow the calculation of an extension

The calculation of diagnoses is based on the concept of minimal conflict sets [16].

Definition 9 (Conflict set) A Conflict Set (CS) for (SD, TRANS) is a set $\{c_1, c_2, \ldots, c_n\} \subseteq$ TRANS, s.t. $\{c_1, c_2, \ldots, c_n\} \cup$ SD is inconsistent. CS is minimal iff $\neg\exists$ CS$' \subset$ CS : conflict set (CS$'$).

The following Algorithm 1 sketches of our approach to calculate a set of diagnoses for a given process definition. $\triangle$ is initialized with the empty set (no diagnosis has been found up to now). Thereafter SD and TRANS are initialized and the core diagnosis algorithm (PFSADIAG) is activated with the corresponding parameters. PFSADIAG activates the Theorem Prover (TP) (in our case a constraint solver) which checks whether SD $\cup$ TRANSH is inconsistent. TRANSH contains all elements of TRANS with the exception of those already selected for resolving a conflict. If TP(SD, TRANSH) does not detect an inconsistency, a corresponding diagnosis has been found. In this case, $\triangle$ is updated with the found diagnosis. Otherwise, an inconsistency still exists in SD $\cup$ TRANSH, i.e., further conflicts have to be resolved (recursive call of PFSADIAG). Note that CS represents a minimal conflict set which has been calculated by TP.

Algorithm 1 is an adapted version of the HSDAG (Hitting Set Directed Acyclic Graph—HSDAG) algorithm presented in [21]. Regarding our example, conflict sets determined by TP calls are: $\{c_2, c_{11}\}$, $\{c_7, c_9\}$, $\{c_2, c_9\}$, and $\{c_1, c_9\}$. The set of minimal (!) diagnoses calculated is $\triangle = \{\{c_1, c_2, c_7\},$

Algorithm 1 PFSA Diagnosis (SD, TRANS): Set of Diagnoses △.

```
△={};                                   /* △ will contain the diagnoses */
SD=GetSystemDescription();              /* initialize SD */
TRANS=GetTransitionConditions();        /* initialize TRANS */
PFSADIAG(SD, TRANS);                    /* calculate diagnoses */
Return △;                               /* return found diagnoses */

PFSADIAG(SD, TRANSH)
{
 CS=TP(SD,TRANSH);                      /* SD ∪ TRANSH inconsistent? */
 if (IsEmpty(CS))                       /* not inconsistent */
  {△=△∪(TRANS-TRANSH)}                  /* diagnosis found; store it in △ */
 else                                   /* inconsistency detected */
 {foreach x in CS                       /* for each factor in conflict */
  PFSADIAG(SD, TRANSH - {x})            /* resolve conflict and search for the diagnosis */
 }
}
```

$\{c_2, c_9\}, \{c_{11}, c_9\}\}$. One minimal diagnosis S for our example PFSA is $\{c_2, c_9\}$, i.e., $\{c_2, c_9\}$ has to be changed in order to make the PFSA consistent regarding the given set of well-formedness rules.

5 Analysis

Applicability for interactive settings. The process flow diagnosis concepts presented in this paper have been implemented as a component of the knowledge-based recommender development environment presented in [3, 9]. Our approach complements the knowledge acquisition interface with a set of intelligent mechanisms allowing the automated identification of faulty transition conditions in process definitions, i.e., does not fundamentally change the structure of the knowledge acquisition interface. Typically, process definitions include between 5 and 40 transition conditions (see Table 1). Furthermore, faulty PFSA definitions exhibit about 2–10 conflicts.[4] Table 1 clearly shows that even for more complex settings with about 10 conflicts and 40 transitions in the PFSA, our algorithm is able to calculate the corresponding diagnoses within a couple of seconds. This result is extremely important since it clearly shows that our approach is applicable to interactive knowledge acquisition settings.

Figure 8 depicts a screenshot of our automated debugging environment for recommender user interface descriptions. This environment has been developed as part of our knowledge acquisition and test environment for knowledge-based recommender applications within the scope of the Koba4MS[5] project. One diagnosis is presented at a time; the user can browse through all the existing diagnoses which are displayed with a corresponding highlighted transition condition. By clicking on a certain faulty transition condition, the debugging environment presents a list of explanations as to why a certain transition condition is faulty, e.g., transition condition c_2 in our example has to be changed in order to make the transition condition $[q_4, q_5]$ accessible for at least one of the paths leading to $q4$.

Analysis approach. We have conducted an experiment with the goal to highlight and quantify potential reductions of development and maintenance efforts facilitated by our process flow debugging environment. For the experiment we have defined three example (faulty) process definitions $(\text{pd}_1, \text{pd}_2, \text{pd}_3)$ with an increasing complexity regarding the number of states and transition conditions (see Table 2). On the basis of these process definitions, participants of the experiment had to identify solutions for the following types of tasks:

1. Diagnosis task (d)—identify a minimum cardinality set of faulty transition conditions (without an automated debugging support): participants had to provide an answer to the question which minimum cardinality set S of faulty transition conditions has to be removed from the set of given transition conditions in the PFSA, such that the transition conditions in the resulting PFSA are consistent with the well-formedness rules (SD ∪ TRANS − S has to be consistent).

[4] Conflicts of this magnitude occur in settings where knowledge engineers develop recommender process definitions who have a basic understanding of the well-formedness properties discussed in this paper.

[5] Knowledge-based Advisors for Marketing and Sales—FFG-808479, supported by the Austrian Research Fund.

Table 1 Performance of PFSA Diagnosis (in secs). The performance evaluation results clearly show the applicability of the debugging algorithm for interactive settings (for typical problem sizes of recommender applications)

Transitions	2 conflicts	3 conflicts	4 conflicts	5 conflicts	10 conflicts
5	0.05	0.071	0.12	0.16	–
10	0.141	0.171	0.251	0.321	0.36
20	0.401	0.621	1.152	1.343	2.268
30	0.891	1.092	1.421	1.692	2.826
40	1.352	1.791	2.096	2.813	3.912

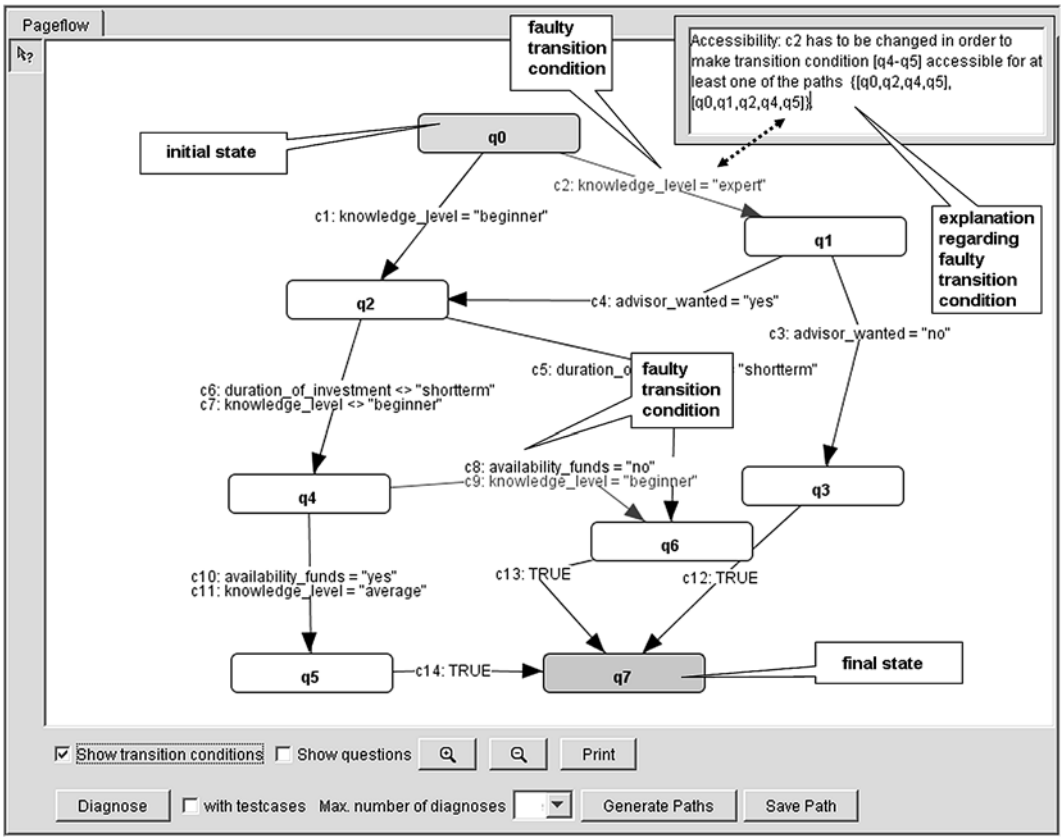

Fig. 8 Process Designer User Interface ($\{c_2, c_9\}$ are indicated as faulty)

2. Repair task (r)—select consistent repair actions: the participants had to select a maximum set of consistent repair actions T out of a proposed set of (partially faulty) repair actions, s.t. SD $\cup$ TRANS $S \cup T$ consistent.

The participants of the experiment were interacting with an online questionnaire where each participant had to solve the given tasks autonomously. The time efforts needed to complete a given task were stored in an underlying knowledge base. Participants were randomly assigned to one of the two testgroups shown in Table 2. For each process definition, the members of one testgroup had to solve a diagnosis and repair task whereas the members of the other testgroup had only to solve a corresponding repair task. Following this approach, we were able to compare process definition maintenance efforts with and without a corresponding debugging support.

The participants (Computer Science students at the Klagenfurt University) of the study ($n = 40$) had knowledge engineering experiences in the development of recommender applications. The types of error identification tasks which had to be solved within the scope of the experiment were similar to those tasks knowledge engineers have to solve within the scope of commercial projects. This means that process definitions from commercial projects have been simplified (see the number of variables and transitions in Table 2) and then provided to the participants of the study. The similarity of the participants' education level as well as the similarity of the posed error identification tasks to real-

Table 2 Assignment of error identification and repair tasks for process definitions (pd) to test groups (dr = manual diagnosis and repair, r = automated diagnosis and manual repair), e.g., testgroup_1 had a diagnosis and repair task for process definition pd_1. Each example process definition is additionally characterized by the number of variables and the number of transitions, e.g., $\text{pd}_1 : v/t = 6/6$ is process definition 1 with 6 variables and 6 related transitions

	$\text{pd}_1 : v/t = 6/6$	$\text{pd}_2 : v/t = 8/9$	$\text{pd}_3 : v/t = 10/11$
$\text{testgroup}_1 (n = 20)$	pd_{1dr}	pd_{2r}	pd_{3dr}
$\text{testgroup}_2 (n = 20)$	pd_{1r}	pd_{2dr}	pd_{3r}

Table 3 Reduced error detection/repair times with debugging support ($mean_r$) compared to error detection/repair times without debugging support ($mean_{dr}$). A (two sample) t-test has been applied to the datasets (times needed for completing the tasks); results show a significantly improved performance due to debugging support

Process definition	mean_{dr} (s)	t-score	p	mean_r (s)
pd_1	110.750	2.263	0.034	71.458
pd_2	155.417	2.277	0.033	78.417
pd_3	246.167	3.308	0.003	72.458

world settings clearly show the external validity (similarity-based) of the results of our experiment. The participants of the study had to solve the given error identification and repair tasks autonomously. For the comparison of time efforts related to diagnosis and repair tasks, we applied an independent (two-sample) t-test (parametric statistical test—see Table 3), which is applicable since the error identification and repair times are normally distributed and the effort data sets for the two testgroups are independent.

Results. Our experiment clearly shows the applicability of our debugging approach in terms of time savings related to development and maintenance processes (see Table 3). The goal of our analysis was to investigate differences in time efforts related to the identification and repair of faulty process definitions depending on whether a corresponding automated debugging support was available or not.

Hypothesis: *Automated debugging support for process definitions leads to significant time savings in the detection and repair of faulty transition conditions*.

The average *repair effort* for process definitions (when automated debugging support has been provided) pd_1 was 71.458 seconds (std. dev. 28.915 seconds), the corresponding average *diagnosis and repair* effort (no debugging support provided) was 110.750 seconds (std. dev. 63.704 seconds). Similar results have been obtained by examining the remaining example (faulty) process definitions which allows us to accept the defined hypothesis (see Table 3). A further interesting aspect resulting from our study is that the time savings successively increase with the complexity of the given process definition.

Next steps. The presented debugging concepts are a first approach to make development and maintenance of recommender process definitions more effective. Due to the feedback from knowledge engineers and domain experts, further concepts will be integrated into future versions of our software. Domain experts (as well as knowledge engineers) often tend to think in terms of examples. We intend to integrate example-driven *testing mechanisms* where the developer himself provides an example set of paths which should be accessible. On the technical level such examples represent additional well-formedness rules for a given process definition (specific type of accessibility well-formedness rule). In many cases there exist a number of *alternative diagnoses* explaining the sources of inconsistencies in a given process definition. In this context we will include additional ranking mechanisms for diagnoses which take into account the probability of a transition condition to be faulty. Currently, the selection of a diagnosis strictly depends on its cardinality, i.e., diagnoses with the lowest number of transition conditions are presented first.

6 Related work

Collaborative filtering [13], content-based filtering [20] and knowledge-based recommendation [1, 3, 30] are the three basic approaches to the implementation of a recommender application. Collaborative Filtering is based on the assumption that customer preferences are correlated, i.e., similar products are recommended to customers with similar interest profiles. Content-based filtering focuses on the analysis of a given set of products already ordered by a customer. Based on this information, products are recommended which resemble products already ordered (products related to similar categories). Using knowledge-based approaches, the relationship between customer requirements and offered products is explicitly modeled [7]—compared to collaborative and content-based filtering, knowledge-based

approaches do exploit deep knowledge about the application domain.

Such knowledge representations are the major precondition for the application of model-based diagnosis techniques [12, 21]. An overview of the application of model-based diagnosis techniques in software debugging can be found in [28]. The complexity of configuration knowledge bases motivated the application of model-based diagnosis (MBD) [21] in knowledge-based systems development [6]. Similar motivations led to the application of model-based diagnosis in technical domains such as the development of hardware designs [11], onboard diagnosis for automotive systems [24] and in software development [18]. The work presented in this paper has a special relationship to the work we presented in [6]. Reference [6] focus on the identification of faults in configuration knowledge bases, where a set of test cases is used to induce conflicts with a configuration knowledge base. In contrast to this work, we provide an abstract representation of finite state models which is checked against a set of well-formedness rules, i.e., well-formedness rules correspond to test cases presented in [6]. Test cases used in [6] are, e.g., configurations calculated by previous versions of configuration knowledge bases. In many domains, such test cases have to be defined by domain experts which makes testing and debugging a time-consuming task. Although our debugging approach for recommender process definitions allows the specification of test cases as well (input sequences which have to be accepted by the process definition), one of the major strengths of the approach is that well-formedness rules (generic test cases) can be automatically derived from given process definitions. Compared to the work presented in this paper, [4] focuses on the analysis recommender knowledge bases which requires completely different algorithms and datastructures.

The representation of recommender processes in the form of finite state representations is discussed in [3, 10]. This approach is novel in the context of developing knowledge-based recommender applications and due to its formal basis it allows a direct and automated translation of the graphical model into a corresponding recommender application. The automated debugging of such process definitions on the basis of MBD [21] has so far not been discussed in the literature. Related work can be found, e.g., in [17], where an algorithm for checking the consistency of workflow definitions is presented. Compared to our work, [17] focuses on assuring workflow properties such as *each component of the workflow has at least one output parameter* or *all components of the workflow are executable*. Compared to these basic types of consistency checks, our work provides intelligent mechanisms that effectively support the automated indication of potential sources of inconsistencies. Our approach clearly focuses on the representation of states and the transition conditions between those states—in this context, the possible actions are limited to the specification of values for input variables (questions). An additional action semantics would further extend the applicability of our work to scenarios—we define this as a topic of future work.

A number of studies focused on the analysis of existing Knowledge Acquisition (KA) environments. A detailed overview on those approaches can be found in [29]. All those studies focused on user behavior when interacting with a certain knowledge acquisition environment. Examples for tested hypotheses in these experiments are: *all users would employ the same set of commands even if told nothing in advance about the modeling environment*; *users will make less mistakes during KA tasks using KA tools*; or *users will be able to complete a KA task in less time using KA tools*. Compared to these evaluations, the experiment of this paper focuses on a very specific aspect related to the identification of faults in recommender user interface descriptions.

7 Conclusions and future work

Automated debugging support for the design of recommender user interfaces can significantly reduce related development and maintenance efforts. In this paper we have presented concepts supporting the identification of minimal sets of faulty transition conditions in finite state models of recommender user interfaces. Although this paper focused on the development of recommender user interfaces, the presented approach is not restricted to this domain but is generally applicable to settings where a finite state model of a user interface is given. The proposed approach has been implemented as part of a commercially available recommender development environment.

References

1. Burke R (2000) Knowledge-based recommender systems. Encycl Libr Inf Syst 69(32)
2. Burke R (2002) Hybrid recommender systems: survey and experiments. User Model User-Adapted Interact 12(4):331–370
3. Felfernig A (2005) Koba4MS: selling complex products and services using knowledge-based recommender technologies. In: Müller G, Lin K (eds) 7th IEEE international conference on e-commerce technology (CEC'05), Munich, Germany, pp 92–100
4. Felfernig A (2007) Reducing development and maintenance efforts for web-based recommender applications. Int J Web Eng Technol 313:329–351
5. Felfernig A, Friedrich G, Jannach D, Zanker M (2006) An integrated environment for the development of knowledge-based recommender applications. Int J Electron Commer 11(2):11–34
6. Felfernig A, Friedrich G, Jannach D, Stumptner M (2004) Consistency-based diagnosis of configuration knowledge bases. Artif Intell 2(152):213–234
7. Felfernig A, Friedrich G, Jannach D, Stumptner M, Zanker M (2003) Configuration knowledge representations for semantic web applications. AI Eng Des Anal Manuf J 17:31–50

8. Felfernig A, Isak K, Russ C (2006) Knowledge-based recommendation: technologies and experiences from projects. In: Brewka G, Coradeschi S, Perini A, Traverso P (eds) 17th European conference on artificial intelligence (ECAI06), Riva del Garda, Italy, pp 632–636
9. Felfernig A, Kiener A (2005) Knowledge-based interactive selling of financial services using FSAdvisor. In: 17th innovative applications of artificial intelligence conference (IAAI'05), Pittsburgh, PA, pp 1475–1482
10. Felfernig A, Shchekotykhin K (2005) Debugging user interface descriptions of knowledge-based recommender applications. In: Workshop notes of the IJCAI'05 workshop on configuration, Edinburgh, Scottland, pp 13–18
11. Friedrich G, Stumptner M, Wotawa F (1999) Model-based diagnosis of hardware designs. AI J 111(2):3–39
12. Greiner R, Smith B, Wilkerson R (1989) A correction to the algorithm in Reiter's theory of diagnosis. Artif Intell 41(1):79–88
13. Herlocker JL, Konstan JA, Terveen LG, Riedl J (2004) Evaluating collaborative filtering recommender systems. ACM Trans Inf Syst 22(1):5–53
14. Hopcroft J, Ullman J (1979) Introduction to automata theory, languages and computation. Addison-Wesley, Massachusetts
15. Jiang B, Wang W, Benbasat I (2005) Multimedia-based interactive advising technology for online consumer decision support. Commun ACM 48(9):93–98
16. Junker U (2004) QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems. In: 19th national conference on AI (AAAI04), pp 167–172
17. Kim J, Spraragen M, Gil Y (2004) An intelligent assistant for interactive workflow composition. In: International conference on intelligent user interfaces (IUI-2004), Madeira, Portugal, pp 125–131
18. Stumptner MM, Wotawa F (2000) Modeling Java programs for diagnosis. In: 14th European conference on artificial intelligence, Berlin, Germany, pp 171–175
19. Montaner M, Lopez B, De la Rose J (2003) A taxonomy of recommender agents on the Internet. Artif Intell Rev 19:285–330
20. Pazzani M (1999) A framework for collaborative, content-based and demographic filtering. Artif Intell Rev 13(5–6):393–408
21. Reiter R (1987) A theory of diagnosis from first principles. Artif Intell 23(1):57–95
22. Resnick P, Iacovou N, Suchak M, Bergstrom P, Riedl J (1994) GroupLens: an open architecture for collaborative filtering of netnews. In: ACM conference on computer supported cooperative work, pp 175–186
23. Ricci F, Venturini A, Cavada D, Mirzadeh N, Blaas D, Nones M (2003) Product recommendation with interactive query management and twofold similarity. In: 5th international conference on case-based reasoning (ICCBR 2003), Trondheim, Norway, pp 479–493
24. Sachenbacher M, Struss Pr, Carlen CM (2000) A prototype for model-based on-board diagnosis of automotive systems. AI Commun 13(2):83–97
25. Mittal S, Falkenhainer B (1990) Dynamic constraint satisfaction problems. In: 8th national conference on artificial intelligence, Detroit, MI. MIT Press, Cambridge, pp 25–32
26. Smyth B, Balfe E, Boydell O, Bradley K, Briggs P, Coyle M, Freyne J (2005) A live user evaluation of collaborative web search. In: 19th international joint conference on artificial intelligence, Edinburgh, Scotland, pp 1419–1424
27. Stolze M, Field S, Kleijer P (2000) Combining configuration and evaluation mechanisms to support to selection of modular insurance products. In: 8th European conference on information systems, pp 858–865
28. Stumptner M, Wotawa F (1998) A survey of intelligent debugging. Eur J Artif Intell 11(1):35–51
29. Tallis M, Kim YG (1999) User studies of knowledge acquisition tools: methodology and lessons learned. In: KAW-99
30. Thompson C, Göker M, Langley P (2004) A personalized system for conversational recommendations. J Artif Intell Res 21:393–428
31. Tsang E (1993) Foundations of constraint satisfaction. Academic Press, London
32. VanNoord G, Gerdemann D (2004) Finite state transducers with predicates and identities. Grammars 4(3):263–286