End-user Composition of Web-based Services: The "Plus Alpha" Approach

Rune Sætre
Computer and Information Science (IDI), NTNU
E-mail: satre@idi.ntnu.no
Mohammad Ullah Khan and Peter Herrmann
Telematics (ITEM), NTNU
Norwegian University of Science and Technology (NTNU)
NO-7491 Trondheim, Norway

Abstract—**The Web is truly becoming ubiquitous now that more and more people own a mobile device with a web browser. Many users want to exploit the fact that they can be online practically 24 hours a day, and they have ideas about what services they need. However, most of the users are not proficient in any programming language, so they are depending on a relatively small number of software engineers to help them create the desired services. This is slowing down the potential progress, and creating a need for better ways to involve users in their own service development. In this paper, we first present a method for end user-based service composition which is developed in the UbiCompForAll project. Thereafter, we investigate the effect of making a small incremental (plus alpha) improvement to an existing scenario. As a showcase, we use the "Doctor's appointment" scenario to illustrate how end users can be involved in the creation of web-based services for themselves. The *plus alpha* scenario clarifies that newly achieved service compositions can significantly profit from previous ones by reusing existing building blocks.**

1. Introduction

There are two ways to create new web-based services. One is by building something new from the bottom up, and the other is by building something on top of the existing web technologies.

"Plus alpha" is a phrase that has been used for many years in Japan[1]. The connotation of the term is related to incremental creativity. When you come up with a new idea, you are basically rehashing an old idea, "plus alpha". Some Japan scholars see the plus alpha concept as a way of understanding how Japanese artists and engineers work, refining the wheel instead of inventing it. We believe that big improvements can be made in current web services just by applying the *plus alpha* approach in a principled manner.

The world of cell-phones and mobile WiFi-devices is changing rapidly, and modern phones now have the same computational power as a desktop computer had only a few years ago. Many small programs and services are being made to harness this power in order to save the users from difficult or boring tasks. For example, the address book application on the phone can remember all the phone numbers for the user, so he or she can call someone just by typing (or saying) their name, or perhaps just by clicking on the correct portrait picture.

The background for this paper is the UbiCompForAll project which is following up on many of the ideas proposed in the SIMS project [2]. The main project goal of UbiCompForAll is to create a methodology for end-user composition of services [3]. One important aspect is the development of a quite simple notation [10] that is better suited for non-experts than, e.g., UML (see Figure 3) which is used in SIMS.

In general, people who are not familiar with Information and Communication Technology (ICT) will not be willing and able to create new services from scratch, but it seems possible that they can create personalized services by composition from already existing service building blocks. Here, it is of course relevant to lay out the interfaces of these building blocks in a way that their composition with others is quite simple. Another crucial factor is to enable the reuse of building blocks for various service compositions. This allows the end-users (composers) to get a better understanding of both the functionality of a building block and the best way to compose it with other ones. In this paper, we describe a typical case of this "plus alpha"-based reuse of building blocks by showing how the elements of an already composed service can be utilized to create another similar

[1]http://everything2.com/title/plus+alpha

service.

In the following, we present an overall description of the UbiCompForAll project and some related work in Section 2. Section 3 explains the UbiCompForAll methodology, including the end-user notation and how the syntax relates to the underlying meta-model and to the tools involved. Section 4 explains the *doctor's appointment* scenario that creates the baseline for the "plus alpha" scenario. Section 5 focuses on how the *plus alpha* approach can help. By extending the *doctor's appointment* scenario we highlight how several parts of a service composition can be re-used in another setting. Section 6 provides the results. Finally, Section 7 discusses the current state of affairs and concludes the paper by outlining the future work.

2. Background

2.1. UbiCompForAll

The scenarios in this paper are taken from a project called UbiCompForAll — Ubiquitous service Composition For All users[2]. *End-user* service composition is a research area where end users develop or modify software artifacts by combining existing services. The user can define the execution behavior of those services through

- a trigger to start the composed service,
- a sequence of steps to be performed,
- a set of information settings.

UbiCompForAll provides a methodology that can be applied in various application domains and that can be supported on several different run-time platforms.

The *composition tool* being developed will allow non-experts (without programming experience) to compose their own services. The hypothesis is that a visual formalism and tools can be developed to support the end users in composing (their own and other's) services. It will be shown that service composition can be supported by generic solutions in the form of methods and middleware that significantly reduce the complexity of developing these new composite services.

After a paper-prototyping phase and a comprehensive state-of-the-art survey, an intuitive notation and approach for end-user composition were selected [10]. In the following prototyping phase, a runtime demonstration system will use the open source Android cell phone architecture to prove the validity of our approach.

Since most of the users in the UbiCompForAll scenarios do not have the necessary computer skills to program the solutions without some help, we must provide a framework that lowers the threshold enough for these ordinary end users. In particular, we offer a set of service building blocks that each may realize quite complex functionality, but that offers comprehensible interfaces that facilitates their composition with other blocks into a personalized service.

[2]http://ubicompforall.org/

2.2. Related Work

This paper describes a natural step on the way towards an integration of the social, mobile and semantic webs, better known as the *ubiquitous web* [7].

There has already been much research on the modelling of business processes, and the *Web Services Business Process Execution Language, version 2.0* (BPEL, for short) can be used to execute composed services. Two extensions to BPEL are especially relevant to our work. The first extension is called BPEL4People[3] and it provides a standardized way to model human-performed activities in the composed business processes [1]. This is useful to us since the UbiCompForAll scenarios also needs modelling of the end users in the compositions.

Another extension to BPEL that is relevant to our work is the "BPEL for Semantic Web Services" (BPEL-4SWS) [9]. BPEL-4SWS enables activities to be described in a flexible manner, based on ontological descriptions of service requesters and providers. Earlier, the services and the user's goals had to be explicitly linked to specific activities with the Web Service Description Language (WSDL). This new loose semantic coupling between activities and goals is similar to the way that Android services are started, through the "intent and filter" mechanisms for starting new *activities, services or broadcast receivers*[4].

A recent approach that goes even further in simplifying the descriptions of available web services is the work by Xiao et al. [11]. The authors assigned single keywords from their ontology to one thousand services during a three week experiment. In cases where multiple similar services are available, ranking can be used to choose among them [4].

Like with most graphical domain specific languages, the semantics of many common domains/scenarios can be captured and understood even by users without any previous experience. Hauser [5] and Holmes et al. [6] use model-to-code transformations to demonstrate that graphical scenario-compositions can be mapped to executable BPEL4People code. In our case, the created BPEL web service could then be executed directly by an application on the Android prototype devices.

3. Methods

The proposed methodology/framework in UbiCompForAll describes a step-by-step procedure that can be used both by the end users, to create their own service

[3]http://xml.coverpages.org/bpel4people.html

[4]http://developer.android.com/guide/topics/intents/

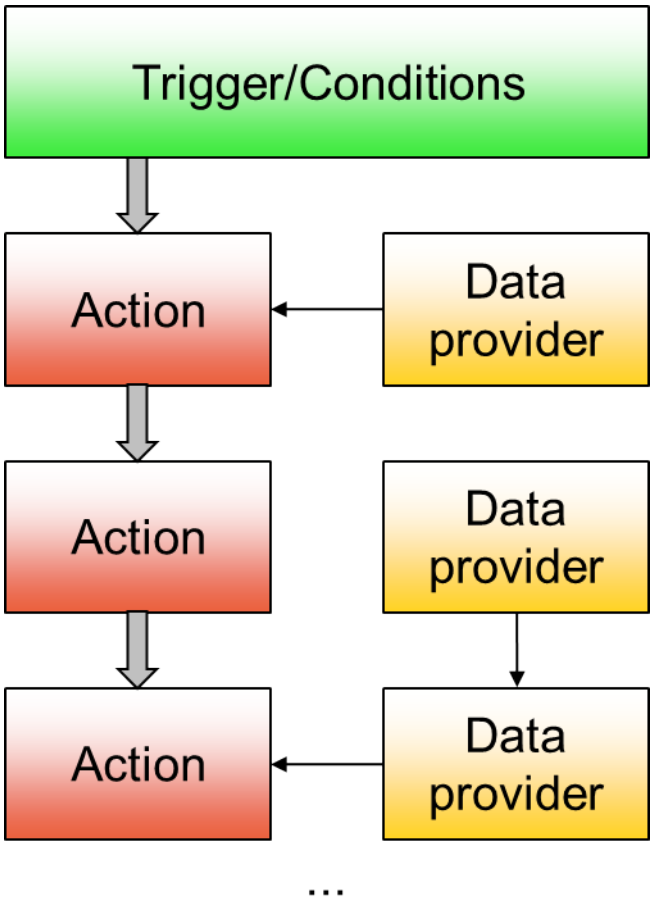

Fig. 1. The basic syntax of the Simple Language for composition. The arrows will be hidden from the end users.

compositions, and by the service developers, to develop the actual service-parts (building blocks). A tool is also provided to aid the users at the various steps.

The syntax of the end-user notation is described in Section 3.1, with two simple example compositions shown in Figures 5 and 6. It is necessary to have a user-friendly and intuitive notation when composing the compositions. In Section 3.2, the syntax is described according to the underlying meta-model. The tools that are needed to create our own *composition tool* are described in Section 3.3, and our simple ontology is introduced in Section 3.4.

3.1. Syntax and the End-user Notation

The term "end user" means both the *composers* and the other users that have access to a composed service made by a composer. They are not expected to have much programming experience, but they can use a PC or a hand-held mobile device. This set of end users require an intuitive notation for service composition and for editing of such example compositions.

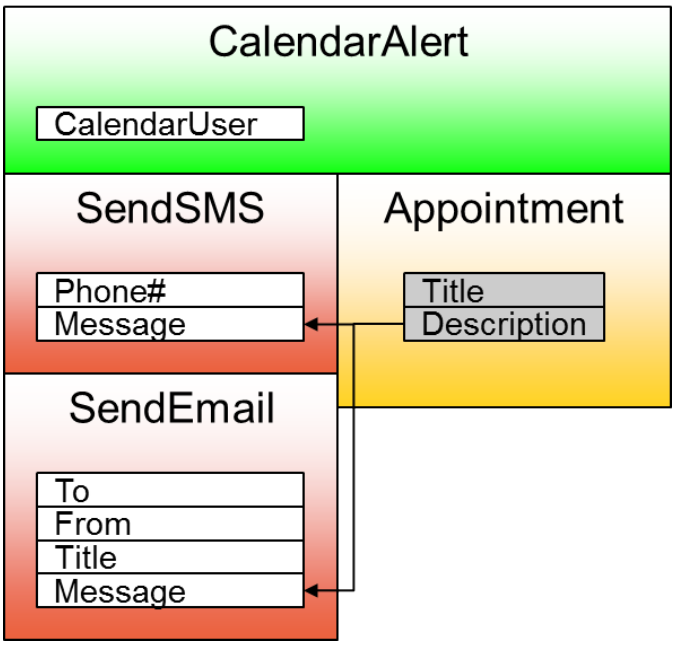

Fig. 2. A more concrete example showing the syntax of the SimpleLanguage. When *CalendarAlert*, Then *SendSMS(Appointment.Description)* and *SendEmail(Appointment.Description)*.

Several different styles of notations have been tested: UML notation, flow-based notation[5], notations used in different related tools and approaches like Microsoft Visual Programming Language, Lego Mindstorms, Scratch, Matlab Simulink, Quartz Composer, Yahoo Pipes etc. Based on this research, we want to make a notation that is intuitive for end users. The main idea is that certain tasks should be performed whenever certain conditions are met. This results in the simple notation syntax shown in Figures 1 and 2.

Figures 2, 5 and 6 present fragments of a composition for the doctors's appointment scenario (Sections 4 and 5). A scheduler event is generated one hour before the appointment and the user(s) are notified through SMS and email. In order to send the SMS, the phone number of the user and the (dynamic) message text to send are needed. These are represented by the properties 'Phone#' and 'Message' respectively. The actual value of the 'message' text can be provided by a text wrapper which uses the departure time from the bus-query as a part of the text message sent to the user.

3.2. Meta-model Description

A meta-model has been developed to describe the concepts that are required to represent the end-user compositions [10]. These concepts and the relationships between them are provided in Figure 3. The composition tool (Section 3.3) uses the end-user notation and this

[5]http://en.wikipedia.org/wiki/Flow-based_programming

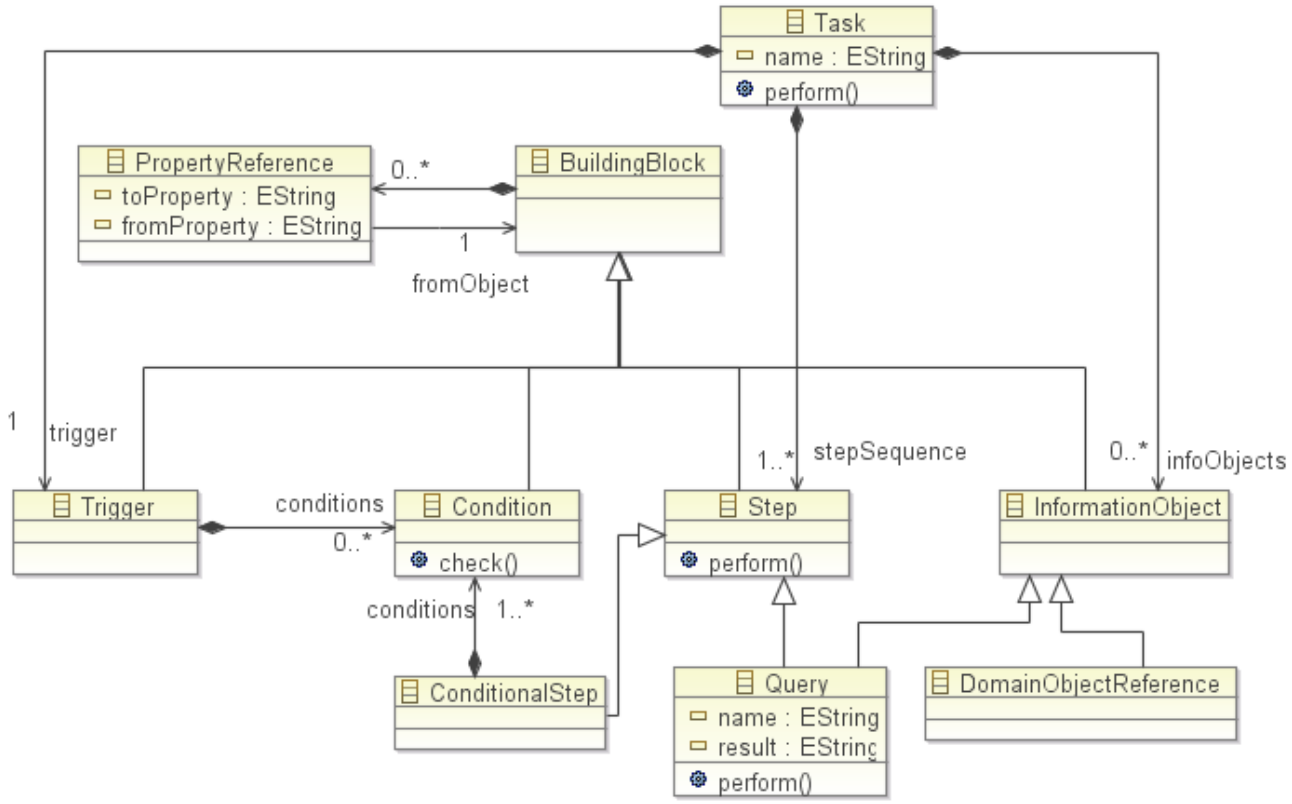

Fig. 3. Simple Language meta-model. The related boxes in Fig 1 are colored as follows: green for *trigger/conditions*, red for *steps*, and yellow for *InformationObjects/Queries/DataProviders*.

meta-model to let the end users create their own service composition.

The composition unit is called a Task and consists of Steps to be performed and a Trigger which must be satisfied to initiate the execution of the task. A Trigger may be a collection of predefined conditions. In addition, the Task may consist of a number of Queries. A Step represents what is done by an single *service unit*, and a sequence of Steps makes up the whole Task. In addition, some of the Steps may require extra information from the outside. It is the job of the InformationObjects to hold such external information, which will be provided by *Query services*.

In addition to the Trigger for the whole Task, an individual Step can also be dependent on one or more predefined conditions. Such a Step is called a ConditionalStep, and the common term for Triggers, Conditions, Steps and InformationObjects are *Building Blocks*.

The PropertyReferences are a way of connecting the individual building blocks. The evaluation of conditions and the execution of the Steps often require the values of certain properties. These values can be set at the composition time or at the run time. The concept InformationObject is introduced in order to hold such values of particular properties. Such properties can be set by Steps and retrieved by property Queries. When the value of a particular property reference depends on other properties, the PropertyReferences can be used to make those references.

3.3. Tools and Technology

A *composition tool* was developed to make it possible for end users without much programming experience to compose their own services. The tools and technologies that are used to create the *composition tool* are the Google Web Toolkit, the Window-Builder and Java for Android. Here, we explain how they are used to implement both the composition and the run-time systems.

The Google Web Toolkit (GWT)[6] is a development toolkit for building and optimizing complex browser-based applications. It provides a Java to JavaScript cross compiler, which enables writing the application in a statically compiled and strongly typed language like Java.

The composition tool has to work both on PCs and on mobile phones. We target end users with limited or no ICT knowledge, so the *composition tool* needs to be easy to use and easy to install. We therefore decided

[6]http://code.google.com/webtoolkit/

to make a *composition tool* that runs in a web browser. The web-based tool enables easy communication between the device and a server (if needed, when used in on-line mode), but it can also be used to create compositions while off-line. GWT is well suited to build such a *composition tool*. Recently, Google has also made WindowBuilder[7] compatible with GWT, and this makes the tool development task easier. Since WindowBuilder helps designing the GUI, much of the programming can be done visually.

3.4. Ontology

The concepts in our ontology (see Figure 4) consists of classes representing different entities relevant to scenarios involving travelling, like in the *Doctor's appointment* scenario described in Section 4. To simplify the end user's service-composition task, we are developing an ontology for the relevant domains.

The ontology can be integrated with the service *composition tool* (described in Section 3.3) so that the service composer (an end-user) can benefit from semantic guiding during the service composition. The run-time platform can also use the ontology to make automatic decisions, for example when automatically selecting between multiple alternative services.

4. The Doctor's appointment Scenario

Summary: Ove (47) creates a service to help his aging mother Oda (75) get to her doctor's appointments. It keeps track of the appointments, reminding her when its time to go, it helps her find her way to the doctor's office, and it lets him keep track of her progress and alerts him of problems.

4.1. Scenario Construction

Scenario:

1) Ove creates a composition for his mother entering her next appointment to the doctor. The service has access to the bus schedule and calculates the time she has to leave the house based on the appointment.
2) The service sends an SMS and sounds an alert on Oda's phone three times: one day before, one hour before and exactly when she should leave her house.
3) When Oda is on her way, the service keeps track of her location using her cell phone, and has a route with schedule specifying where she should be at what time relative to the bus and appointment times. The cell phone shows a short message about the bus to take and its time of departure. Alerts are sent to Ove if Oda deviates from the expected schedule. All events are logged so that Ove can access and monitor them from his computer and phone.
4) Oda does well: She catches the bus, gets off at the desired stop, and start walking towards the doctor's center. However, she meets an old friend on the way and forgets about the appointment while talking. The system detects her delay through insufficient progress in her position and alerts her about the appointment again. She ultimately makes it to the doctor in time.
5) Oda receives similar support on her way back home too.

[7]http://code.google.com/javadevtools/wbpro/

4.2. Analysis

With Doctor's appointment and the other UbiCompForAll scenarios in mind, some important distinctions between design-time (composition) and run-time (execution) should be made:

At design time (See Figure 5):

- The composer will create application compositions by using the service building blocks like appointment handling and the sending of SMS resp. emails (see Fig. 2).
- The composition is created in a activity-like fashion. This means that tasks are created (see Section 3.2) which execute a particular block or perform a particular sequence of steps only when a certain condition holds. For instance, the location tracking of Oda will only be started after she has left the house. Such conditions are dynamic in nature so that the evaluated result may depend on some information obtained at run time. Thus, the composer has to be aware already on design time that deviations of the planned course of events may occur.
- Ontologies may be used to discover certain building blocks that can help solving specific problems. An ontology can also aid in gluing together building blocks that should be executed in a specific sequence, based on the expected input and output from each building block. Further, they may help to lay out solutions to catch unintended behavior due to not foreseen deviations in the course of events.

At run time:

- The composed scenarios are selected based on the conditions evaluated at run time.
- The run time platform should take care of the fact that the user's needs may change, e.g., the user may re-compose the application at run time. This can be complicated since in many scenarios synchronization and real time issues may occur.
- Ontologies can be used in identifying and selecting from alternative scenarios and services that serves

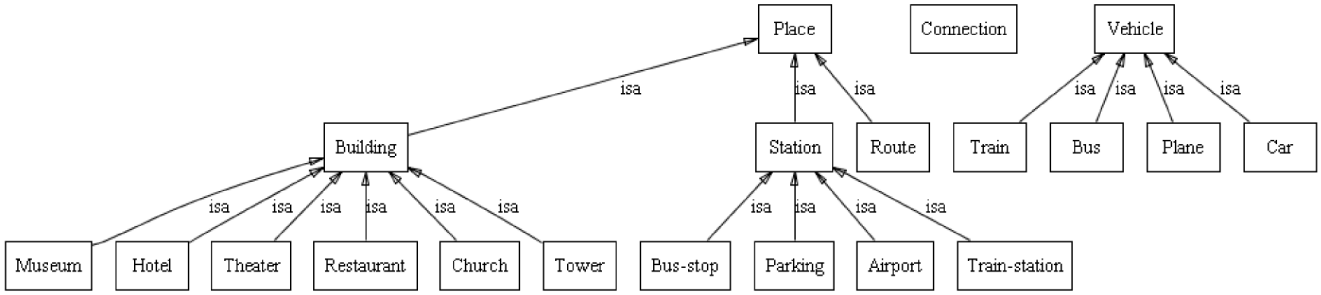

Fig. 4. Simple places and transportation ontology

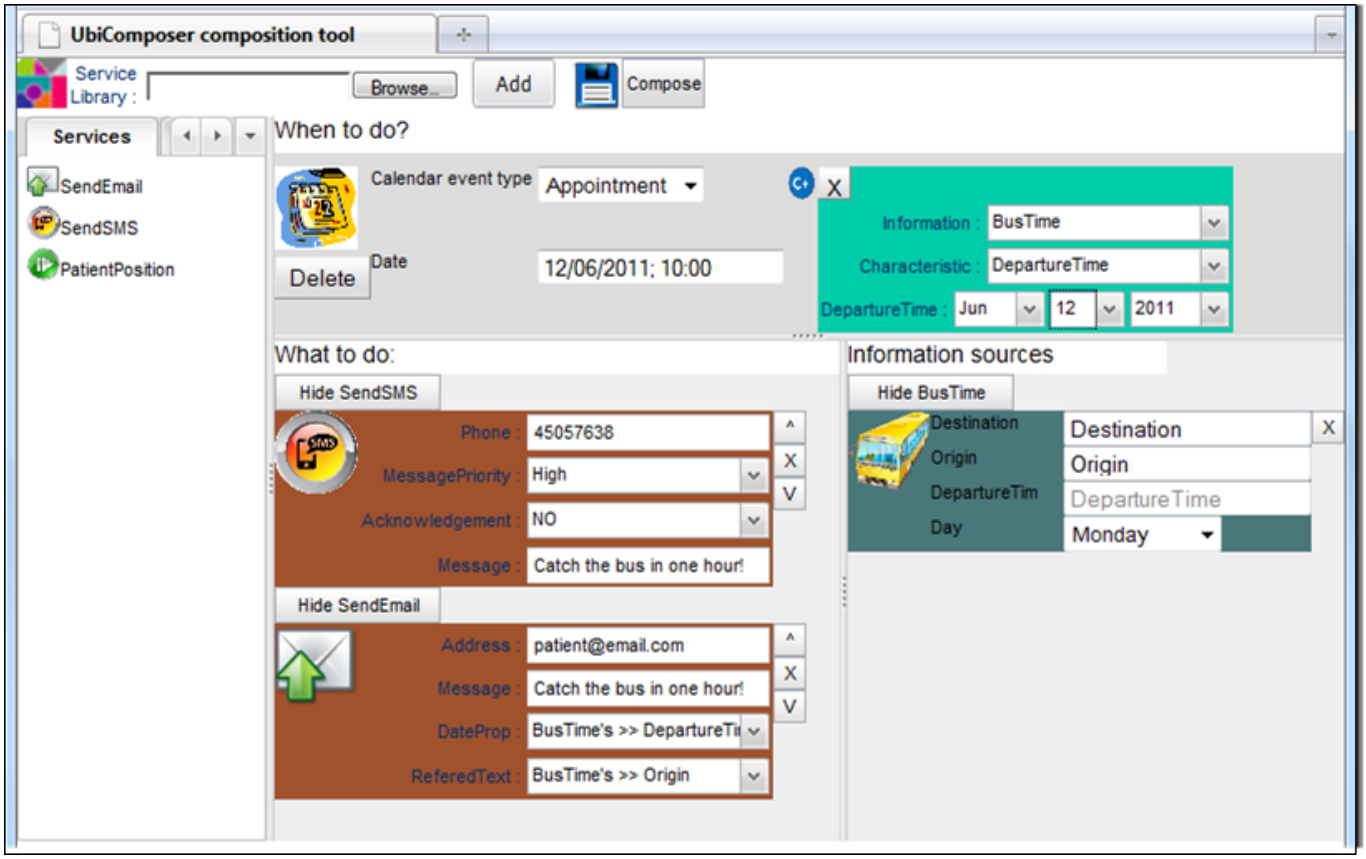

Fig. 5. Example composition for the doctor's appointment scenario

the same composition and the same building block, respectively.

5. The Doctor's appointment *plus alpha* scenario

With the "Doctor's appointment application" in place, Ove is now re-assured that Oda is doing good on her own. He knows that he will be notified when something is not according to the plan, so he can focus more on his other tasks. One day he hears about a doctor that is highly recommended because of his experience with cases like Oda's. Ove wants more support than he gets from Oda's current doctor, whose office is quite far away, so he would like Oda to change her doctor.

This is an example of a goal-driven query, where someone is looking for a new doctor. In Norway, every citizen is assigned one specific doctor that should always be the first contact-point no matter what kind of illness the patient is facing. Every citizen is allowed to change their primary doctor twice per year, but the most popular doctors are fully booked and not available for new patients most of the time. The government provides a

list[8] with all the doctors and the number of available free patient-slots for each doctor. The list is updated continuously, but new patients are only welcomed at random times. Usually, these new openings appear a few minutes after seven in the morning, but only after around twenty old patients have moved away from a doctor. However, there are also many small updates to the list during the day, and sometimes only one new slot is made available.

Unfortunately, there is no public initiative to provide a waiting list system for the popular doctors, so it is up to the frustrated patients to watch the list carefully every day. And when the desired doctor suddenly becomes available, they have to make their move very quickly, before someone else fills up the slots again.

Ove has been watching the official vacancy list for doctors for several weeks now, but there has been no openings except a single one that he missed, because someone took it before he could talk to Oda and make the change. He is getting very tired of remembering to check the internet page every day, and he does not want to miss another opportunity, so he would like to get an automatic reminder as soon as the doctor becomes available again.

Ove does not have the necessary programming skills to make such a waiting-list service on his own, so he starts looking for other similar services. Failing to find any, he decides to give the *UbiComposer* composition tool another try.

5.1. Scenario Construction

Ove likes the procedure of the Doctor's appointment scenario, which notifies him when Oda is out of schedule. The notification is done by sending an SMS to Ove's mobile phone. In addition, he receives an email if his phone is in silent mode, e.g., when he is in a meeting. As this procedure already proved to be convenient, he would like to use it also in the new scenario, to be notified when Oda's proposed new doctor has vacancies.

Following the "plus alpha" approach, Ove likes to reuse as much as possible from the original scenario. With help of the composition tool and its ontology, he starts to collect building blocks to build the new scenario:

- Ove needs a way to extract the necessary information from the vacancy home page. He recalls, that he used an extractor to filter out bus times in the original scenario.
- Further, he has to find a way to run the extractor automatically every few minutes during the day but here he can take the triggering mechanism which notified Oda before she had to leave and simply adapt it such that it runs every minute.
- Finally, he needs a notification mechanism to be informed as soon as the desired doctor is able to accept Oda on his list. As mentioned above, he has the notification mechanism in place and it is not too difficult to take the building blocks SendSMS and SendEmail and to modify the corresponding task descriptions.

Having all ingredients in place, Ove does only need to link the extractor to the web page containing the waiting list and to adjust it in a way that the number of free patients for the particular doctor is sent out. Then he has to adapt the trigger to start the extractor every minute and to execute the notification mechanism if the number of vacancies is larger than 0. Finally, he has to link the triggering mechanism with the reused notification mechanism such that an SMS and, if applicable, an email is sent when there are new vacancies.

5.2. Analysis

It is evident that the reuse factor in this scenario is very high. Basically, all ingredients can be directly taken from the Doctor's appointment scenario and just have to be composed in another way. Thus, Ove can use the experience he gained while creating the original scenario seamlessly and the whole scenario can be performed within 30 minutes.

From our experience with model-based system development, we came to the conclusion that the reuse of building blocks in different projects of a certain application domain can be significant (see, e.g., [8]) and we assume a quite similar effect also for end user composition. Of course, non-experts have a learning curve when starting to compose services from building blocks since the used notation and composition technique are new to them. Nevertheless, when they have got the hang of it, the reuse potential will make the creation of new examples quite easy.

6. Results

The concepts presented in this paper are still being implemented and developed further, but during this work several building blocks have already been provided. So, even though only some preliminary results are ready, the end users can already make their own "plus alpha" service compositions. The existing building blocks provide functionality for extracting information from online resources, sending and receiving messages, and so on. More building blocks will be published on the UbiCompForAll project home page towards the end of the project.

The building blocks are based on functionality provided by underlying existing services. In some cases, appropriate services cannot be found, so the domain adapters have to provide the missing services that they

[8]https://tjenester.nav.no/minfastlege/innbygger

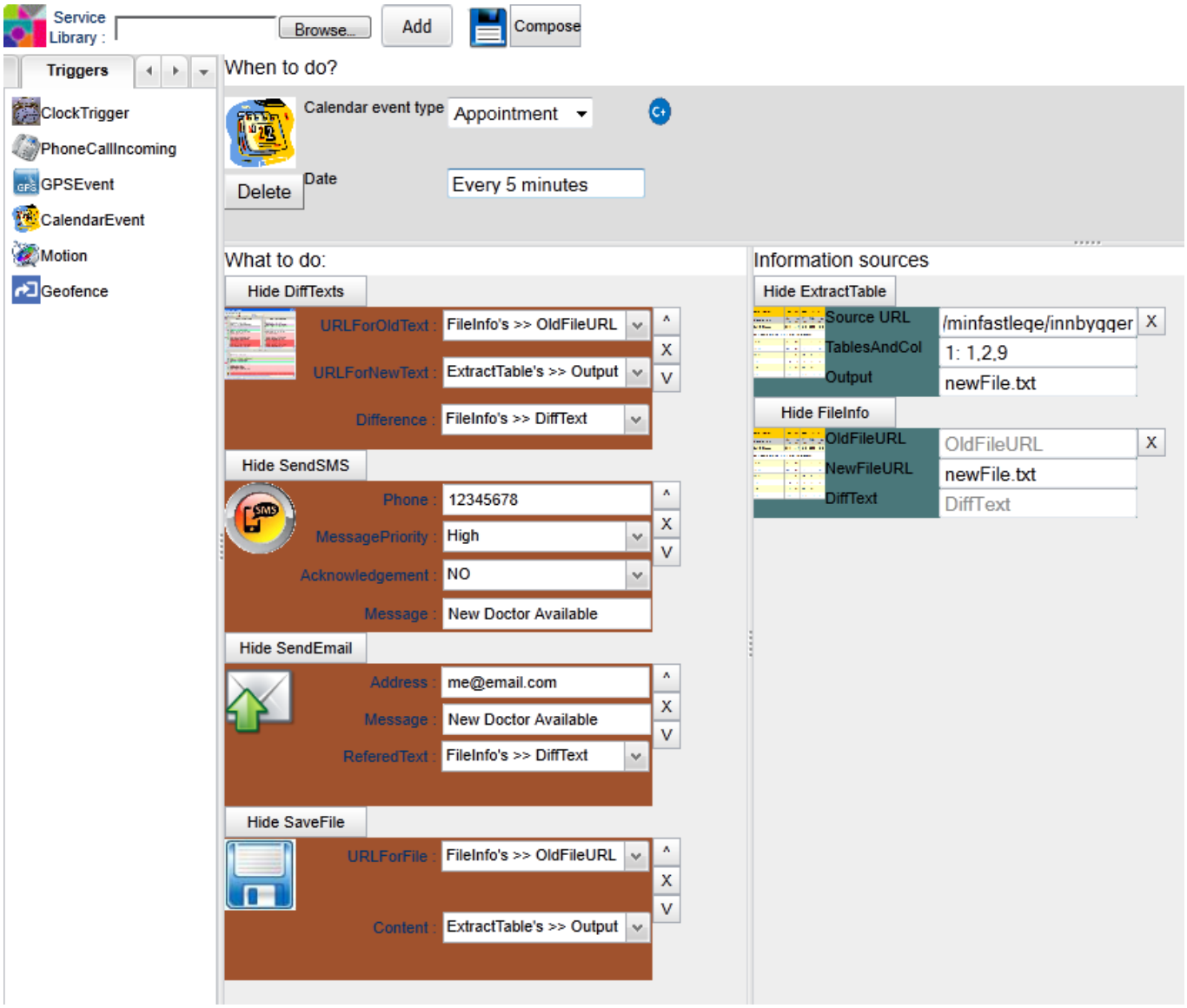

Fig. 6. Example composition of an extractor of online tables, with a diff and message step

want to include as building blocks. For example, the only freely available table extractor service is Google Refine[9], but since it is not made for online use, we had to implement our own online extractor for online tables. This new service is called ExTable, and is published on the Web[10].

The composed "plus alpha" service is potentially usefull for all the citizens of Norway, so it was made available as a web service as well[11]. In just one month after being published, the service has already attracted several users from all over the country.

[9]http://code.google.com/p/google-refine/
[10]http://www.idi.ntnu.no/~satre/extable/
[11]http://www.idi.ntnu.no/~satre/ubicomp/fastlegevakten/

7. Discussion and Future Work

In order to understand how the plus alpha approach can be generalized to work in as many settings as possible, two different scenarios were analyzed. With the goal of UbiCompForAll (to create a methodology for end-user composition of services) in mind, we try to find sub-parts of the composition scenarios which are similar enough to be re-used in new settings. Some building blocks are very general, and can therefor be used in practically any composition. For example, extracting data from a generic table found online is necessary for many of the scenarios that involve some kind of information extraction.

Our implementation also follows an incremental (plus alpha) approach where the concepts and the implementation are updated based on feedback from end-user

testing, performed on several different platforms with several different scenarios. The ontology that is being developed for *places and transportation* (Figure 4) can be used in all the scenarios related to traveling. It can assist the users in selecting appropriate services and in setting the necessary composition parameters in the composition tool. The ontology sits in the intersection between what the users expects and what the services can do, so it allows users to search for services using some predefined tags. If no services are found, the results can be expanded to include similar services that are tagged with terms that are close in the ontological hierarchy. The users can be asked to judge the appropriateness of these possibly related services.

Even for a very simple ontology like the one in Figure 4, there are some inherent challenges in creating *the right* ontology. For example, one could ask why there is no line from "Train-station" to "Building". The answer is that this really depends on the intended use of the ontology. There is already much research on automatic creation of ontologies from text. In our future research, we will target automatic creation of ontologies, for example for a new city or area.

Acknowledgments

This work is a part of the UbiCompForAll project, which is funded by the Research Council of Norway. We are thankful for their support. Special thanks to the project members Erlend Stav, Jacqueline Floch, Lars Thomas Boye, Jon Atle Gulla and Alfredo Pérez Fernandez for their continued support, feedback through frequent discussions, and contributions to the figures in this paper. Finally, many thanks to the three anonymous reviewers that helped improve the quality of the camera-ready version of this paper. Their constructive feedback was greatly appreciated.

References

[1] A. Agrawal, M. Amend, M. Das, M. Ford, C. Keller, M. Kloppmann, D. König, F. Leymann, R. Müller, G. Pfau, K. Plösser, R. Rangaswamy, A. Rickayzen, M. Rowley, P. Schmidt, I. Trickovic, A. Yiu, and M. Zeller. WS-BPEL extension for people (BPEL4People), version 1.0. *Joint white paper by Active Endpoints, Adobe Systems, BEA Systems, IBM, Oracle, and SAP*, 2007.

[2] J. Floch, C. Carrez, P. Cieslak, M. Rój, R. Sanders, and M. Shiaa. A comprehensive engineering framework for guaranteeing component compatibility. *Journal of Systems and Software*, 83(10):1759 – 1779, 2010.

[3] J. Floch, P. Herrmann, M. U. Khan, R. Sanders, E. Stav, and R. Sætre. End-user service composition in mobile pervasive environments. In *EUD4Services 2011 workshop*, Torre Canne, Italy, June 07 2011.

[4] D. Grigori, J. C. Corrales, M. Bouzeghoub, and A. Gater. Ranking BPEL Processes for Service Discovery. *IEEE Transactions on Services Computing*, 3(3):178–192, July 2010.

[5] R. Hauser. Automatic transformation from graphical process models to executable code. Technical report, Eidgenössische Technische Hochschule Zürich, May 2010.

[6] T. Holmes, H. Tran, U. Zdun, and S. Dustdar. Modeling Human Aspects of Business Processes – A View-Based, Model-Driven Approach. In I. Schieferdecker and A. Hartman, editors, *Model Driven Architecture – Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, chapter 17, pages 246–261. Springer, Berlin / Heidelberg, 2008.

[7] A. Hotho and G. Stumme. Towards the ubiquitous Web. *Semantic Web*, 1(1):117–119, 2010.

[8] F. A. Kraemer and P. Herrmann. Automated Encapsulation of UML Activities for Incremental Development and Verification. In *Proc. of the 12th Int. Conf. on Model Driven Engineering, Languages and Systems (Models)*, volume 5795 of *LNCS*, pages 571–585, 2009.

[9] J. Nitzsche and B. Norton. Ontology-Based Data Mediation in BPEL (For Semantic Web Services). In W. Aalst, J. Mylopoulos, N. M. Sadeh, M. J. Shaw, C. Szyperski, D. Ardagna, M. Mecella, and J. Yang, editors, *Business Process Management Workshops*, volume 17 of *Lecture Notes in Business Information Processing*, chapter 53, pages 523–534. Springer Berlin / Heidelberg, 2009.

[10] R. Sætre, M. U. Khan, E. Stav, A. P. Fernandez, P. Herrmann, and J. A. Gulla. Towards Ontology-driven End-user Composition of Personalized Mobile Services. In *Proceedings of the 2011 NLDB conference*, pages 242–245, 28-30 June 2011.

[11] H. Xiao, Y. Zou, R. Tang, J. Ng, and L. Nigul. Ontology-driven service composition for end-users. *Service Oriented Computing and Applications*, 5(3):159–181, Sept. 2011.