Contents lists available at SciVerse ScienceDirect

Pervasive and Mobile Computing

journal homepage: www.elsevier.com/locate/pmc

A semi-automatic end-user programming approach for smart space application development

Marko Palviainen [*], Jarkko Kuusijärvi, Eila Ovaska

VTT Technical Research Centre of Finland, P.O. Box 1000, FIN 02044 Espoo, Finland

ARTICLE INFO

Article history:
Received 19 April 2012
Received in revised form 16 April 2013
Accepted 29 April 2013
Available online xxxx

Keywords:
Command-oriented end-user programming
Goal-oriented end-user programming
Smart Modeler
Smart space application
Ontology

ABSTRACT

This article describes a semi-automatic end-user programming approach that: (i) assists in the creation of easy-to-apply *Semantic End-User Application Programming Interfaces*(S-APIs) for the APIs of legacy software components; and (ii) enables the usage of S-APIs in *command-oriented* and *goal-oriented* end-user application programming. Furthermore, a reference implementation is presented for the approach that provides visual programming tools and an agent-based execution environment for smart space applications. The use of the approach is exemplified and tested in a case study in which S-APIs are created for a home automation system and for a personal assistant application, and then utilized in end-user programming performed in desktop and mobile environments.

1. Introduction

The Internet of Things [1] is a trend that develops enabling technologies for interoperable devices and applications that are capable of interacting across vendor and industry domain boundaries. Imagine a situation where there are two applications developed for end-users: a personal assistant application for smartphones and a home automation application for controlling the home environment. However, in real life there is a need for an application that merges the functionalities of these two applications. For example, it would be nice to have an application that enables an end-user to use the home automation system via the smartphone or to utilize the context-sensing capability of the smartphone in the home automation system. Unfortunately, in most cases the integration of applications is not possible due to missing open interfaces, or at least the integration of functionalities from separate applications requires significant integration effort and software professionals to perform the integration tasks. In practice, these issues prevent the existence of many useful applications that could make everyday life easier.

It should be possible for a person entering a smart environment to compose an application for his/her needs. Unfortunately, the existing software engineering approaches do not provide adequate support for end-user programming. Application Programming Interfaces (APIs) declare the operations and inputs and outputs for the operations that are made available for other software components/systems. Unfortunately, although APIs often provide documentation and usage instructions for the operations, typically only software professionals that have a lot of knowledge of software engineering can use these APIs. In this article we use the term *professional APIs* to cover these APIs. We argue that there is a great need for easy-to-apply APIs intended for end-users that want to create applications for different purposes. In this article we use two kinds of APIs for this purpose: *Semantic End-User Application Programming Interfaces* (S-APIs) and *Execution APIs*. The next paragraphs describe these APIs in more detail.

[*] Corresponding author. Tel.: +358 405051016.
E-mail addresses: marko.palviainen@vtt.fi (M. Palviainen), jarkko.kuusijarvi@vtt.fi (J. Kuusijärvi), eila.ovaska@vtt.fi (E. Ovaska).

1574-1192/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.pmcj.2013.04.002

S-APIs are intended for development purposes. They facilitate creation of execution sequences for applications and use of professional APIs by providing easy access to the information/operational capabilities that are available in the professional APIs. Research on natural programming shows that end-users solve programmatic tasks using familiar logical constructs like "if" and "when", combined with consequence words like "then" or "and" to specify and sequence procedures [2]. Mathematical operations and looping actions are avoided [3]. This kind of structure is utilized in our approach. Thus, unlike professional APIs, the S-API does not just define operations but structures, i.e., *commands* that also describe the execution branches to which the end-users can bind other commands. Thus, the end-user does not need to use if–else structures in programming; instead it is the concern of the commands to control the execution flow. Thus, the "if" is implemented inside a command, whereas the execution branches define the "then" branches. As a result, it is easy for the end-user to create execution sequences from available commands. Of course, the expressiveness of this kind of language is more limited than programming languages such as C++ and Java.

Execution APIs are intended for execution purposes. They support execution of the end-user's applications. End-users develop applications for physical environments that contain shared computing resources (e.g., mobile computers, embedded devices, and wireless networks) for themselves and their applications. These kinds of dynamic environments require that the Execution APIs also take care of resource allocation and thus hide the complexity of execution from the end-users.

In this article a smart space is understood to be a Semantic Information Broker (SIB) that provides a named search extent for information. We have previously developed an Interoperability Platform (IoP)-based architecture for cross-smart space applications [4,5]. This article extends our previous work and shows how the functionalities of professional APIs can be used in end-user programming of cross-smart space applications. This article focuses on S-API creation and end-user programming activities, and it also shows how these activities are related to each other, along with guidelines for S-API development. This article makes the following contributions:

- Firstly, the article presents a novel semi-automatic end-user programming approach that: (i) assists software professionals in creating easy-to-apply S-APIs and Execution APIs for end-user programming; and (ii) enables end-users to use S-APIs in the *command-oriented* or in the *goal-oriented* application programming.
- Secondly, the article presents a reference toolset for the approach. The end-users are mostly non-programmers and thus such programming of smart space applications requires: (I) introduction of programming elements in a way that is understandable for end-users; (II) support for end-user programming, in which a person entering a smart environment composes an application matching his/her needs; and (III) flexibility with respect to adding new devices and software components [6]. The reference toolset is designed for the above described requirements. The reference toolset includes RDFScript language, visual programming tools and an agent-based execution environment for supporting requirements I and II, whereas the S-APIs enable easy access to the capabilities of the smart objects (requirement III).

The article is structured as follows: After the introduction, Section 2 presents background related to the end-user programming approach. The semi-automatic end-user programming method is presented in Section 3. To illustrate the approach Section 4 shows an example that shows how the approach can be utilized in desktop and mobile environments. Section 5 presents an evaluation for the approach. Our findings and experiences are discussed in Section 6. A discussion of the work related to our approach follows in Section 7. Finally, conclusions are drawn in Section 8.

2. Background

This section discusses background related to our end-user programming approach. The background consists of two main parts: An architecture (discussed in Sections 2.1 and 2.2) and the Smart Modeller tool (discussed in Section 2.3) which support use of our end-user programming approach together.

2.1. Interoperability platform

The Interoperability Platform (IoP) for smart spaces is based on three stone bases [7]: (i) The Internet that provides a generic communication platform; (ii) The interoperability model that makes computational units able to interoperate each others; and (iii) Context-aware computing that provides domain-specific assistance, optimized to the situation in hand. Some of the key principles of IoP are [7]: The IoP deals with information and manages a shared information search domain called Smart Space (SS), accessible and understood by all the authorized applications. Information is about the objects existing in the environment or about the environment itself. The information is represented in a uniform and use-case independent way. Information interoperability and semantics are based on common ontologies that model information. The rest of principles have been defined in detail in [7].

The IoP [8–10] follows the *blackboard* architecture and provides a *publish–subscribe* paradigm for smart space applications. The goal of the IoP is to make the information in the physical world universally available to various services and applications, regardless of their location, which aligns well with the Web of Things vision too. In the IoP architecture, a smart space application consists of two kinds of agents (in Fig. 1): Semantic Information Brokers (SIBs) and Knowledge Processors (KPs) that communicate by using the Smart Space Access Protocol (SSAP). An SIB is a lightweight Resource Description Framework (RDF) [11] database that provides *add*, *remove*, *query* and *subscribe* functions for the KPs and for the semantic

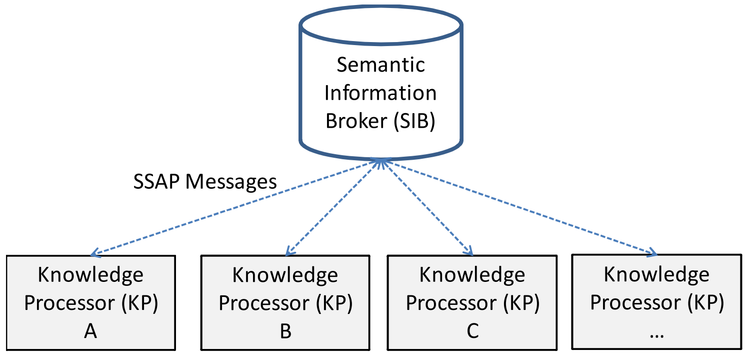

Semantic Information Broker (SIB)
SSAP Messages
Knowledge Processor (KP) A
Knowledge Processor (KP) B
Knowledge Processor (KP) C
Knowledge Processor (KP) ...

Fig. 1. An example of a smart space application.

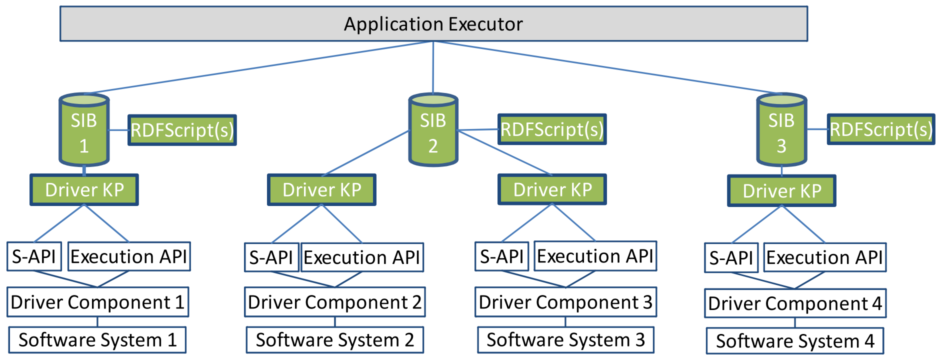


Fig. 2. An IoP-based architecture [4,5] for cross-smart space applications.

information stored to the SIB. KPs can join with SIBs, produce and/or consume semantic information in an SIB, and leave SIBs.

2.2. IoP-based architecture for cross-smart space applications

In many cases the smart space application cannot just be based on an SIB and communicating KPs but requires multiple SIBs. For example, there may be a need for an application that uses the personal SIB of a smartphone and the shared SIB of a smart building. Our IoP-based architecture supports creation of cross-smart space applications [4,5] (in Fig. 2). The following paragraphs briefly describe the main components of the architecture:

RDFScript language—It is a far too complex task for many end-users to compile executable applications from source code (e.g., Java or C++). Thus, we decided to develop an interpretable script language, called RDFScript for end-user programming of cross-smart space applications. End-user programming produces an RDFScript that describes the execution sequence for the cross-smart space application. The execution sequence consists of: (i) commands, (ii) the inputs, outputs, and execution branches of the commands, (iii) connectors between the execution branches and commands, and (iv) connectors between the outputs and inputs of commands.

Driver component—Execution of commands of S-APIs is based on the driver components. Smart space applications are often developed for physical environments that provide shared computing resources (e.g., mobile computers, embedded devices, and wireless networks) for end-users and their applications. These kinds of dynamic environments require that the driver components also take care of resource allocation and thus hide the complexity of execution from end-users.

Application Executor—This is the central entity that controls the execution cycle of a cross-smart space application and uses the Execution APIs of Driver KPs via SIBs for coordinating the processing activities in the Driver KPs. It (i) interprets the RDFScript, (ii) joins the selected smart spaces, and (iii) executes the application.

Driver KP—The ready-made Driver KP component will connect the driver component to the desired smart space and make it available for the Application Executors. The driver component/Driver KP can be either a part of the legacy software system or a separate processing element. In many cases the source code of the legacy software system is not available (e.g., for 3rd party developers), and thus it can be hard to deploy the driver component/Driver KP directly to an embedded software system. In this case there is a need for an external driver component that uses the professional APIs remotely.

In the architecture SIBs are utilized in the development, sharing and execution of the end-users' applications. This article extends the previous work and shows how it is possible to support reuse of the functionalities of professional APIs in

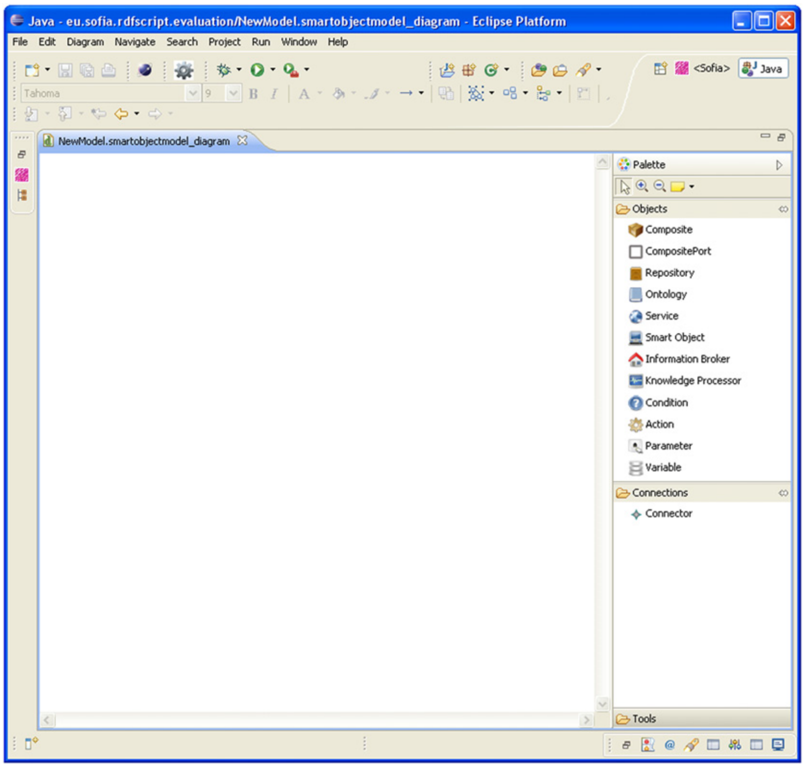


Fig. 3. The main window of the Smart Modeler.

command-oriented and in goal-oriented end-user programming of cross-smart space applications that use the capabilities of separate software components/systems.

2.3. Smart Modeler

We have previously developed a tool, called Smart Modeler [12] for the visual modeling of smart space applications (in Fig. 3). The main goal of the Smart Modeler is to provide an integrated development environment for smart space applications. The tool consists of a common framework and a set of extensions, so that new extensions can be easily introduced when needed. Various extensions to the Smart Modeler enable further automation of the process, contributing to the ease and speed of development. The extensions are Eclipse plug-ins and perform processing related to the smart space application models. Furthermore, to support interoperability, standard-based solutions are preferred in the tool. For example, the Smart Modeler is capable of importing/exporting smart object models as RDF graphs. Thus, tools that are able to read the RDF format are capable of utilizing the information that is provided in the smart object models. We decided to use the Smart Modeler as a starting point in the implementation of the reference tools for our end-user programming approach.

3. End-user programming approach for the development of cross-smart space applications

Our end-user programming approach provides the Semantic End-User Application Programming Interfaces (S-APIs) and an integrated development process for end-user programming of cross-smart space applications. The S-APIs and process are described in more detail in the following subsections. Furthermore, Section 4 provides an example how the approach can be utilized in desktop and mobile environments.

3.1. Semantic end-user application programming interfaces

At least two kinds of functions exist in the professional APIs (see Fig. 4): (i) *Methods* that provide specific kinds of functionalities for their users; and (ii) *event monitors* that deliver the observed events for the observers. Two kinds of commands are provided in S-APIs: commands that perform the desired functionalities and commands that activate the event monitors and thus enable the creation of applications capable of reacting to the events. Inputs, outputs, and execution branches are defined for each command. An input defines a name, type, and possibly a default value for data consumed by a

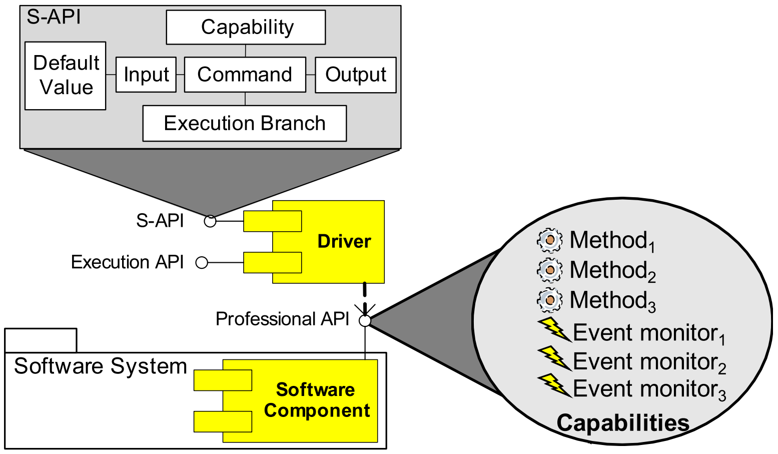


Fig. 4. The S-API provides access to the capabilities of the APIs used by software engineers.

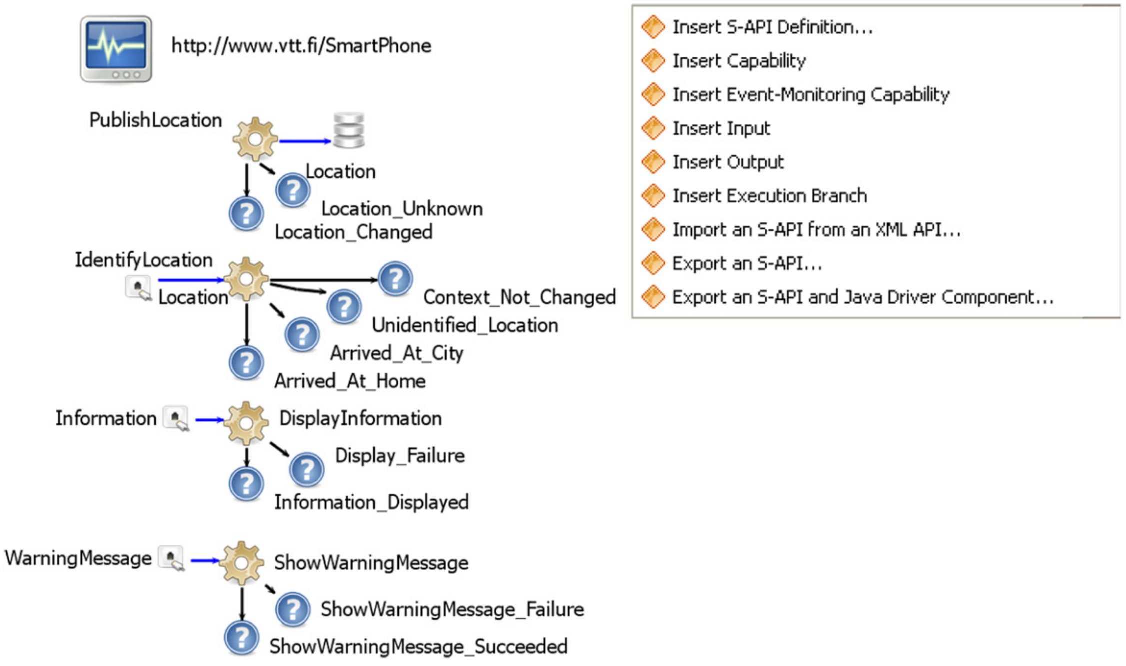


Fig. 5. The S-API of the personal assistant application and the toolset menu for S-API/driver component development.

command, whereas an output defines a name and type for the data produced by the command. A name and type is defined for each execution branch. Execution branches are defined for the typical and exceptional consequences of the execution of a command. The type defines whether the execution branch is a default or a failure execution branch.

The S-APIs are developed by using the Smart Modeler in the desktop environment. In order to improve usability, we modified the Smart Modeler so that its extension can also be a *toolset*, i.e., a dynamically created composite of tools that are specially selected and initialized for the model element(s) that the user has selected in the tool. We implemented then a toolset for the S-API development (in Fig. 5).

Figs. 5 and 6 depict example S-APIs for the smartphone and for the home automation system. The S-API of the smartphone provides the following commands:

1. The *PublishLocation* that publishes an event about the changed user's location to an SIB.
2. The *IdentifyLocation* that identifies the context of the user's location (e.g., home, work, or city).
3. The *DisplayInformation* that displays the defined information on the smartphone display.
4. The *ShowWarningMessage* that displays a defined warning message for the user of the smartphone.

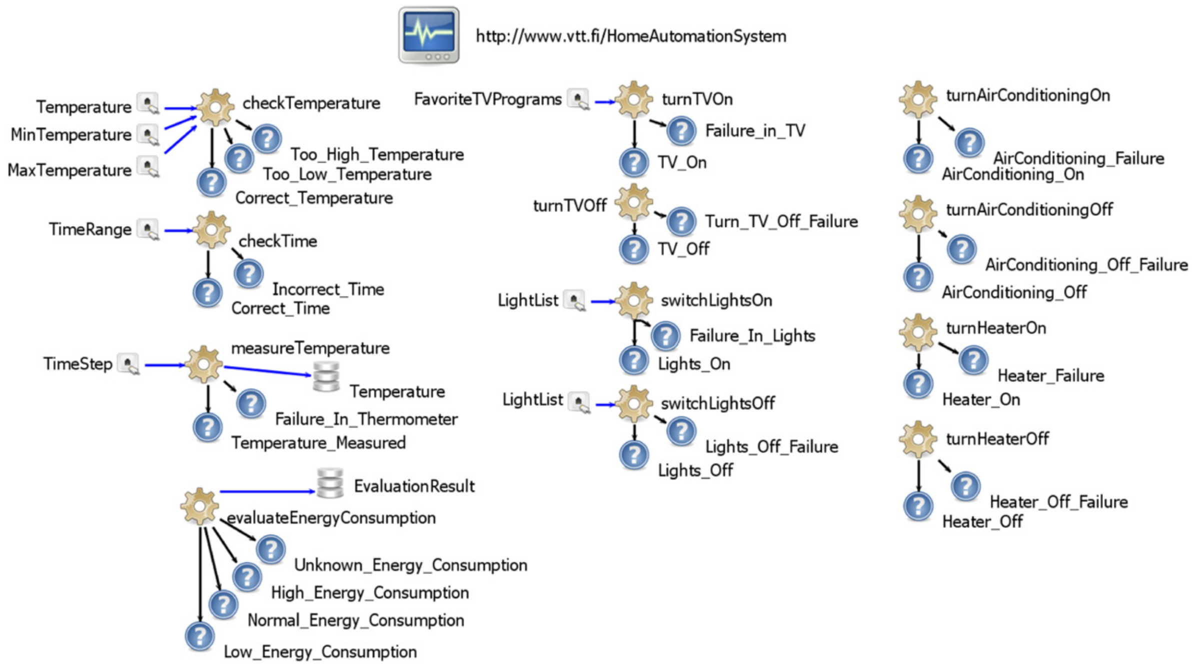


Fig. 6. The S-API of the home automation system.

The S-API of the home automation system provides the next commands:

1. The *CheckTime* that identifies whether the current clock time is within the defined range.
2. The *MeasureTemperature* that periodically measures temperature.
3. The *CheckTemperature* that identifies if the temperature is correct, too high, or too low.
4. The *SwitchLightsOn/Off* that switches the defined lights on or off.
5. The *TurnTVOn/Off* that turns the television off or on to display the specified TV programs.
6. The *TurnHeaterOn/Off* that turns the heating system on or off.
7. The *TurnAirConditioningOn/Off* that turns the air conditioning on or off.
8. The *EvaluateEnergyConsumption* that evaluates the energy consumption of the home environment and publishes the evaluation results to the SIB.

3.2. Integrated development process for end-user programming

This subsection discusses the activities related to our end-user programming approach and shows how these activities are connected to each other. Use of the approach requires both bottom-up development and top-down development efforts. Before end-user programming activities, software professionals must focus on the more difficult tasks and do bottom-up development that produces easy-to-apply S-APIs and driver components in the following steps (see Fig. 7):

1. *Modeling*—This step produces an S-API for the professional APIs. The existing professional APIs are used as a starting point in the development of S-APIs. The software professional has to inspect the existing professional APIs (see Fig. 7), decide which kinds of capabilities are provided for end-users, and finally create an S-API for the selected capabilities.
2. *Code generation*—This step automatically generates a skeleton for the driver component from the defined S-API. The generated driver component has initial implementations for the S-API and Execution API, and thus it is possible to continue directly to step 4 and to deploy and test the component. Otherwise the process continues to step 3.
3. *Integration*—The developer implements the resource allocation and commands' execution code and thus integrates the component(s) of the software system to the driver component.
4. *Deployment and testing*—This step will ensure that the driver component behaves as expected.

It is important to note that software professionals can use the top-down development method in the creation of the driver components too. For example, they can model the behavior and cooperation of the driver component and legacy software system and generate a wrapper implementation from the model. However, this article focuses only on the bottom-up development of S-APIs and driver components.

As depicted in Fig. 7, the S-API is used as an integration element between the bottom-up development and top-down development processes. The S-APIs and existing applications (if they exist) provide a starting point for end-user programming. The end-user uses top-down modeling that does not just outline the behavior of an application but precisely specifies and configures the behavior of his/her application in the next steps:

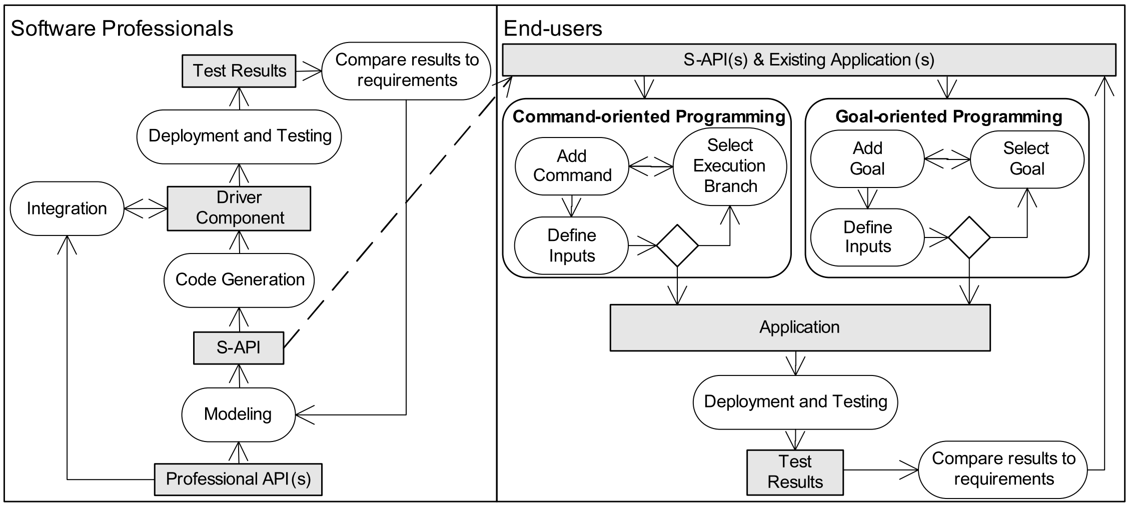


Fig. 7. An integrated development process for software professionals and end-users.

1. *End-user programming*—This is performed in the command-oriented mode or in the goal-oriented mode. The end-user can create a totally new application or modify an existing application for his/her purposes. The end-user programming is based on a visual editor and on the available S-APIs.
2. *Deployment and testing*—This step tests the behavior of the end-user's application. The end-user must execute the application in the target execution environment that will reveal (at least) part of the errors related to the application. The process continues to step 1 if the application does not behave as expected. Otherwise the application is ready to be utilized.

Testing efforts are required for the driver components and for the end-users' applications. However, testing issues are not the focal point of the article and are not discussed in more detail. The rest of the article focuses on S-APIs and driver component development and the command-oriented and goal-oriented end-user programming of applications.

4. An example of cross-smart space applications

Imagine a situation in which there are two kinds of stakeholders: software professionals and end-users. The first software professional develops a personal assistant application for smartphones whereas the second develops a home automation application. The end-user wants to use the capabilities from both of these applications and thus wants to compose an application for his/her needs from the capabilities that are provided in the APIs of the personal assistant application and the home automation application. The next sections describe how our end-user programming approach can be utilized in the creation of an application that integrates the personal assistant application and the home automation application through smart spaces.

4.1. Development of S-APIs and driver components

The creation of an S-API is performed in the next steps. Firstly, (s)he uses the *Insert S-API Definition* tool via the popup menu that is depicted in Fig. 5 and defines a name (Uniform Resource Identifier) for the S-API. Secondly, (s)he will specify the commands for the selected capabilities (using the *Insert Capability* and/or *Insert Event Monitoring Capability* tools) and define inputs (using the *Insert Input* tool), outputs (uses the *Insert Output* tool), and execution branches (using the *Insert Execution Branch* tool) for each command.

In the code generation step (in Fig. 7) the user uses the *Export an S-API and Java Driver Component* tool that exports the S-API in N3 format and produces a driver component skeleton for the S-API. In the generated driver component, the capabilities are always available and the commands just execute a code line that prints the name and the inputs of the command. Later, in the integration step, the developer can use the tools and code editors of the Eclipse environment and write actual code to the driver component that uses the professional APIs and handles allocation of the capabilities and execution of the commands. This typically requires a few hundred lines of coding. Of course, this number strongly depends on the target software component/system for which the driver component is implemented.

4.2. Command-oriented end-user programming in the desktop environment

Two options are provided for command-oriented programming: a visual graph-based editor (in Fig. 8) for the desktop environment and a list-based editor (in Fig. 9) for the mobile environment. In the desktop environment software

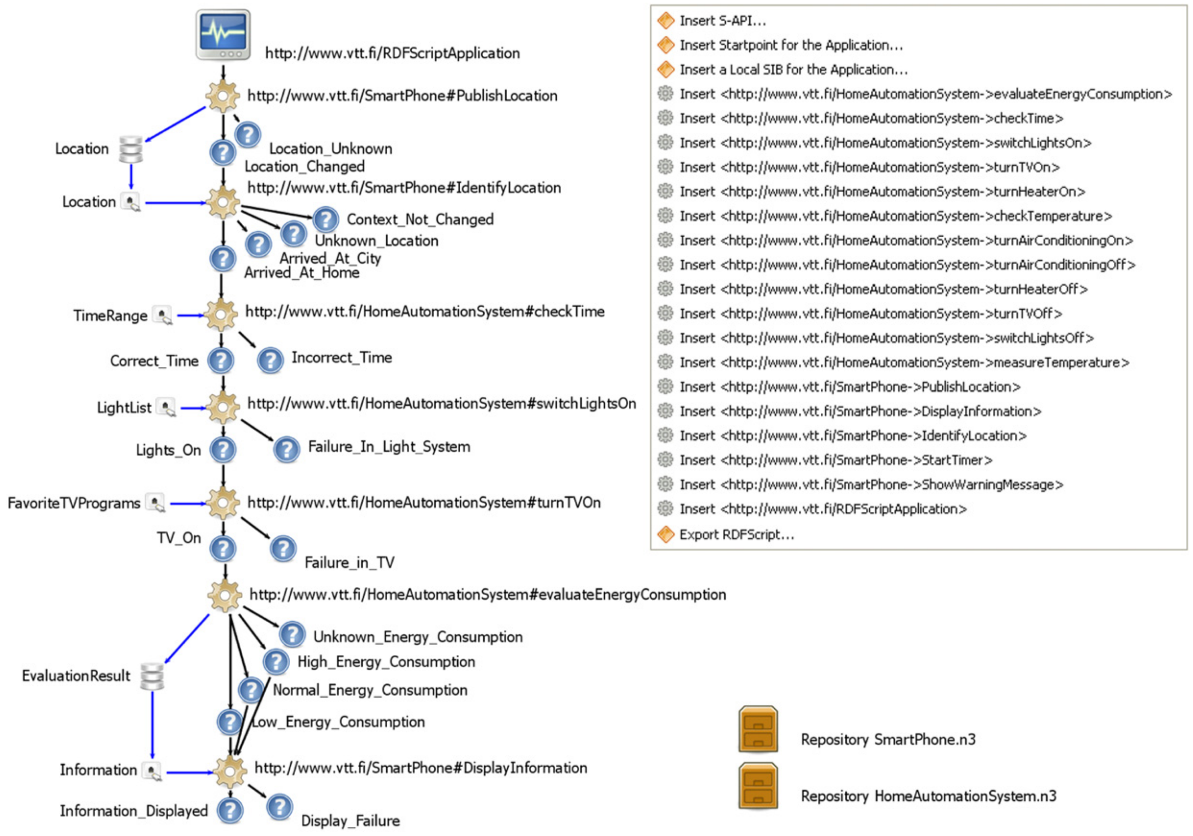

http://www.vtt.fi/RDFScriptApplication
http://www.vtt.fi/SmartPhone#PublishLocation
Location
Location_Unknown
Location_Changed
Location
http://www.vtt.fi/SmartPhone#IdentifyLocation
Context_Not_Changed
Unknown_Location
Arrived_At_City
Arrived_At_Home
TimeRange
http://www.vtt.fi/HomeAutomationSystem#checkTime
Correct_Time
Incorrect_Time
LightList
http://www.vtt.fi/HomeAutomationSystem#switchLightsOn
Lights_On
Failure_In_Light_System
FavoriteTVPrograms
http://www.vtt.fi/HomeAutomationSystem#turnTVOn
TV_On
Failure_in_TV
http://www.vtt.fi/HomeAutomationSystem#evaluateEnergyConsumption
EvaluationResult
Unknown_Energy_Consumption
High_Energy_Consumption
Normal_Energy_Consumption
Low_Energy_Consumption
Information
http://www.vtt.fi/SmartPhone#DisplayInformation
Information_Displayed
Display_Failure
Insert S-API...
Insert Startpoint for the Application...
Insert a Local SIB for the Application...
Insert <http://www.vtt.fi/HomeAutomationSystem->evaluateEnergyConsumption>
Insert <http://www.vtt.fi/HomeAutomationSystem->checkTime>
Insert <http://www.vtt.fi/HomeAutomationSystem->switchLightsOn>
Insert <http://www.vtt.fi/HomeAutomationSystem->turnTVOn>
Insert <http://www.vtt.fi/HomeAutomationSystem->turnHeaterOn>
Insert <http://www.vtt.fi/HomeAutomationSystem->checkTemperature>
Insert <http://www.vtt.fi/HomeAutomationSystem->turnAirConditioningOn>
Insert <http://www.vtt.fi/HomeAutomationSystem->turnAirConditioningOff>
Insert <http://www.vtt.fi/HomeAutomationSystem->turnHeaterOff>
Insert <http://www.vtt.fi/HomeAutomationSystem->turnTVOff>
Insert <http://www.vtt.fi/HomeAutomationSystem->switchLightsOff>
Insert <http://www.vtt.fi/HomeAutomationSystem->measureTemperature>
Insert <http://www.vtt.fi/SmartPhone->PublishLocation>
Insert <http://www.vtt.fi/SmartPhone->DisplayInformation>
Insert <http://www.vtt.fi/SmartPhone->IdentifyLocation>
Insert <http://www.vtt.fi/SmartPhone->StartTimer>
Insert <http://www.vtt.fi/SmartPhone->ShowWarningMessage>
Insert <http://www.vtt.fi/RDFScriptApplication>
Export RDFScript...
Repository SmartPhone.n3
Repository HomeAutomationSystem.n3

Fig. 8. The RDFScript toolset menu and a smart space application that integrates the personal space with the home space.

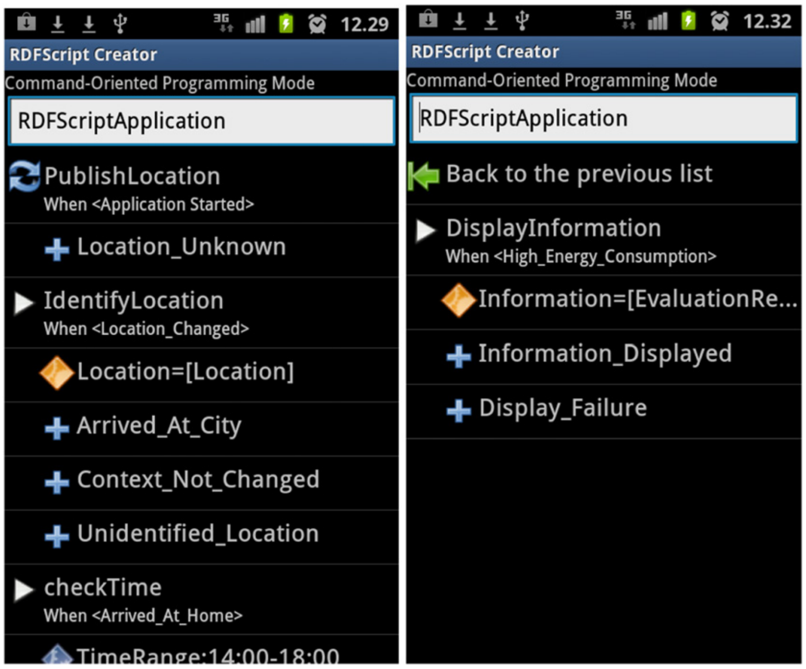
professionals can use the RDFScript toolset (in Fig. 8) for simulation and testing purposes, whereas more advanced end-users can use the S-APIs and driver component toolset and create the S-APIs/driver components for their own applications. Fig. 8 depicts an application that identifies the user location and then if the user is at home and the time is between 18:00 and 20:00 (defined in the time range parameter), it firstly switches the defined lights on, then turns the television on to show the end-user's favorite TV program, evaluates recent energy consumption at home, and finally displays the evaluation results for the user.

In the visual editor the user attaches the desired S-APIs to the model by using the "*Insert S-API*" tool and then defines (using the "*Insert Start Point for the Application*" tool) a name (URI) for his/her application. Now it is very easy to insert commands to the application. The developer just needs to use the *Element Importer* tool that shows the commands of S-APIs in the popup menu and will then automatically insert the selected command at the beginning of the application or to the chosen execution branch of the application. The user must then configure the inputs of the command by defining an *absolute value* for each input or by then connecting the inputs to the outputs of the previous commands in the command sequence. Thus the input will have an *absolute value* given by the user or a value that a previous command has produced as an output. The type checking that uses the types of inputs/outputs defined in S-APIs ensures that only compatible input/output connections are defined in the editor. It is important to note that the user does need to connect all outputs of commands to the inputs of the next commands. However, application execution requires that all inputs are configured in the application. The application is finally stored in N3 format by using the *Export RDFScript* tool.

4.3. Command-oriented end-user programming in the mobile environment

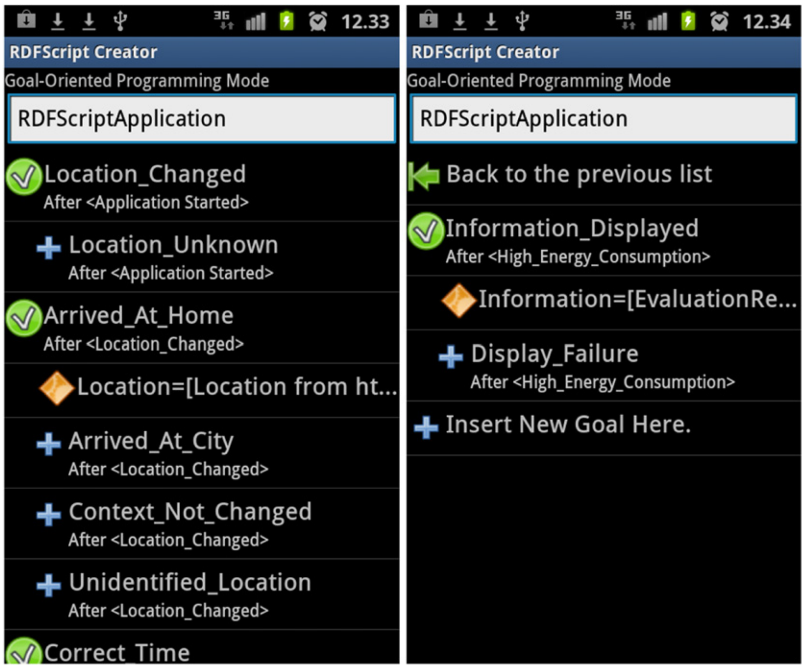
Mobile devices are always with the user and therefore enable the user to take advantage of the possibilities of available smart spaces. Mobile devices have smaller screen sizes than desktop computers when comparing both the physical size and the resolution. In addition, nowadays mobile devices are often touchscreen devices, where the user controls the phone with his/her fingers. Thus, it is difficult to create smart space applications via visual graphs (like in the Smart Modeler) in a touchscreen-enabled mobile device. Therefore, we decided to develop a new editor called the RDFScript Creator for the Android platform and touchscreen-enabled mobile devices.

The RDFScript Creator mostly provides the same features as the Smart Modeler: Firstly, the user can easily download applications and S-APIs from smart spaces, then utilize the S-APIs and command sequences of existing applications in the


12.29
RDFScript Creator
Command-Oriented Programming Mode
RDFScriptApplication
PublishLocation
When <Application Started>
Location_Unknown
IdentifyLocation
When <Location_Changed>
Location=[Location]
Arrived_At_City
Context_Not_Changed
Unidentified_Location
checkTime
When <Arrived_At_Home>
TimeRange:14:00-18:00

12.32
RDFScript Creator
Command-Oriented Programming Mode
RDFScriptApplication
Back to the previous list
DisplayInformation
When <High_Energy_Consumption>
Information=[EvaluationRe...
Information_Displayed
Display_Failure

Fig. 9. The command-oriented list view.


12.33
RDFScript Creator
Goal-Oriented Programming Mode
RDFScriptApplication
Location_Changed
After <Application Started>
Location_Unknown
After <Application Started>
Arrived_At_Home
After <Location_Changed>
Location=[Location from ht...
Arrived_At_City
After <Location_Changed>
Context_Not_Changed
After <Location_Changed>
Unidentified_Location
After <Location_Changed>
Correct_Time

12.34
RDFScript Creator
Goal-Oriented Programming Mode
RDFScriptApplication
Back to the previous list
Information_Displayed
After <High_Energy_Consumption>
Information=[EvaluationRe...
Display_Failure
After <High_Energy_Consumption>
Insert New Goal Here.

Fig. 10. The goal-oriented list view.

creation of new applications, and finally publish applications to the smart spaces. The RDFScript Creator provides two modes for end-user programming: the command-oriented programming mode and the goal-oriented programming mode. End-user programming is performed in a list view of hierarchical lists. Figs. 9 and 10 depict command-oriented and goal-oriented representations for the application in Fig. 8.

In the command-oriented mode, the list view (Fig. 9) displays the commands, inputs and execution branches of the commands. The user can add and remove commands, define inputs for commands, and define when the command is executed, i.e., connect the command to the output branch of the previous command if the command is not the first one in the sequence. The idea is to provide natural language sentences, e.g., *when the location is home and when the time is correct, turn on the TV*. The links enable the user to navigate to the sublists or to the upper-level lists. If an event source relates to a goal or a command it is shown with an icon in the left-hand side of the list item. This icon means that every time an event (e.g., a location change event) is received, the execution sequence is executed again starting from this command/goal.

4.4. Goal-oriented end-user programming in the mobile environment

To further support application development, we created another viewpoint for the development—goal-oriented programming that enables the user to create the application by selecting the goals that are achieved by executing the different commands (in Fig. 10). In this viewpoint the user defines the goals that should be obtained in the application. The user adds goals and defines inputs for goals, and in this way produces a goal sequence for his/her application. The idea of this mode is to provide natural language sentences, such as *if location is changed, and I arrived at home, then the TV is turned on* and so on.

A command sequence can be represented as a goal sequence and vice versa. Thus the command-oriented programming and goal-oriented programming are not exclusionary programming methods; it is possible to start with the command-oriented programming and then continue with the goal-oriented programming, and finally use the command-oriented programming for finalizing the application. Furthermore, the user can save the application to the SIB, modify it in the desktop environment, and later continue testing of the application with his/her mobile device.

4.5. Summary

Our end-user programming approach, along with the supporting tools, provides several advantages compared to existing approaches:

- *Semantic End-User APIs*—Contribute to the reuse of information/operational capabilities that are available in the Professional APIs in end-user programming that can be performed either in the desktop environment, the mobile environment, or partially in both. The reference tools assist in the creation of S-APIs/driver components.
- *Supports command-oriented end-user programming*—The execution branches of commands facilitate creation of execution sequences. The end-user just has to select an execution branch and then a command that is automatically connected to the chosen execution branch. The command sequence will specify the end-user's application.
- *Supports goal-oriented end-user programming*—The end-user composes application of the goals that are automatically produced from a set of S-APIs or from the command sequences. The goal-oriented programming provides a new viewpoint for end-user programming and is a promising way for creating applications in mobile devices. Furthermore, a command sequence can be represented as a goal sequence and vice versa. Thus the command-oriented programming and goal-oriented programming methods are not exclusionary programming methods, but they can be successively used in application development.
- *Reuse of capabilities, goals, and models*—Unlike existing approaches, the S-APIs do not only support the reuse of capabilities but also the reuse of goals. Firstly, the S-APIs assist the end-users to utilize these capabilities of existing software components in their applications. Secondly, the execution branches of commands specify the alternative situations to which the executed commands will lead. Both the commands and the execution branches and goals of the commands can be utilized in end-user programming. Thirdly, unlike the source code of a traditional (e.g., Java/C++) application, the RDFScript application does not only specify the command sequence for the application but also the alternative execution branches that relate to the commands. Thus, it is easy for end-users to modify the execution flow of an existing application and to produce new applications for different purposes.

5. Evaluation method and results

A small-scale usability test was performed for the end-user programming approach. The target was to study the usefulness and benefits of the command-oriented and goal-oriented programming modes and to evaluate how well the users can perform the different steps of these two programming modes.

5.1. Experiment setup

Use of the approach requires S-APIs and driver components. The creation of S-APIs for the smartphone and for the home automation system took a few hours and implementation of the driver components took 3 working days in total. The driver component skeletons that were transformed from the S-APIs greatly assisted implementation of the driver components. The driver component for the smartphone required 254 lines of coding (the skeleton size was 232 lines) and the driver component for the home automation system required 244 lines of coding (the skeleton size was 415 lines).

In practice there must be a mechanism that is capable of (semi-)automatically joining the available/correct smart spaces or the user must know how to connect to the available smart spaces. However, the configuration of a smart environment is not the focal point of this article but it is just assumed that the correct smart spaces are selected for development and execution of a cross-smart space application. Thus, a ready-configured smart environment was provided for end-user programming in the tests.

Evaluators

Our goal was to test how well the approach suits various kinds of users. Thus, eight non-programmers and eight programmers participated in the experiment. The non-programmers were users who did not have previous experience of programming but who were interested in developing applications for their own purposes. In the programmers' group, there were four experienced programmers and four software enthusiasts. The experienced programmers were software developers who work in the software industry and have a deep understanding of software development methods, processes, and programming languages. The software enthusiasts were people who do not work in the software industry but are capable of producing small applications using programming languages such as Java/C/C++.

Evaluation method

The following tasks were conducted:

- In task 1, the participants had to use either command-oriented or goal-oriented programming and compose an application that switches lights on, turns the television on, checks temperature and finally turns air conditioning on if the temperature is too high.
- In task 2, the participants had to change the programming mode and create the same application again.
- In the task 3, the users had to modify a ready-made application that controls the heating/air conditioning system at home. The users utilized command-oriented programming and added commands to the application that would show a message on the user's smartphone if there was a failure in the heating or air conditioning system.

Multiple iterative usability test and improvement cycles were performed for the RDFScript Creator. The RDFScript Creator had to be improved between the evaluations due to evaluators' difficulties to configure commands' inputs. Thus, each evaluator performed the test only once. The evaluators used Google Nexus S/Samsung Galaxy S II mobile phones and the latest version of the RDFScript Creator in the tests. The evaluators did not have previous experience of the Android platform. Thus, in the beginning, we had to provide usage instructions/guidance for the evaluators about the approach and the user interface of the Android platform.

5.2. Usability test and improvement cycles

The following improvements were made to the S-APIs and the RDFScript editor:

The naming of commands and execution branches were improved in S-APIs. In the beginning just a name, type and default value were defined for the inputs of the commands. The usability tests revealed that better guidance is needed to assist the definition of inputs. Thus, we modified the approach so that the S-API can also define a unit for an input value and provide examples to assist the definition of inputs. This information is then displayed for the user in the RDFScript editor.

The output–input dependences between commands were difficult for the non-programmers. The home automation system has the *MeasureTemperature* command that measures a temperature and the *CheckTemperature* command that takes the measured, minimum and maximum temperature values as an input and checks if the temperature is correct, too high, or too low. In the first usability tests the users had to first add the *MeasureTemperature* command to the sequence and then define that the *CheckTemperature* command uses the measured temperature value as an input. This was difficult for the users and thus we decided to modify the editor so that it would recommend commands that produce suitable input data for a selected command. As a result, the user can add a command and define its inputs manually or use automatic input values produced by other commands. For example, if the user decides to use the automatic temperature value in the *CheckTemperature* command, the editor automatically adds the *MeasureTemperature* command (if needed) to the sequence and then connects its output to the temperature input of the *CheckTemperature* command. This greatly improved the editor's usability and in later tests it was much easier for the users to perform tasks 1 and 2.

Copy–paste functionality was also added to the editor to facilitate the creation of warning messages in task 3.

5.3. User experiences

Fig. 11 depicts (min. and max.) execution times for tasks 1, 2, and 3 and average execution times for programmers and non-programmers. As can be seen, tasks 1 and 2 were faster to perform in the command-oriented mode for the programmers and non-programmers. The created applications were checked and end-user programming was continued until the tasks were performed correctly. As a summary, a total of 3 or less end-user programming and testing cycles were required for tasks 1, 2, and 3 in the command-oriented (COP) and goal-oriented programming (GOP) modes.

The average time to perform tasks 1 and 2 with COP was 7 min and 6 s for programmers and 10 min and 30 s for non-programmers. Furthermore, the average time to perform tasks 1, 2 and 3 with COP was 15 min for programmers and 16 min and 24 s for non-programmers. The COP follows more ordinary use of structural programming languages such as Java and C++, and therefore the programmers are typically faster in using the command oriented approach. However, it is important to note that the non-programmers were faster than programmers in task 3 that was also performed with COP.

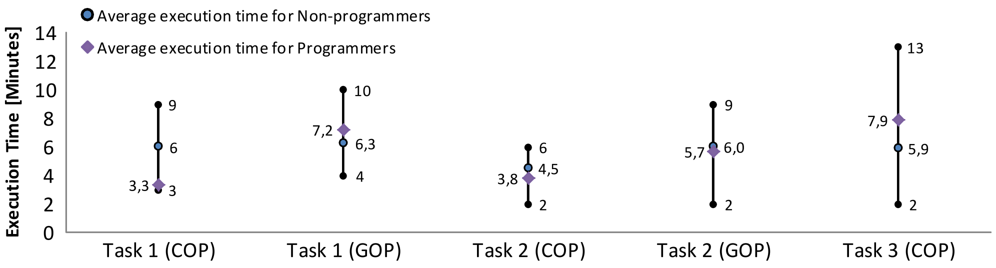


Fig. 11. The minimum, maximum, and average execution times in minutes for tasks 1, 2, and 3.

Surprisingly, the non-programmers were slightly faster in use of GOP. The average time to perform tasks 1 and 2 with GOP was 12 min and 54 s for programmers and 12 min and 20 s for non-programmers. The explanation may be the fact that the goal oriented end-user programming approach simulates the end-users' way of thinking: how to catch the goal whereas the programmers prefer the command-oriented thinking.

In summary, the average time to perform tasks 1 and 2 with COP and GOP and task 3 with COP was 28 min and 42 s for non-programmers and 27 min and 54 s for programmers. Thus, the differences between non-programmers and programmers were quite small and it seems that previous programming experience does not give such a great advantage in the use of the end-user programming approach. Furthermore, for both groups use of COP was faster than use of GOP.

Fig. 11 depicts the average, minimum, and maximum values for results of the multi-selection questions. As can be seen, both the programmers and non-programmers were quite satisfied with the approach. However, use of the approach was easier for the programmers than non-programmers, and, the programmers had more motivation to create applications to assist them in everyday life. It is understandable because of their work. It is in their interest to innovate new applications. In evaluators there was only one non-programmer that had no interest in creating applications for his/her needs. Our impression is that it takes some more time from non-programmers to get familiar with the end-user programming approach and tool than from professional programmers before inventing what to apply them. As can be seen from Table 1, the evaluators could accept a small fee for a device/product if it were possible to use its functionalities in end-user programming.

In summary, both groups experienced that the approach helps in end-user programming. Moreover, both groups preferred the command-oriented programming mode to the goal-oriented-programming mode (row 13 in Table 1). Furthermore, it is noteworthy that the non-programmers preferred the command oriented approach (see rows 13 and 14 in Table 1) more than the goal oriented one. This may indicate that the command oriented approach is easier to learn than the goal oriented one. Anyway, professional programmers were nearly equally confident of both approaches.

In the testing groups, there were two non-programmers and three programmers that preferred the GOP mode or equaled it with the COP mode. However, the both testing groups wanted to typically create quite small applications (i.e., applications that have less than 10 commands). In tests it was easy for both groups to find commands, to insert the required commands to a command-sequence, and configure the inputs of the commands. However, it is important to note that only two S-APIs were used in the tests. Thus, when the number of commands/S-APIs increases, it is more difficult to find, select and configure commands for an application. There are typically a greater number of goals than commands because multiple goals are often related to a single command. Thus, in tests for both groups searching for goals in GOP mode was more difficult than searching for commands in COP mode.

5.4. *Command-oriented programming vs. goal-oriented programming*

The command-oriented programming mode was easier than the goal-oriented programming mode for most of the programmers and non-programmers. In addition, it seems that the command-oriented programming is a faster way to create applications. However, it is easy to move between these two programming modes. Thus, we think that the goal-oriented programming does not replace the command-oriented programming but can enhance application development: The goal-oriented programming mode provides an alternative viewpoint for application development and can assist end-users in comprehending the behavior of an application.

The usability tests revealed that the RDFScript editor still needs further development. The following lists the observations collected during the experiments, and suggests improvements that could enhance usability of the different steps of both programming modes:

Insertion of commands—The commands of S-APIs are displayed in alphabetical order for the user. In the tests, commands of two Driver KPs were used and thus searching for commands was not difficult for the users.

Insertions of goals—Searching for goals was especially difficult for the users. The reason for this is the fact that several execution branches typically relate to a command. Thus, the number of goals is much bigger than the number of commands.

The enhanced editor should further assist the usage of capabilities for smart environments that provide a great number of commands/goals for end-user programming: Firstly, it should be possible to select the S-APIs which are actively used and

Table 1
The results of the multi-choice questions.

Question number	Statement/question	Non-programmers			Programmers		
		Average	Minimum	Maximum	Average	Minimum	Maximum
1	I have many ideas of small applications that could make my everyday life easier (1 = Agree and 6 = Disagree).	**3.6**	2	6	**2.6**	1	5
2	I would like to create applications that use the functionalities of separate computing environments (e.g. smart phone, computer, GPS navigator, Web services, home automation system, entertainment system, and the computer / Electronic Control Unit (ECU) of my car) (1 = Agree and 6 = Disagree).	**2.5**	2	3	**2.3**	1	5
3	I would accept to pay a little fee for a device/product if it is possible to use the device's/product's functionalities in my applications (1 = Agree and 6 = Disagree).	**2.8**	1	4	**2.3**	1	5
4	It is easy to create applications of commands (1 = Agree and 6 = Disagree).	**1.8**	1	2	**1.7**	1	2
5	It was easy to find commands in the tasks (1 = Agree and 6 = Disagree).	**2.0**	1	4	**1.4**	1	2
6	It was easy to add new commands to the command-sequences (1 = Agree and 6 = Disagree).	**2.0**	1	3	**1.9**	1	3
7	It was easy to configure command's inputs (1 = Agree and 6 = Disagree).	**2.8**	2	5	**2.1**	1	4
8	It is easy to see what happens in the created command-sequence (1 = Agree and 6 = Disagree).	**2.8**	1	4	**2.6**	2	3
9	It was easy to find a desired goal (1 = Agree and 6 = Disagree).	**2.9**	2	4	**1.9**	1	3
10	It was easy to configure goal's inputs (1 = Agree and 6 = Disagree).	**3.4**	2	5	**2.7**	1	5
11	It was easy to insert goals to the goal-sequence (1 = Agree and 6 = Disagree).	**2.8**	1	4	**2.3**	1	5
12	It is easy to see what happens in the created goal-sequence (1 = Agree and 6 = Disagree).	**3.1**	2	4	**2.6**	1	4
13	I think that it is easier to create an application through the command-oriented programming than by using the goal-oriented programming (1 = Agree and 6 = Disagree).	**1.9**	1	3	**3.0**	2	6
14	I think that it is easier to see the application logic in the goal view than in the command view (1 = Agree and 6 = Disagree).	**4.6**	3	6	**3.3**	1	5
15	How complex applications would you like to create? (Applications that have 1 = 1–4, 2 = 5–10, 3 = 11–20, and 4 = over 20 commands/goals.)	**1.9**	1	3	**1.9**	1	4
16	The used approach helps to the end-user programming (1 = Agree and 6 = Disagree).	**2.0**	1	3	**1.4**	1	2
17	The command-oriented programming is simple to use and easy to learn (1 = Agree and 6 = Disagree).	**2.6**	1	5	**1.6**	1	2
18	The goal-oriented programming is simple to use and easy to learn (1 = Agree and 6 = Disagree).	**3.1**	2	5	**2.1**	1	3
19	Please rate the used end-user programming approach (1 = Not satisfactory and 6 = Excellent).	**4.5**	3	5	**5.1**	4	6

thus reduce the commands/goals visible to the user. The available commands/goals are now presented in alphabetical order. It would be nice if the editor could show the most suitable commands for a chosen execution branch first. Furthermore, the S-API could define keywords for commands to facilitate searching for commands and goals.

Definition of inputs—This is performed via the same view in the command-oriented and goal-oriented programming modes. In both cases, the configuration of inputs was easy for the users after usability improvements (described in 5.2) were made for the editor. The textual input values of commands are configured now through a text field. Specialized input controls (e.g., *sliders for numeric input values* and *check boxes for fixed input values*) and validations for the input values would further improve the editor's usability.

Selection of execution branches/goals—Descriptive names were given for output branches in S-APIs used in the experiments. Thus, it was easy for the users to select the correct execution branches/goals for new commands and goals.

5.5. Summary

The experiment revealed that S-APIs/Driver KPs have a significant effect on the usability of the approach and the fact that the end-user testing/feedback helped to reveal usability flaws in the S-APIs. In usability tests the end-users learned to use the end-user programming approach quickly. In fact, it took only few minutes for the participants to compose a new smart space application in tasks 1 and 2 and to modify an existing application in task 3.

It seems that the approach is suitable for relatively small applications that contain less than 20 commands. In the case of more extensive applications, it is difficult to comprehend the overall behavior of the application. Many participants stated that there should be an overall view for the command/goal sequence. However, the small screen of a mobile device sets limitations. One solution to this problem is to limit the details (e.g., inputs and execution branches) that are displayed in the list views. Furthermore, it might be a good idea to packetize command/goal sequences to composites (e.g., this approach is used in Smart Modeler [12]) so that the user could create new applications quickly from available command/goal composites.

6. Discussion

The S-APIs and execution components will significantly ease end-user programming. Although the reference toolset is a prototype, it proves that the creation of S-APIs and driver components can be conducted iteratively, smoothly and cost effectively. So far, the reference toolset supports end-user programming of applications for smart environments only. However, we believe that it is possible to use the approach in other kinds of execution environments too.

The goal-oriented programming mode enables the end-user to create the application in a way in which (s)he can think of the goals (s)he wants to achieve in his/her application instead of concentrating on the commands that have to be executed in order to fulfill a goal. End-user programming is easy and quick to do with the mobile editor: usually it takes only few minutes to create an application. Naturally, the time it takes depends on the application, the user's background and knowledge of the target environment, and the available commands/goals.

6.1. Considerations

The presented approach is suitable only for software systems that provide professional APIs for developers. Furthermore, there must be developers that want to implement S-APIs/driver components for end-user programming. An important motivation for this work is the fact that the S-APIs/driver components produce additional value and can gain new users for existing software systems and components. Furthermore, the provided tools facilitate S-API/driver component development and thus lower the threshold for implementation of these components.

The approach offers a new way to use the capabilities of professional APIs in end-user programming. However, the following issues may limit use of the approach.

- *The scalability of the approach*—Mapping professional APIs with complex behavior to simple commands can result in a large number of commands. Consequently, the user will have to choose an appropriate command from a large number of commands. Thus, here is a need for command searching and command selection methods that assist the end-user in discovering the correct commands for his/her application. Furthermore, the S-APIs could give meta-information related to commands to assist in the search and selection of commands for the situation in hand.
- *Limited expressiveness*—The RDFScript language is designed for end-user programming and thus provides only limited expressiveness compared to programming languages such as Java/C. For example, it does not offer if–else,while, for, and try–catch structures that exist in many programming languages.
- *Limited access to the capabilities of professional APIs*—Only those capabilities for which there are Driver KPs provided can be used in the end-user programming. The S-API creators must select the capabilities that are made available to the end-users. Typically the S-APIs provide access only to part of the capabilities that are available in the professional APIs. However, it is possible to develop extended versions of the S-APIs/driver components later that will better cover the capabilities of the professional APIs.
- *Common ontologies are needed*—Integration of the capabilities of the professional APIs is based on input and output data that is shared between the commands of Driver KPs. The common ontologies (see, for example, [13]) are needed for the input and output data of commands.
- *Need for run-time adaptation*—The run-time adaptation allows part of the design decisions to be postponed until run-time [14]. For example, the evolution of S-APIs/ontologies can cause inconsistencies between inputs and outputs of commands. Therefore, there is a need for execution components that adapt the output data to different input formats and thus enable the command to be used together with the commands of S-APIs that are based on different ontologies and ontology versions.
- *No direct support for testing driver components*—However, it is important to note that the S-API can be utilized in driver components testing too. For example, command sequences can be generated from the S-API to support testing of driver components.
- *No direct support for testing end-user applications*—For example, it should be possible to collect the trace data about delays and errors (e.g., regarding exceptions) of the application, driver components, and executed commands. A separate visualizer tool could visualize the trace data and thus allow the end-user to evaluate the behavior of his/her application.

Although the approach has been applied to the development of smart space applications, it has not yet been validated in an industrial context. However, the advantage of the approach is in using the capabilities of legacy software systems in end-user programming, and this will motivate industrial use of the approach and supporting tools. The prerequisite is that developers be familiar with the approach.

6.2. Guidelines for S-API/driver component developers

The S-APIs have a significant effect on the usability of the approach and the S-API developers must consider end-users while the S-APIs are developed. If the software developer does not have knowledge of what the end-users want to do, (s)he will probably create an S-API that offers commands and goals that do not make sense to the end-users or the end-users may not comprehend how the commands should be used. The experiments showed that the S-APIs should be developed in an iterative fashion in co-operation with end-users. More integration effort is needed if the S-API of an implemented driver component is changed. Thus, end-user testing of S-APIs should be performed as early as possible, before a lot of development effort is put in to the driver components.

The following lists guidelines to assist in development of the S-APIs/driver component:

Co-operation with end-users—End-user testing/feedback is necessary to ensure that the S-APIs are suitable for end-user programming. An important benefit of the approach is that end-user testing of S-API(s) does not require the driver components but can be started after the S-API(s) are available. Thus, testing can be started at a very early stage with the outlined application. For example, the developer must consider how fine-grained commands are provided in the S-API. The large number of commands that do only small tasks enable the end-user to tailor the behavior of his/her application more. However, in this case the end-user must compose a longer command sequence for his/her application.

Scenario-based development of S-APIs—It is the S-API developers' responsibility to select the capabilities that are provided for end-user programming. We have noticed that application scenarios facilitate the development of S-APIs. The developer can first outline an application that utilizes the new S-API. (S)he can then develop an initial version of the S-API and create an outlined application by using the end-user programming approach and the created S-API.

Give descriptive names for commands, inputs, outputs, and execution branches—The naming conventions used will greatly affect the usability of the S-APIs. The end-user must find the correct commands for his/her application, and thus a very descriptive and systematic naming convention is needed for commands. For example, a naming convention in which the name of a command consists of a substantive and a verb (e.g., Lights_Turn_On) could facilitate searching for commands.

Minimize the number of inputs for commands— Commands that do not require input parameters at all are easier to use in end-user programming. Thus, commands that have a great number of inputs make end-user programming laborious. For the sake of simplicity and ease of use, default values for most command inputs are to be defined so that only specific inputs are needed to be configured by the end-user.

Maximize the number of the outputs for commands—The processing results should be made available for other (or not yet available) commands that can possibly use these results for different purposes in the future.

Implement extensive error handling mechanisms for the driver components—The exceptions thrown by a driver component are delivered to the Application Executor that can display the error messages for the user. These exceptions should provide descriptive information about missing, wrongly formatted (e.g., a textual value is given for an input that requires an integer value) or illegal input values. Our usability tests proved that this kind of support will greatly help the testing of cross-smart space applications and makes correction of errors easier for end-users.

6.3. Future research directions

The reference toolset, RDFScript language, visualization tools and agent-based execution environment assist usage of the approach in the context of smart space applications. However, there is still a lot of work to do in the reference toolset. The following lists possible directions of future research for the approach and for the reference toolset:

- *Usability improvement of the toolset*—The aim of the reference toolset is not to provide optimized tools for the approach but just to test the feasibility of the approach. Thus, although the reference toolset facilitates end-user programming, more usability testing/improvements are still needed for the toolset.
- *Improvement of the execution environment*—The current implementation of the execution environment does not suit real-time applications well. Firstly, the execution is based on scripts and interpretation, and therefore, the execution speed of an RDFScript application is slower than the execution speed of binary code. Furthermore, the execution of RDFScript applications increases the workload of SIBs and network traffic between KPs and SIBs. Thus, improvement is needed for optimizing the performance of RDFScript applications. This can be done in two ways: Firstly, the SIBs can act as a temporary data store, i.e., for triplets that are only delivered for the subscribers through SIBs. Secondly, the Driver KPs can publish information that enables direct messaging channels to be established between the Application Executors and Driver KPs. In that way, performance would be greatly improved. Unfortunately, this increases complexity and may set limitations for the execution of the RDFScript applications too.
- *Extensions to the S-APIs*—The S-APIs could also contain run-time information for optimizing the execution of smart space applications. It is easy to implement a Driver wrapper component that both delivers the requests for the actual driver component and at the same time produces measured quality information for the capabilities of the driver component. For example, quality information such as number of errors, execution times, and execution delays can be produced and embedded in S-APIs at run-time, stored to the SIB, and later utilized in the execution of the smart space applications. For example, it could be possible to use this quality information for selecting capabilities that provide either the best

possible performance or those that provide secure and reliable processing capabilities with lower performance. Of course, the Application Executor must have a quality-aware capability selector component that is capable of using the provided quality information in the selection of capabilities.

7. Related work

The section provides a brief survey of work related to ours. We developed a bottom-up development methodology previously (see [15,12]), in which software professionals use Model-Driven Development (MDD) and Ontology-Driven Software Engineering (ODSE) practices in creating software agents for smart space applications. Furthermore, the Ontology-driven Piecemeal Software Engineering (OPSE) approach contributes to the development of smart spaces by using the software pieces that follow the design principles defined by means of ontologies in incremental software development [16]. In addition, the Agile Smart Space Development and Evolution methodology introduces a scenario-based development process for smart spaces that consists of smart space initiation, smart space development, smart space operation and evolution phases [7]. We have also used the top-down and the bottom-up modeling techniques in the incremental smart space application development [12]. Firstly, a smart object, its software, and installed KPs form an increment for a smart space application. Secondly, the top-down modeling is used to support teamwork; it outlines the overall behavior of the smart space application and thus facilitates communication between different stakeholders. Thirdly, the bottom-up modeling produces a more concrete description of the behavior of a single smart object and its KPs, and thus facilitates implementation of the parts of the smart space application. This article extends our previous work and introduces an integrated process for end-user programming of cross-smart space applications.

A smart space application can be understood as an agent-based application that has similar kinds of features to those of the service-oriented applications and the component-based applications. The next paragraphs provide comparisons to these three implementation techniques.

Agent-based development— Booch [17] identifies three ways for tackling complexity in software: (i) decomposition, in which a large problem is divided into smaller, more manageable chunks, each of which can then be dealt with in relative isolation; (ii) abstraction that defines a simplified model of the system, emphasizing some of the details or properties while suppressing others; and (iii) organization that identifies and manages interrelationships between various problem-solving components. The agent-oriented philosophy supports decomposition, abstraction, and organization of the complex systems [18]. The basic 3-layer agent structure is common to Agent Programming Language (APL) approaches such as ALPHA [19] and S-APL [20]. For example, the S-APL agent architecture consists of the behavior engine, a declarative middle-layer, and a set of sensors and actuators [20]. The S-APL is designed for bottom-up development, i.e., the S-APL configures a behavior for a single agent at time. Like the S-APL, the approach presented in this article is also based on RDF-based language and on the semantic data storage that is used for the data and code. However, unlike existing approaches, our approach supports top-down development that enables end-users to create applications that coordinate the behavior of legacy software systems through S-APIs, Execution APIs, and driver components that are executed in the provided agents.

Service-oriented software development—A Service-Oriented Architecture (SOA) describes both the services of a distributed software system, the interactions that occur among the services to realize certain behavior, and maps the services into one or more implementations in specific technologies [21]. The Web implementation of REpresentational State Transfer (REST) architectural style [1] is used in many projects for interfacing with smart appliances [22]. In REST each request from the client to the server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server, i.e., the session state is kept entirely on the client [1]. The WebHook pattern enables the client to avoid constant polling of the sensors and to 'subscribe' to a resource by passing a callback URL to be notified whenever a defined condition is fulfilled [23]. In summary, the REST API and the callbacks make the controlling and monitoring the resources of smart appliances possible. Our approach can be used in service-oriented software development too. The S-API and Execution API can be implemented as a Web service, whereas event-monitoring can be based on the WebHook pattern. In practice, the approach partially follows the REST architecture style, i.e., the information required for processing is (almost) entirely kept on the client-side (in Application Executors). However, resource allocation and event-monitoring require information about the users of event-monitoring commands that is stored on the server-side (to the Drivers).

Component-Based Software Development (CBSD)—This involves multiple roles [24]. Framework builders create the infrastructure for components to interact, developers identify suitable domains and develop new components for them, application assemblers select domain-specific components and assemble them into applications, and end-users employ component-based applications to perform daily tasks [24]. Our approach supports CBSD. The reference toolset provides an infrastructure for components to interact via SIBs, and aids software professionals in creating driver components for selected domains and end-users in assembling these driver components into their smart space applications. Furthermore, our approach makes it possible to assemble the application before the actual domain-specific driver components are implemented. Thus, it is possible to assemble the application of the driver component skeletons and test the assembled application before the actual software code is implemented.

The rest of this section focuses on end-user programming issues. The most important end-user programming methods and the mashup, bottom-up or top-down development methods are discussed and compared to our approach in Sections 7.1–7.4.

7.1. End-user programming methods

The end-user programming methods aim at bridging the gap between use and programming of an application [25,26]. Typically, focus is directed more on reuse of legacy software than on creating new software components or source code. For example, the simplicity, support for immediate feedback, and avoidance of misleading appearances are important in end-user programming tools [27]. The following lists end-user programming approaches taken from [28,25,29]:

1. *Programming-by-example*—Using a particular instance of execution, input–output relations, or existing programs as a basis for creating new programs [28]. For example, modification of a working example speeds up development as it provides stronger scaffolding than writing code from scratch [30].
2. *Visual programming*—Replacing textual programming notation with a graphical one with blocks and connectors [28]. Visual programming concepts and tools assist the user in creating small applications [22].
3. *Script-based creation* makes programming easier and more natural for users who want customized applications and are capable of doing basic programming without having to set up C++ or Java environments, for example [28]. Script languages sacrifice execution efficiency, provide a higher abstraction level for programming than typical system programming languages, weaker typing than system programming languages, and an interpreted development environment [31].
4. *Repository-based creation of applications* supports the reuse of software components. For example, [29] presents an end-user programming approach for Web applications that consist of a (i) Pattern Library, (ii) Pattern Language, and (iii) Command Language. The Pattern library contains patterns (e.g., for dates, times, phone numbers, email addresses, and URLs), parsers (e.g., an HTML parser), and wrappers for websites such as Google or Amazon. The library patterns can be glued together with a pattern language called text constraints, which uses relational operators such as before, after, in, and contains operators to describe a set of regions in a page. The Tcl scripting language is used as a command language. The commands take patterns as arguments to indicate how to manipulate a Web page.
5. *Tailoring of applications* can be based on *customization* that modifies the parameters of components, *integration* that creates or modifies assemblies of components, and *extension* that creates new components by writing program code. The direct activation technique also belongs to this category and requires that the tailoring functionality be accessible from the use context when the need for tailoring occurs [32]. For example, the FreEvolve platform [24] provides an API for integrating tailoring functionality with software components that allow non-programmers to tailor an application by reassembling components at runtime visually [33].

If we compare our approach to the existing end-user programming approaches, it can be seen that it supports visual programming, script-based and repository-based creation, and tailoring of applications. Firstly, the visual programming and scripts are used in the creation of the RDFScript applications. Secondly, the repositories support reuse of command sequences or goal sequences in end-user programming. Thirdly, the approach supports tailoring of applications in the following way: It is easy for the end-user (i) to customize the parameters of commands for the existing applications, (ii) to integrate the capabilities of the driver components to form new applications, and (iii) to extend the existing applications for new purposes. This is done by downloading an RDFScript application from a SIB, modifying/reconfiguring it with the provided visual tools, and finally executing the modified application in the smart environment.

7.2. Mashups development

Integrated Web applications, popularly known as mashups are a coalescence of data sources and APIs into an integrated end-user experience [34]. The next paragraphs list few classifications for the end-user programming methods of mashups. The level of abstraction is used to distinguish how much programming and computer knowledge is required in the end-user development of mashups [35]. Furthermore, since raising the level of abstraction requires encapsulating concepts, the number of choices available on higher levels of abstraction is usually restricted [35,36]. Article [35] divides the end-user development methods of mashups on three levels:

1. *On a high level of abstraction* no programming knowledge is required, but flexibility is usually restricted to reusing and configuring mashups that are developed by others. Working with a high level of abstraction is supported in several ways: reuse of complete mashups created by others, high-level mashup and widget parameterization, automatic reuse of data extractors for websites, and programming-by-example concepts, such as extracting data by example and dragging and dropping feeds on widgets.
2. *On an intermediate level of abstraction* knowledge about concepts such as dataflow, data types or UI widgets is required, but the technological details of these concepts are encapsulated in notations. Therefore, what can be changed or created is limited by these notations. The visual mashup development environments are often based on visual Domain-Specific Languages (DSLs). Visual dataflow languages facilitate the creation of information mashups, whereas the visual workflow/process orchestration languages support the creation of process mashups. For example, Clickscript [37] is a Web mashup editor and a Firefox plug-in that allows visual creation of Web mashups by connecting building blocks of resources (websites) and operations (greater than, if/then, loops, etc.) [22]. The visual languages are used in combination with property dialogs for the parameterization of mashup elements. Data sources are primarily accessed through mashup

elements in those notations. Dialog-based wiring of widgets provides an alternative to visual data-flow languages which saves screen space, but depends on visual widgets. On an intermediate level of abstraction, there is often a distinction between design and run-time mode, although the mashup development environments try to reduce this distinction, e.g., by using previews.

3. *On a low level of abstraction* programming knowledge is required, but the greatest flexibility is achieved. Textual DSL editors for languages such as HTML, JavaScript and custom scripting languages are used to represent mashups or elements of mashups.

The end-user programming paradigms of mashups are also divided into *operation-centric* and *data-centric* paradigms [28]. The *operation-centric* paradigms, namely visual programming and script-based creation, require the user to have a certain understanding of mashups and Web services, and this can mean that a newbie to programming and Web technologies might have difficulties coming up with a working service [28]. The *data-centric* approaches – such as use-time composition approaches, programming by demonstration, and form-based creation – are easier for non-programmers but do not easily extend to the creation of arbitrary, robust, and reusable services. They often lead either to throwaway on-the-fly compositions or to restricted portal-type, parameterized widget collections [28].

In our approach, the end-user programming resembles the mashup development. Thus, the aim is to integrate the informational/operational capabilities of software systems into an integrated end-user experience/application. If we use the classifications of mash-ups to our approach, we can see that it is a data-centric approach on a high level of abstraction. In our approach, the end-user does not need to have a deep understanding of data structures but (s)he must just configure the inputs of commands.

7.3. Bottom-up development methods

Bottom-up development typically focuses on a single software component at a time. For example, the existing bottom-up development methods support design (e.g., a state model for a component's behavior), implementation (e.g., code generator that produces a skeleton for a component), and testing (e.g., unit-testing methods) of software components. The previously discussed component-oriented, service-oriented, and agent-oriented software development methods provide support for bottom-up development. Like our approach, these methods also support creation of reusable parts (e.g., components, Web services, or agents) that can be utilized in various kinds of software systems. The existing bottom-up development methods support creation of professional APIs for software components. Our approach supports the bottom-up development that produces S-APIs for software components that do not just describe the operations of the software components, but also describe execution branches for the operations and thus assist in the creation of execution sequences for the S-APIs' operations.

7.4. Top-down development methods

Top-down development methods focus on the software systems as a whole and these methods support design (e.g., modeling and evaluation of software system architecture), implementation (e.g., frameworks, software product lines, and integration platforms such as SOA), and testing (e.g., integration testing) of software systems. Top-down development methods are typically used in the development of large and complex systems that are possibly composed of a great number of objects at different abstraction levels [38]. We think that the smart space application is also an example of a (possibly) large and complex software system: Firstly, the smart space application is a distributed application that is often based on multiple semantic information brokers, smart objects, and information-level and service-level entities that exist in the smart environment. Secondly, the smart space applications are often based on shared resources and are executed in parallel in a smart environment. The following lists a few top-down development methods for end-users that have similar kinds of features to those provided in our approach.

A model-driven service engineering methodology called ServFace is a top-down development method that assists non-programmers in building applications for Web services [39]. The approach provides a service front end that represents a single operation of a WSDL-described Web service in the form of a generated UI component. The application is created in a visual editor that produces the control-flow and data-flow for the application. Furthermore, the approach developed in the LAPIS project [29] uses a repository mechanism called Pattern Library, which assists end-users in reusing patterns, parsers, and wrappers for the Web application/page development.

Unlike existing top-down development approaches, our approach assists end-user programming that is assisted by the S-APIs. For example, the visual service front-end descriptions used in ServFace do not facilitate creation of control-flow for the application but only provide a visual representation for an API described in WSDL. Similarly, unlike the approach developed in the LAPIS project, our approach uses a repository mechanism in which the repository contains patterns (commands and goals) that assist the combination of patterns in forming execution sequences for the end-user's applications.

8. Conclusions

This article described a semi-automatic end-user programming approach that assists in the creation of easy-to-apply Semantic End-User Application Programming Interfaces (S-APIs) for legacy software components and enables use of S-APIs in command-oriented programming and goal-oriented end-user programming. Furthermore, the described reference toolset enables end-user programming of smart space applications in desktop and mobile environments, facilitates the creation of S-APIs and Java driver components, supports command-oriented and goal-oriented end-user programming of smart space applications, and provides ready-made execution components for cross-smart space applications.

The S-APIs support reuse of goals and capabilities of legacy software systems. Furthermore, unlike the source code of a Java or C++ application, the model of the end-user application specifies the alternative execution branches that relate to the commands of the application. Thus, it is easy for end-users to produce new applications for different purposes from the execution flows of existing applications. Goal-oriented programming provides a new viewpoint for application development. Additionally, a command sequence can be represented as a goal sequence and vice versa. Thus, command-oriented programming and goal-oriented programming are not exclusionary programming methods but can be successively used in application development.

In the future we will focus on improving the usability of the toolset and the performance of the execution environment. Moreover, we will develop testing tools both for driver component testing and end-user application testing. Our aim is to develop run-time extensions to S-APIs, a more compact format for the RDFScripts, and direct messaging channels between Application Executors and Driver KPs to increase the execution speed of end-user applications. Quality information such as number of errors, execution times, and execution delays can be produced and embedded in S-APIs at run-time and utilized in the application execution too. This quality information can be used, for example, in selecting capabilities that provide either the best possible performance or those that provide secure and reliable processing capabilities with lower performance.

Acknowledgments

Publication of this article has been supported by the ODEP project funded by the National Technology Agency (Tekes) and by the SOFIA/Artemis projects (http://www.sofia-project.eu/) funded by the National Technology Agency (Tekes), VTT Technical Research Centre of Finland, and the EU.

References

[1] R.T. Fielding, Architectural styles and the design of network-based software architectures, Ph.D. Thesis. University of California, Irvine, 2000.
[2] J.F. Pane, C.A. Ratanamahatana, B.A. Myers, Studying the language and structure in non-programmers' solutions to programming problems, International Journal of Human–Computer Studies 54 (2001) 237–264.
[3] Z. Nan, M.B. Rosson, Playing with information: how end users think about and integrate dynamic data, in: Presented at IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC, 2009.
[4] M. Palviainen, J. Kuusijärvi, E. Ovaska, Architecture for end-user programming of cross-smart space applications, in: Presented at the 4rd International Workshop on Sensor Networks and Ambient Intelligence, SeNAmI 2012, Lugano, Switzerland, 2012.
[5] M. Palviainen, J. Kuusijärvi, E. Ovaska, Framework for end-user programming of cross-smart space applications, Sensors 12 (2012) 14442–14466.
[6] A. Katasonov, Enabling non-programmers to develop smart environment applications, in: Presented at IEEE Symposium on Computers and Communications, ISCC, 2010.
[7] E. Ovaska, T. Salmon Cinotti, A. Toninelli, Design principles and practices of interoperable smart spaces, in: Xiaodong Liu, Yang Li (Eds.), In the book: Advanced Design Approaches to Emerging Software Systems: Principles, Methodology and Tools, 2012, pp. 18–47.
[8] A. Lappeteläinen, J.M. Tuupola, A. Palin, T. Eriksson, Networked systems, services and information, in: Presented at the 1st International Network on Terminal Architecture Conference, NoTA2008, Helsinki, Finland, 2008.
[9] J. Honkola, H. Laine, R. Brown, O. Tyrkko, Smart-M3 information sharing platform, in: Presented at IEEE Symposium on Computers and Communications, ISCC, Riccione, Italy, 2010.
[10] A. Toninelli, S. Pantsar-Syväniemi, P. Bellavista, E. Ovaska, Supporting context awareness in smart environments: a scalable approach to information interoperability, in: Proceedings of the International Workshop on Middleware for Pervasive Mobile and Embedded Computing, ACM, Urbana Champaign, Illinois, 2009.
[11] W3C, RDF Vocabulary Description Language 1.0: RDF Schema, in: D. Brickley, R.V. Guha (Eds.), W3C Recommendation, 2004. Online: http://www.w3.org/TR/rdf-schema/.
[12] M. Palviainen, A. Katasonov, Model and ontology-based development of smart space applications, in: A. Malatras (Ed.), Pervasive Computing and Communications Design and Deployment: Technologies, Trends, and Applications, IGI Global, 2011.
[13] S. Pantsar-Syvaniemi, K. Simula, E. Ovaska, Context-awareness in smart spaces, in: Presented at the IEEE Symposium on Computers and Communications, ISCC, 2010.
[14] J. Mazzola Paluska, H. Pham, U. Saif, G. Chau, C. Terman, S. Ward, Structured decomposition of adaptive applications, Pervasive and Mobile Computing 4 (2008) 791–806.
[15] A. Katasonov, M. Palviainen, Towards ontology-driven development of applications for smart environments, in: Presented at the 1st International Workshop on the Web of Things (WoT 2010) at PerCom Conference, IEEE, Mannheim, Germany, 2010.
[16] E. Ovaska, Ontology driven piecemeal development of smart spaces, in: The 1st International Joint Conference on Ambient Intelligence, AmI 2010, Malaga, Spain, in: Lecture Notes in Computer Science, vol. 6439, Springer, 2010, pp. 148–156.
[17] G. Booch, R. Maksimchuk, M. Engle, B. Young, J. Conallen, K. Houston, Object-Oriented Analysis and Design with Applications, third ed., Addison-Wesley Professional, 2007.
[18] F. Garijo, M. Boman, N. Jennings, Agent-oriented software engineering, in: Multi-Agent System Engineering, in: Lecture Notes in Computer Science, vol. 1647, Springer, Berlin, Heidelberg, 1999, pp. 1–7.
[19] J. Bosch, Architecture in the age of compositionality, in: Lecture Notes in Computer Science, vol. 6285, 2010, pp. 1–4.
[20] A. Katasonov, V. Terziyan, Semantic agent programming language (S-APL): a middleware platform for the semantic web, in: Presented at the IEEE International Conference on Semantic Computing, 2008.
[21] A. Brown, S. Johnston, K. Kelly, Using service-oriented architecture and component-based development to build web service applications, Rational-IBM White Paper, 2003.
[22] M. Kovatsch, M. Weiss, D. Guinard, Embedding Internet technology for home automation, in: Presented at the IEEE Conference on Emerging Technologies and Factory Automation, ETFA, 2010.
[23] D. Guinard, V. Trifa, E. Wilde, Architecting a mashable open world wide web of things, Technical Report 663. ETH Zurich, 2010.
[24] O. Stiemerling, Component-based tailorability, Ph.D. Thesis, University of Bonn, 2000.

[25] A.I. Mørch, G. Stevens, M. Won, M. Klann, Y. Dittrich, V. Wulf, Component-based technologies for end-user development, Communications of the ACM 47 (2004) 59–62.
[26] A.I. Mørch, Tailoring tools for system development, Journal of End User Computing 10 (1998) 22–29.
[27] J.F. Pane, B.A. Myers, Usability issues in the design of novice programming systems, CMU-HCII-96-101, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1996.
[28] S. Kotkaluoto, J. Leino, A. Oulasvirta, P. Peltonen, K.J. Räihä, S. Törmä, Review of service composition interfaces, Department of Computer Sciences, University of Tampere, 2009.
[29] R.C. Miller, End user programming for web users, in: Presented at the End User Development Workshop at CHI Conference, Ft. Lauderdale, Florida, USA, 2003.
[30] B. Hartmann, L. Wu, K. Collins, S.R. Klemmer, Programming by a sample: rapidly creating web applications with d.mix, in: Presented at the 20th Annual ACM Symposium on User Interface Software and Technology, Newport, Rhode Island, USA, 2007.
[31] J.K. Ousterhout, Scripting: higher-level programming for the 21st century, IEEE Computer 31 (1998) 23–30.
[32] V. Wulf, B. Golombek, Direct activation: a concept to encourage tailoring activities, Behaviour and Information Technology 20 (2001) 249–263.
[33] M. Won, O. Stiemerling, V. Wulf, Component-based approaches to tailorable systems, in: H. Lieberman, F. Paternò, V. Wulf (Eds.), End User Development, in: Human–Computer Interaction Series, vol. 9, Springer, Netherlands, 2006, pp. 115–141.
[34] N. Zang, M.B. Rosson, V. Nasser, Mashups: who? What? Why? in: Presented at Conference on Human Factors in Computing Systems, CHI'08, Florence, Italy, 2008.
[35] L. Grammel, M.-A. Storey, A survey of mashup development environments, in: M. Chignell, J. Cordy, J. Ng, Y. Yesha (Eds.), The Smart Internet, in: Lecture Notes in Computer Science, vol. 6400, Springer, Berlin, Heidelberg, 2010, pp. 137–151.
[36] L. Grammel, M.A. Storey, An end user perspective on mashup makers, Technical Report DCS-324-IR: University of Victoria, 2008.
[37] http://www.clickscript.ch/.
[38] A. Achilleos, K. Yang, N. Georgalas, Context modelling and a context-aware framework for pervasive service creation: a model-driven approach, Pervasive and Mobile Computing 6 (2009) 281–296.
[39] A. Namoun, T. Nestler, A. De Angeli, Service composition for non-programmers: prospects, problems, and design recommendations, in: Presented at the 8th IEEE European Conference on Web Services, ECOWS, 2010.