2011 23rd IEEE International Conference on Tools with Artificial Intelligence

A formal approach to personalization

Georges Dubus, Fabrice Popineau, Yolaine Bourda
SUPELEC Systems Sciences (E3S) - Computer Science Department
Gif sur Yvette, France
Email: georges.dubus@supelec.fr, fabrice.popineau@supelec.fr, yolaine.bourda@supelec.fr

***Abstract*—Personalized systems are a response to the increasing number of resources on the Internet. In order to facilitate their design and creation, we aim at formalizing them. In this paper, we consider the relationship between a personalized application and its non-personalized counterpart. We argue that a personalized application is a formal extension of a non-personalized one. We aim at characterizing the syntactic differences between the expression of the personalized and non-personalized versions of the application. Situation calculus is our framework to formalize applications. We introduce two scenarios of non-personalized application that we personalize to illustrate our approach.**

***Keywords*-situation calculus; personalization; web application; adaptive systems; weaving;**

I. Introduction

The Internet is quickly becoming the main way of communicating and sharing information. The amount of content available online has increased exponentially. A single website often contains so much information that a user can't grasp it all, and may even be unable to navigate to find the information he wants. Personalized systems – systems that behave differently depending on the user – emerged to provide guidance to users in such cases.

Different types of personalized systems exist to answer different problems. Adaptive hypermedia guide the navigation in a set of resources, recommender systems offer items to the user that are likely to interest him, and personalized information retrieval systems provide the user with answers to his queries that are tailored to his needs. These different systems have distinct inner workings and distinct logical models, if any. Some systems cross the borders between those types. For example, an e-learning web application may provide an adaptive hypermedia with learning resources, and also recommendations of external resources, such as a book to read to go further in a subject.

Personalized systems are not as widely used as they should be because their creation is a difficult process [1]. There are tools to help with the creation of some of the systems (adaptive hypermedia, recommender systems), but they are dedicated to one single system, and there are no general tools covering all the personalized systems. We aim at defining a general formalization of personalized systems. This will enable us to create tools to help with the design of personalized systems and to prove that such systems behave correctly.

To define that formalization, we consider the idea that a personalized application can be viewed as an extension of a non-personalized application. We consider applications whose behaviour can be expressed with the situation calculus, and show that the personalization can be expressed by a syntactical extension of the equations describing the application. We illustrate this idea using two scenarios of applications to which we add personalization.

Section 2 presents the idea that the personalization can be expressed as an extension of an application. Section 3 explains the formalism we use. Section 4 illustrates our proposal using the formalism with two examples. Section 5 concludes.

II. Personalization

A. Personalization

A personalized application is an application that reflects some features of the user in a user model and apply this model to adapt various aspects of the application to the user.

When no profile is available for a user (for example an unauthenticated user), the application behaves in the same way as with any other unknown user: it behaves like a non-personalized application. The dual of that remark is that the personalized application is the result of adding personalization to a non personalized behaviour.

This approach of splitting the personalized application in various parts may ease the design of such personalized applications. Our goal is to show those personalized applications can be built as extensions of non-personalized applications on a formal level.

B. Approach

To deliver personalized information to a user is a complex task that may require a good amount of intelligence. We build our view of the problem in the context of artificial intelligence rational agents as presented by [2]. For that reason, we base our approach on a logical formalism. Expressing the applications in a logical formalism will

1082-3409/11 $26.00 © 2011 IEEE
DOI 10.1109/ICTAI.2011.43

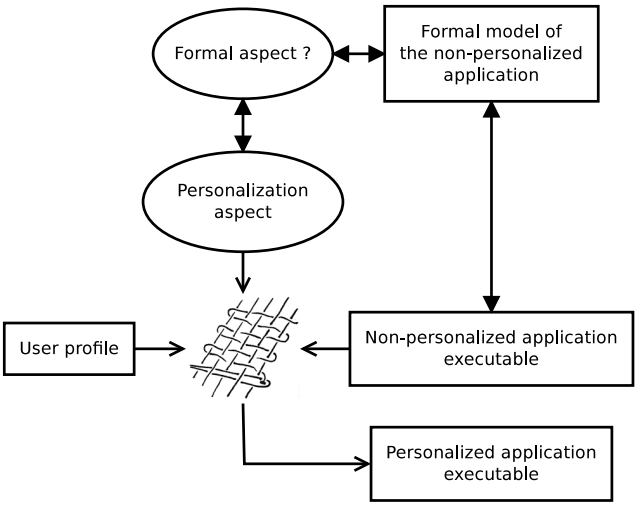

Figure 1. Weaving the personalization

also allow us to highlight syntactically the component that is the personalization.

It's not obvious that any personalized application can be expressed in a logical formalism. At first, we will only consider applications that can be expressed logically. Past works have shown that logical formalisms are relevant for the study of personalized applications [3], [4].

The idea of combining a behaviour with a component is present in the aspect-oriented programming paradigm [5]. In our approach, the behaviour of the application is defined in a formalism such as Golog, and compiled into an executable. The operation of adding personalization can be seen as weaving[1] the non-personalized application with the user profile using a personalization aspect to get the personalized application, as shown in figure 1. This means changing the execution of the non-personalized executable by adding or changing some behaviours based on the user profile.

Our ultimate goal is to find an equivalent of the weaving at the formal level. As in the aspect-oriented paradigm, we would like to weave the formal model of the non-personalized application to generate a formal model of the personalized application.

C. Advantages

This approach opens the way to the creation of automatic or semi-automatic tools for the creation of personalized applications. If we extract the syntactic difference between a non-personalized application and its personalized counterpart, then we'll be able to create a tool transforming a non-personalized application into a personalized one, or at least easing the transformation.

Furthermore, the use of a logical formalism for the characterization of the behaviour of the application enables the proof of various properties of the application.

[1]Weaving is the term used by the aspect-oriented community to designate the action of modifying the behaviour of a program using aspects (isolated representations of significant concepts in a program).

III. Explanation of the formalism

A. The choice of the situation calculus

We want to model the behaviour of a web application interacting with one or many users. For that, we need a formalism able to express the behaviour of the application from its point of view. This would enable us to implement the application using its model. We would also like a formalism that allows to prove the validity of the model.

We choose the situation calculus. This formalism models agents interacting with their environment and acting upon it. It has been used to express agents interacting on the web [6], [7] or to express a hypertext system [8]. It has also been used previously to express the personalization of adaptive hypermedia [3]. We want to explore its use to express the personalization in other types of personalized applications. Because the model is defined logically, it allows to prove all kind of properties on the modelled applications. For example, it enables the proof that there will always be a response from the application, whatever the user does.

B. Situation calculus

The situation calculus [9] is a first order language designed to represent changing worlds. All changes are the result of primitive *actions*. Actions may take parameters. For example, $Drop(x)$ is the action of dropping an object x. We denote the *initial situation*, in which no action has happened yet, by s_0. All changes are the results of applying a sequence of primitive actions to the initial situation. The result of the application of an action α to a *situation* s is the situation $do(\alpha, s)$. Both actions and situations are first order terms.

An introduction to the situation calculus and the extensions we use here can be found in [10].

To fulfill the need to take user response into account, we use guarded action theories [11]. In a guarded action theory, there is a finite number of *sensing functions* (or *sensor functions*), which are unary functions whose argument is a situation that model a sensor available to the agent. These functions are used to define two sets of axioms which specify an action theory:

- Guarded successor state axioms, of the form

$$\alpha(\vec{x}, a, s) \implies [F(\vec{x}, do(a, s)) \equiv \gamma(\vec{x}, a, s)] \quad (1)$$

where α is a fluent formula, F is a relational fluent and γ is a fluent formula. These axioms enable

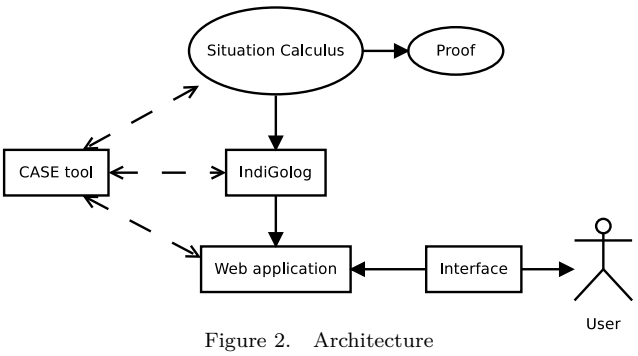

Figure 2. Architecture

regression, i.e. deriving the value of a fluent in a situation from the value in previous situations.

- Guarded sensed fluent axioms, of the form

$$\beta(\vec{x}, s) \implies [F(\vec{x}, s) \equiv \rho(\vec{x}, s)] \quad (2)$$

where β is a sensor-fluent formula, F is a relational fluent and ρ is a sensor formula. These axioms enable sensing the value of an axiom in a certain situation, by using *sensing functions*.

The guards α and β are used to specify when the regression and sensing formulas are usable. This enables us to have fluents that can be sensed in certain situation, and can be modified by actions in other situations.

C. IndiGolog

IndiGolog [12], [13] (extension of Golog [14] and ConGolog [15]) is a high level logic programming language in which actions of a guarded action theory of the situation calculus are instructions. It enables the construction of complex programs by assembling the primitive actions of the situation calculus with various constructs. This language has been used to program robots [15] or agents of the semantic web [6].

D. Architecture

In our framework, the application is composed of an agent in IndiGolog that interacts with users through an interface that abstracts the interaction by providing sensor functions to the agent.

Figure 2 depicts our architecture: on the theoretical level, the action theory defined using the situation calculus enables us to prove some properties of the systems. This action theory is implemented using IndiGolog on the second level. Our web application is expressed using this implementation and interacts with the user. A CASE tool helps with the design of the personalized system.

The aim of the CASE tool is to help the creation of the action theory and its IndiGolog implementation, and to help the addition of personalization to the non-personalized application.

IV. Illustration

We illustrate our proposal by modeling two scenarios using situation calculus, and adding personalization to the modeled applications.

We choose different kinds of personalized systems to build our scenarios: a recommender system and an adaptive educational game.

A. News aggregator

1) Presentation: We consider a simplified scenario of a news aggregator. The system proposes news articles to the user. For each article, the user states whether he read it and liked it, or ignored it. The system then uses this information to recommend the user articles he may like. This recommendation can be based on various factors. For example, the system might recommend articles that are related to articles the user liked.

This scenario can be extended to a full fledged application where news articles sources are handled, recommended, banned and so on.

2) Model: This application handles news articles. The relation $article(x)$ denote that x is a news article. The relation $related(x_1, x_2)$ denotes that two articles are related to each other.

There are 3 fluents: $presented(x, s)$, $liked(x, s)$ and $ignored(x, s)$ which denotes that an article has been presented to, liked by or ignored by the user.

There is one primitive action: $present(x)$ (the agent presents an article to the user).

In the initial situation, nothing has been presented to the user:

$$\forall x.article(x) \implies \neg presented(x, s_0) \wedge \neg liked(x, s_0) \wedge \neg ignored(x, s_0) \quad (3)$$

We only present articles which have never been presented. The fluent *presented* is set to true by the action of presenting.

$$Poss(present(x), s) \equiv \neg presented(x, s) \quad (4)$$

$$True \implies [presented(x, do(a, s)) \equiv a = present(x) \vee presented(x, s)] \quad (5)$$

There are three sensor functions:

- **likedLastArticle**(s) tells if the user has liked an article by calling an external function.
- **passedLastArticle**(s) tells if the user has passed an article by calling an external function.
- **lastArticle**(s) returns the last presented article.

$$\textbf{lastArticle}(s) = x \implies liked(x, s) \equiv \textbf{likedLastArticle}(s) \quad (6)$$

$$\textbf{lastArticle}(s) = x \implies passed(x, s) \equiv \textbf{passedLastArticle}(s) \quad (7)$$

Also, we know that these fluents can change only when the article is proposed to the user. There is a similar axiom for the fluent $passed(x, s)$.

$$a \neq present(x) \implies liked(x, do(a, s)) \equiv liked(x, s) \quad (8)$$

We also introduce the fluent $recommended(x, s)$ that denotes an article is recommended for the user. A possible recommendation strategy could be to recommend articles that are related to articles that the user liked.

$$recommended(x, s) \equiv \exists x'.related(x, x') \wedge liked(x', s) \quad (9)$$

The behaviour of the application expressed in Golog is:

$$(\Sigma(\pi(x).[recommended(x)?;\ present(x)]))^* \rangle\rangle \quad (\pi(x).present(x))^* \quad (10)$$

The first process, which presents recommended articles to the user, has the highest priority. This ensures that only recommended items are presented, as long as there are some. When there are no more recommended articles to present, the other process sends random articles to the user until a new item is recommended.

The first process uses the search operator Σ to make sure only recommended articles are chosen in the nondeterministic choice. No search operator is used in the second process, so the article to present is chosen arbitrarily (possibly randomly).

It is possible to prove that this application will not stop presenting articles before all the articles are read.

3) Personalization: We personalize this application by taking into account the subjects that interest the user. For example, the user has previously been asked what topics he's interested in, and the topics are stored in his profile. This profile is accessible to the agent through the fluent $interestedIn(y, s)$ which denotes that the user is interested in the subject y. We also introduce a fluent $dealsWith(x, y, s)$ which denotes an article x deals with the subject y.

The user interacting with the personalized application will get the same recommendations as with the non-personalized application. In addition to these recommendations, he will also be recommended articles that deal with subject he's interested in. For that, the definition of $recommended(x, s)$ is changed to:

$$recommended(x, s) \equiv \exists x'\ (related(x, x') \wedge liked(x', s)) \underline{\vee \exists y\ interestedIn(y, s) \wedge dealsWith(x, y, s)} \quad (11)$$

The changes introduced by the personalization are not necessarily monotonic. For example, a user could ask not to see articles from a specific author. The personalization would then remove articles from the recommendation pool instead of adding them. The addition to the equation (9) would then be:

$$\wedge \exists y\ author(y, x) \wedge \neg dislikedAuthor(y, s) \quad (12)$$

Further personalization could include using the articles the user like to update his profile.

B. Educative game for children

1) Presentation: We consider a simplified version of a game to teach vocabulary to children. A picture and four words are presented to a child. He as to chose the word corresponding to the picture. Pictures of the same concept are presented again until the concept is known by the child. The concepts presented to the child are selected randomly among those unknown to him. The difficulty is adapted depending on the child's results. The use case is as follows:

- A picture and four words are presented to the child.
  - If the child selects the wrong word: the error is shown with an image of the selected word.
  - If the child selects the right word: a congratulation message is displayed, and the success is remembered in order to know that the child knows the word.
- The system asks another question (a picture and four words).

2) Model: This application handles concepts that we want a user to learn. They are denoted by the relation $concept(x)$.

The fluents are:

- $proposed(x, s)$ — a concept is proposed to the user as an alternative in the current question.
- $rightanswer(x, s)$ — the concept is the right answer to the current question.
- $known(x, s)$ — the concept is known to the user.

The primitive actions are:

- $addWrongAnswer(x)$ — the application adds a wrong answer to the current question.
- $addRightAnswer(x)$ — the application adds a right answer to the current question.
- ask — the prepared question is asked to the user. This is only possible if a right answer has been added to the question.

There are also two sensor functions:

- **lastRightAnswer** returns the right answer to the last asked question.
- **correct** states whether the user answer correctly to the last question.

The preconditions axioms are:

$$Poss(ask, s) \equiv \exists x.rightanswer(x, s) \quad (13)$$

$$Poss(addWrongAnswer(x), s) \equiv \neg proposed(x, s) \wedge \neg known(x, s) \quad (14)$$

$$Poss(addRightAnswer(x), s) \equiv \neg proposed(x, s) \wedge \neg known(x, s) \wedge \neg(\exists x')rightanswer(x', s) \quad (15)$$

The guarded successor state axioms are:

$$True \implies [proposed(x, do(a, s)) \equiv a = addWrongAnswer(x) \vee a = addRightAnswer(x) \vee proposed(x, s) \wedge \neg a = ask] \quad (16)$$

$$True \implies [rightanswer(x, do(a, s)) \equiv \vee a = addRightAnswer(x) \vee rightanswer(x, s) \wedge \neg a = ask] \quad (17)$$

$$a \neq ask \vee \neg rightanswer(x) \implies [known(x, do(a, s)) \equiv known(x, s)] \quad (18)$$

The guarded sensed fluents axioms:

$$\mathbf{lastRightAnswer}(s) = x \implies [known(x, s) \equiv \mathbf{correct}(s)] \quad (19)$$

The Golog program expressing the behaviour of the application is:

$$\begin{array}{l}\textbf{proc } addOneWrongAnswer() \\ \quad \Sigma(\pi(x, addWrongAnswer(x))) \\ \textbf{endProc}\end{array} \quad (20)$$

$$\begin{array}{l}\textbf{proc } addOneRightAnswer() \\ \quad \Sigma(\pi(x, addRightAnswer(x))) \\ \textbf{endProc}\end{array} \quad (21)$$

$$\begin{array}{l}\textbf{proc } prepareQuestion() \\ \quad addOneWrongAnswer; addOneWrongAnswer; \\ \quad addOneWrongAnswer; addOneRightAnswer \\ \textbf{endProc}\end{array} \quad (22)$$

$$(prepareQuestion; ask)* \quad (23)$$

The program defines three procedures:

- *addOneWrongAnswer* adds a wrong answer by selecting one possible wrong answer.
- *addOneRightAnswer* adds a right answer by selecting one possible right answer.
- *prepareQuestion* adds three wrong answer and one right answer in order to prepare a question.

The main program consists only of: prepare a question, ask it and repeat.

3) Personalization: We add personalization two this scenario in two different ways:

- We change the number of possible answers depending on how quickly the user answers.
- We define a notion of difficulty for the questions, and change the difficulty depending on how quickly the user answers.

Firstly, we need to define what a quick answer is. For that, we introduce the sensor function **lastAnswerTime**(s) which returns the time the user took to answer the last question. We decide a user answered quickly to a question if he answered correctly in less than 10 seconds. The fluent $quickAnswer(x, s)$ denotes that the user answered quickly to the question whose right answer is x.

$$\mathbf{lastRightAnswer}(s) = x \wedge correct(s) \implies quickAnswer(x, s) \equiv \mathbf{lastAnswerTime}(s) < 10 \quad (24)$$

$$a \neq ask \vee \neg rightanswer(x) \implies quickAnswer(x, do(a, s)) \equiv quickAnswer(x, s) \quad (25)$$

We also introduce a functional fluent $totalQuickAnswer(s)$. $totalQuickAnswer(s) = n$ that denotes that n different questions have been answered quickly (there are n different x such that $quickAnswer(x, s)$)[2].

Using that fluent, we can define a first approach of personalization for this application: changing the number of possible response depending on how quickly the user answered so far. Here, we will add one possible answer each 10 fast answered questions.

We first need to re-define the *prepareQuestion* procedure to create a question with a variable amount of answers, then modify the main program to use it. $prepareQuestion(n)$ prepares a question with n wrong answers.

$$\begin{array}{l}\textbf{proc } prepareQuestion(n) \\ \quad \textbf{if } n = 0 \textbf{ then } addOneRightAnswer \\ \quad \textbf{else } addOneWrongAnswer; \\ \quad\quad prepareQuestion(n - 1) \\ \textbf{endProc}\end{array} \quad (26)$$

$$(prepareQuestion(3); ask)* \quad (27)$$

The personalized application's main program is:

$$(prepareQuestion(3 + totalQuickAnswer()/10); ask)* \quad (28)$$

Our second approach of personalization requires the question to have a difficulty. Each concept must have a difficulty. The fluent $difficulty(x, n)$ denotes that the concept x have the difficulty y, where n is a number. Any concept chosen for a question must be easier than the current difficulty. The current difficulty is defined from the number of quick answers so far.

$$currentDifficulty(s) = totalQuickAnswer(s)/10 \quad (29)$$

The precondition axioms for *addWrongAnswer* (14) and *addRightAnswer* (15) change to become:

$$Poss(addWrongAnswer(x), s) \equiv \neg proposed(x, s) \wedge \neg known(x, s) \wedge \exists n[difficulty(x, n) \wedge n < currentDifficulty(s)] \quad (30)$$

[2]The actual definition of $totalQuickAnswer(s)$ is left for now because defining "There exists n different terms" with a variable n can be tricky using only first-order logic, and is much easier to implement in a language such as prolog than to define.

$$Poss(addRightAnswer(x), s) \equiv \neg proposed(x, s) \wedge \neg known(x, s) \wedge \neg(\exists x')rightanswer(x', s) \wedge \underline{\exists n[difficulty(x, n) \wedge n < currentDifficulty(s)]} \quad (31)$$

C. Analysis

We have modeled two scenarios of applications and added personalization to them. The personalization was easily added by adding the relevant fluents (mostly representing the user profile) and modifying the equations to use those fluents.

Furthermore, we can see that the modifications of the application belong in one of two categories:

- Modifications in order to add flexibility while keeping the same behaviour: the golog program (26) is a rewrite of the program (22) in which the number of wrong answer can vary. *prepareQuestion*(3) as define in (26) has exactly the same effect as *prepareQuestion* as defined by (22).
- Modifications of the behaviour such as:
  - Modifications of parameters: the program (28) is a modification of the program (27) in which the argument to *prepareQuestion* can vary depending on the user profile;
  - Addition or modification of conditions taking into account the user profile: the equation (11) is a modification of the equation (9) by adding a condition to take into account the user profile.

V. Conclusion

We developed the idea that a personalized application can be seen as a formal extension of a non-personalized application. We illustrated this idea by modeling two scenarios of personalized applications. These examples showed that once the application is defined in the logical formalism of situation calculus, personalization can easily be added by minor modifications of the formalization of the non-personalized application.

The next step in our work is to extract and characterize the transformation between non-personalized and personalized application in order to apply it automatically, or at least provide tools able to help in the transformation.

References

[1] N. Stash, A. Cristea, and P. D. Bra, "Adaptation languages as vehicles of explicit intelligence in Adaptive Hypermedia," *International Journal of Continuing Engineering Education and Life Long Learning*, vol. 17, no. 4, p. 319–336, 2007.

[2] S. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Prentice hall, 2010.

[3] C. Jacquiot, Y. Bourda, F. Popineau, A. Delteil, and C. Reynaud, "GLAM: A generic layered adaptation model for adaptive hypermedia systems," in *Adaptive Hypermedia and Adaptive Web-Based Systems*. Springer, 2006, p. 131–140.

[4] M. Baldoni, C. Baroglio, and V. Patti, "Web-based adaptive tutoring: an approach based on logic agents and reasoning about actions," *Artificial Intelligence Review*, vol. 22, no. 1, p. 3–39, 2004.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP*. SpringerVerlag, 1997.

[6] S. McIlraith and T. Son, "Adapting Golog for programming the semantic web," in *Fifth International Symposium on Logical Formalizations of Commonsense Reasoning*, 2001, p. 195–202.

[7] ——, "Adapting golog for composition of semantic web services," in *Principles of knowledge representation and reasoning-international conference*, 2002, p. 482–496.

[8] R. Scherl, "A GOLOG Specification of a HyperText System," in *Logical Foundations for Cognitive Agents*. Springer-Verlag, 1999, p. 309–324.

[9] J. Carthy and P. Hayes, "Some philosophical problems from the standpoint or artificial intelligence," *Machine Intelligence*, 1969.

[10] G. Dubus, F. Popineau, and Y. Bourda, "Situation calculus and personalized web systems," in *International Conference on Intelligent Systems Design and Applications*, 2011.

[11] G. D. Giacomo and H. Levesque, "Projection using regression and sensors," in *International joint conference on artificial intelligence*, vol. 16, 1999, p. 160–165.

[12] G. D. Giacomo, H. Levesque, and S. Sardina, "Incremental execution of guarded theories," *ACM Transactions on Computational Logic (TOCL)*, vol. 2, no. 4, p. 495–525, 2001.

[13] G. Giacomo, Y. Lespérance, H. Levesque, and S. Sardina, "IndiGolog: A high-level programming language for embedded reasoning agents," *Multi-Agent Programming:*, p. 31–72, 2009.

[14] H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl, "GOLOG: A logic programming language for dynamic domains," *The Journal of Logic Programming*, vol. 31, no. 1-3, p. 59–83, 1997.

[15] G. D. Giacomo, Y. Lespérance, and H. Levesque, "ConGolog, a concurrent programming language based on the situation calculus," *Artificial Intelligence*, vol. 121, no. 1-2, p. 109–169, 2000.