Establishing requirements for End-user Service Composition tools

Andy Ridge · Eamonn O'Neill

Received: 8 October 2013 / Accepted: 10 June 2014 / Published online: 1 July 2014
© The Author(s) 2014. This article is published with open access at Springerlink.com

Abstract End-user Service Composition (EUSC) is defined as service composition where the person who creates the composite service also uses it. We derived requirements for EUSC tools using a custom method based on the scenario-based requirements analysis method. Ten participants were given scenarios, a demonstrator application and examples of EUSC tools before being interviewed. Analysing their responses, we elicited 139 requirements across topics including service “types” (e.g. pervasive services), composition flow (i.e. control/data flow) and the use of templates in composition. This is the first published work to have used a robust requirements analysis method in EUSC, adapted specifically to take account of prior work on EUSC. Some of our requirements validate those found in prior research or current EUSC tools but many are unique to our findings. The requirements found in this work inform and inspire the development of future EUSC tools, and the methodology is easily adapted for application in other domains.

Keywords Service Composition · End-user Service Composition · Scenarios · SCRAM

1 Introduction

Service Composition (SC), particularly when aimed at end-users, is a relatively young area, and this is reflected in the small amount of work that has been done on deriving requirements for tools to support end-users in performing

SC. End-user Service Composition (EUSC) is defined as SC where the user who composes a composite service is the same person who ultimately uses it. In most cases, we may assume that such a user has a relatively low level of technical knowledge compared with, for example, a professional service developer.

This work focuses on SC by end-users, with no explicit restriction on the technologies that underpin the overall composition process. Traditional SC approaches tend to be driven by the technology underpinning the composition or the developer, rather than by the end-user of the composition tool or composite service [1]. We argue that EUSC and the technologies that underpin it can benefit from active investigation of what end-users require from composition.

Prior work has derived a limited number of requirements for EUSC by using focus groups to gather user opinions and perceptions of various elements of SC on the Web [2, 3]. The requirements derived across this prior work cover the area sporadically, providing few general requirements and few requirements for specific areas of EUSC.

The field would benefit from an approach to deriving a comprehensive set of requirements for EUSC tools. While prior work has focused on Web-based SC, which is arguably one of the more compelling areas in the field, the range of available EUSC tools is now broader than this, notably on mobile platforms. Hence our work targets EUSC tools in general, and the wide range of services available across diverse domains, and is not limited to the Web.

The process and requirements presented here focus on requirements gathering from end-users—so-called ordinary people—rather than business customers in a B2B relationship. The process of SC is an inherently technical one, and potential users are likely to need exposure to concepts such as apps, Web 2.0 services, etc., in order to properly understand it. Furthermore, it is unlikely that a user who has not

---

A. Ridge (✉) · E. O'Neill
Department of Computer Science, University of Bath,
Bath BA2 7AY, UK
e-mail: A.D.Ridge@bath.ac.uk

previously been exposed to such mobile applications (apps) and services would seek to use SC to solve the problems that she might have. We therefore restricted the target end-users in this work to owners of a smartphone (any platform) who were familiar with these concepts and who had experience of using apps and services on these devices.

The main contributions of this paper are (1) to describe a generalizable and repeatable process for deriving a robust and coherent set of requirements for EUSC tools; and (2) to present the comprehensive set of 139 requirements that we gathered. We demonstrate that our methodology is generic by outlining how it could be applied in different domains in addition to our chosen domain of EUSC. Our methodology also significantly extends and adapts the one on which it is based: we show how the data from the sessions is analysed and requirements elicited, and we adapt the methodology from targeting business customers to targeting end-users.

Our requirements cover a broad range of topics—relating both to composition in general and to a number of specific problems in composition. We structure the functional requirements based on an adapted model of the life cycle for EUSC, and the non-functional requirements are categorized against the quality standards presented in ISO/IEC 25010 [4]. Some of the requirements found in our work validate those found in prior work while others are unique to our findings. Thus, our findings can inform the design of future EUSC tools.

The work reported here is aimed at deriving requirements for EUSC tools, i.e., the tools that end-users use in the process of performing SC. To clarify, it is *not* aimed at deriving requirements for the composite services produced through that process. Another clarification required is in terminology. There are various definitions for the entities involved in EUSC but no current standardization, so we now provide definitions of terms that will be used in the rest of this paper. We define “components” or “component services” as the input services to the composition process; “composites” or “composite services” as the output composed services, and both components and composites generically as “services”.

The rest of this paper is structured as follows. Relevant background is discussed in the next section. In Sect. 3, we present the method that was used in our study. Section 4 presents the results, and Sect. 5 presents the analysis process and derivation of requirements. Section 6 presents a small illustrative subset of the requirements as well as an evaluation of the set as a whole. Finally, we present limitations, future work, and our conclusions.

2 Background

In this section, we present related work from EUSC, including other EUSC requirements gathering approaches and a brief overview of the method used in SCRAM.

2.1 End-user Service Composition

SC is defined as the process of creating services at runtime from a series of component services [5].¹ SC has been used widely within business but has only relatively recently been aimed at consumers as end-users. This section will discuss the SC life cycle, the different layers at which SC can operate, and available EUSC tools.

There are two main views on the life cycle for EUSC, the first of which splits it into four stages [6]. These are as follows:

1. Service request—the user requests the composition that she wants to create [6] (N.B. This is relevant only in automated SC);
2. Service discovery—the user or the system (or both) discovers the component services to be used in the composition process [5, 6];
3. Composition—the user or the system (or both) combines the services, coordinating them in order to create a new, composite service [5, 6];
4. Service execution—the output of the composition stage is executed [6]. However, to be truly dynamic, this life cycle needs to make some provision for adapting the composite service once it has been created.

The other view on this life cycle identifies 6 stages in service-oriented development, which are further broken down into 16 activities [7]. These stages and activities are outlined in Table 1.

Drawing on these views of the EUSC life cycle, for the purposes of this paper, we consider the following main stages of the EUSC life cycle:

1. *Request and Discovery*—The user makes a request for the service(s) that they want and discovers components or composites that meet her needs.
2. *Composition*—The discovered components are coordinated together to form a composite service.
3. *Verification and Validation*—The user verifies that the composite service executes correctly and meets her needs.
4. *Annotation and Deployment*—The user annotates the service with relevant information and optionally deploys it to some service repository to be discovered by other users.
5. *Execution and Management*—The user executes the service and can subsequently adapt it to meet changes in specification or the execution environment.

¹ A number of definitions exist but they are all variations on this theme.

Table 1 Stages of the EUSC life cycle [7]

Stage	Activity	Activity description
Inception	Feasibility	Estimating the feasibility of the project
	Project planning	Planning of the overall project
Analysis	Domain	Analysis of the problem domain
	Activity	Analysis of activities that the system needs to support
Specification	Needs	Identifying the needs that must be satisfied
	Requirements	Identifying the requirements that the service must adhere to
	QoS	Identifying the Quality of Service requirements for the service
Realization	Discover	Discover existing services that might meet the specified requirements
	Design	Design the required service
	Construct	Construct the required service
	Verify	Verify that the service works as designed
	Validate	Validate that the service meets the needs in the specification
Provisioning	Annotation	Annotate the service so that it can be discovered in its own right
	Deployment	Deploy the service to ensure it is accessible
Management	Monitoring	Monitor the operation of the service to ensure it continues to meet specified needs
	Adaptation	Adapt the service to react to changes in the specification or the environment

There are three layers at which SC can operate: the application layer, the service layer, and the presentation layer [8]. Our work is not specifically aimed at any one of these layers. Furthermore, the differences between application- and service-layer composition are often imperceptible to the user of the tool.

A standard “benchmark” against which to compare SC tools is Yahoo! Pipes [7],² which has been available for several years. More recently, Web-based SC tools such as “if this then that”—IFTTT³ and Zapier⁴ have been released. SC tools in other domains include mobile (Android-based) applications such as Tasker,⁵ and hybrid Web and mobile approaches such as Microsoft’s On{X}⁶. There are also examples of desktop SC applications

available for Mac OS X: automator—a workflow automation tool, and Quartz Composer—a multimedia composition tool.

2.2 EUSC requirements

In this section, we describe requirements that have already been gathered for EUSC tools, or simply used to specify them. We split this prior work into two groups: (1) those which explicitly sought to gather requirements for EUSC tools, and (2) those where the requirements have simply been stated before the creation of an EUSC tool.

2.2.1 EUSC requirements gathering

To our knowledge, only one other research group has sought to gather requirements from end-users for EUSC, in that case in the area of Web-based EUSC [2, 3].

To gather requirements for EUSC tools on the Web, Namoun et al. [3] carried out a focus group study to identify users’ perceptions of services and SC in order to design a future EUSC tool. Their focus group sessions were made up of six steps. First, participants were asked to define common SC terms, before being provided with definitions of these terms. They were shown an initial mock-up of the design of a SC tool and asked to comment on its design. Next, participants were guided through an example composition from a script. Participants were then invited to provide their views on SC and evaluate various other design mock-ups for the tool. The penultimate stage involved the researcher demonstrating the process of a SC using a prototypical design of the SC tool whose design was presented earlier. Finally, participants were asked to give their views on SC and the approach taken in the design of the prototypical tool.

Following analysis of the responses given by participants across each of these sections, Namoun et al. [3] presented the following set of requirements:

- R1. Display services by their user interface
- R2. Use a semi-automatic approach to composition
- R3. Avoid technical jargon
- R4. The composition “canvas” should be large and interaction easy
- R5. Services should be secure
- R6. Feedback to users should be continuous and proactive

Other work in deriving requirements for SC followed a similar process. Mehandjiev et al. [2] also used focus groups to assess potential users’ opinions of SC and the different representations of the flow of composition—control flow and data flow.

Their sessions included five stages. As with [3], the session began with participants being asked to define terms

² <http://pipes.yahoo.com>.

³ <http://ifttt.com>.

⁴ <http://www.zapier.com>.

⁵ <http://tasker.dinglich.net>.

⁶ <http://onx.ms>.

relating to SC, in this case *service* and *software service*. They were then given a 20-min introduction of SC concepts, followed by being asked to complete a form to assess their technical knowledge and prior experience with software services. Next, participants were asked to perform three tasks to assess how they would compose “atomic” services into composite services. A final presentation and associated questionnaire were then given to assess users’ opinions on the design alternatives for composition—control flow, data flow, or their assisted composition approach. Participants’ responses to an exit questionnaire yielded the following requirements:

- R7. The data being passed between services should not be presented
- R8. Sets of data should be treated as a single item
- R9. Users should be assisted in solving control flow dependencies
- R10. Users should be assisted in solving data flow problems
- R11. The UI of the composed service should be represented

The requirements presented in [3] (R1–R6) are a small set of general requirements for EUSC on the Web, whereas those from [2] (R7–R11) are more focused on a single aspect of composition: the flow between components. The overall set of requirements is deficient in two ways: (1) there are few requirements that are general, applying to EUSC as a whole, and (2) only one-specific area of the EUSC process has been investigated. Thus, we feel that it is necessary to gather a more comprehensive set of requirements to address both of these deficiencies.

2.2.2 Other EUSC requirements

Several other authors have listed requirements for EUSC in work where gathering such requirements was not an explicit aim. Instead, requirements are often simply stated before they create an EUSC tool, without any discussion of the source of the requirements. The first of these was an investigation of EUSC at the presentation layer, i.e., composition of service interfaces by Nestler et al. [9], in which they list 5 general requirements for their tool, the ServFace Builder. Mehandjiev et al. [10] present a user-first method for composition. After presenting their composition approach, they provide 6 recommendations gathered from focus groups that were presented with their approach and asked to evaluate it. Albinola et al. [11] detail the creation and structure of a mashup framework called Mashlight, and list requirements that Mashlight should adhere to throughout the work. Finally, Bottaro et al. introduce an architecture for composition of pervasive services in the home, prior to which they identify 5 requirements [12].

A summary of the requirements found in these works is discussed in the preliminary requirements capture section, and requirements are presented in “Appendix 1”. The above sets of requirements (both gathered and stated) give a sporadic coverage of the EUSC domain, which motivates a more formal approach to gathering a comprehensive set of requirements. Our work sets out to derive this comprehensive set of requirements that cover a range of design areas within EUSC and associated tools. We derived these requirements using a method that was influenced by the scenario-based requirements analysis method (SCRAM).

2.3 Requirements gathering techniques

Most requirements gathering techniques focus on a business customer rather than being targeted at end-users. Our goal is to gather requirements from a group of potential end-users, so any such technique is likely to require some modification.

2.3.1 Interviews

Interviews with stakeholders are often used as part of the requirements gathering process [13]. Interviews normally involve a stakeholder who already uses a similar system (in the B2B context) and hence has the “right” and “wrong” answers as to what can be classed as a requirement for a system [13]. These interviews are either closed—questions to be posed to stakeholders are defined beforehand—or open—where the agenda of the interview is not defined beforehand—or, more commonly, a blend of the two [13].

Normally, interviews would also be used by requirements engineers to gather more information about the domain in which the stakeholder operates [13]. In this case, however, the requirements engineer is the domain expert (i.e. expert in EUSC) rather than the interviewee, meaning that some methodological adaptation is needed. This role reversal mitigates issues that can arise from stakeholders using terminology unfamiliar to the requirements engineer or inadequately describing the requirements due to their greater knowledge in the domain. Furthermore, these interviews would normally take place within a business context and have to take into account political and organizational issues—another concern that we do not have to address.

2.3.2 Scenarios

Scenarios are normally used in requirements gathering as a means of identifying tasks to be completed using the tool that is the output of the system development process as a whole [13]. In our case, the stakeholders in our requirements gathering process are potential end-users of EUSC

tools who may not have any knowledge of EUSC prior to the interview session. Therefore, in addition to the use of scenarios to help identify tasks, they are a means of conveying this knowledge to the stakeholder.

Potts' Inquiry Cycle [14] is an example of a scenario-based requirements analysis method that utilizes scenarios as a mechanism for identifying problems when performing requirements analysis [14, 15]. However, they provide little detail on how these problems are identified or how requirements are extracted [14, 15].

2.3.3 Scenario-based requirements analysis method (SCRAM)

SCRAM is a requirements analysis method that utilizes introductory scenarios, a concept demonstrator application, and examples of other potential designs to present a concept to potential users of a system and gather requirements for it. SCRAM consists of 4 stages [16]:

1. *Preliminary requirements capture and domain familiarization*: this is preliminary research that is required to gather requirements and design rationale to facilitate creating a prototype.
2. *Storyboarding and design visioning*: a prototype of the required system is designed and created. Scripts are also created to outline the process that the demonstrator would undertake if it were a fully fledged application.
3. *Requirements exploration*: users are presented with the concept demonstrator (along with other designs) and scenarios to demonstrate the problem area. Probe questions are asked at key points in the demonstration script, and design decisions are illustrated with design rationale documents.
4. *Session analysis*: the data are analysed to derive requirements that can be reported back to the user.

3 Methodology

This section gives an overview of the method that was applied in our requirements gathering sessions. We also describe the tasks that were undertaken before and after the sessions, before any analysis was carried out.

Our requirements gathering method is influenced by SCRAM, outlined briefly above. However, published works that describe SCRAM provide guidance only for the earlier stages of the requirements gathering process, giving little detailed guidance for the latter stages [16, 17]. We used SCRAM as the basis of our method because we felt that it would be a relatively simple process to adapt to focus on end-users rather than business customers. In

particular, we felt that the prototypical demonstrator would be a very effective way of presenting EUSC to our participants. Scenarios to which participants can relate are also of particular value in developing shared understandings, given that most end-users of EUSC tools are likely to be unfamiliar with EUSC.

We divide the method into three sections:

1. Pre-study: the activities performed before the sessions were carried out.
2. Study: the method carried out within the study sessions themselves.
3. Post-study: the process of transforming the data gathered in the sessions into a set of requirements.

3.1 Pre-study method

Our pre-session method was based on SCRAM since this part of the method sets up the prototype demonstrator to be used in the study sessions.

3.1.1 Preliminary requirements capture

Prior to the creation of the prototypical demonstrator application, an initial set of requirements must first be gathered to which the demonstrator must adhere. Sutcliffe and Ryan present this as the first stage of SCRAM, but little instruction is provided as to how requirements should be gathered [16, 17]. The instruction provided assumes that the requirements are being gathered in a business context rather than from end-users, necessitating a different approach in our case. We collated requirements from the work described in the Background section, yielding a short list of 19 requirements, which can be found in "Appendix 1". The topics covered in the preliminary requirements range from general EUSC [6], to EUSC on the Web [9], mashups [11, 18], to pervasive EUSC [12].

Following the preliminary collation of requirements, we performed reviews of the EUSC literature and available tools to identify the main functions of EUSC tools that had not been identified in this initial set of requirements. The main outcome of these reviews was the identification of functions that the prototype must perform that were missing from the preliminary requirements capture. The functions identified were based on aspects of the EUSC life cycle [6, 7]:

1. *Specification and Discovery*—The user specifies what they want and discovers components that they could use in composition to meet their specification.
2. *Composition*—The user coordinates the components discovered in previous step to create a composite service.

3. *Verification and Validation*—The user verifies that the composite they have created meets with their initial specification.
4. *Annotation and Deployment*—The user records information about the composite they have created, as well as potentially being able to share the composite with other users.
5. *Execution*—The user executes the composite service that they have created.

We then sought to group together the requirements in a manner in which they can be used to motivate the design of the prototype. Geyer suggests that a design space model is a useful method for presenting requirements for a system [19], but since we have already collected the requirements, we require only the design space structure. One suggested structure for a design space model groups requirements or design decisions into three groups: functional, non-functional, and structural [20].

Functional requirements identified the functions that the prototype needed to perform, which were based on the stages identified in the composition process from the prior literature and a tool review. *Non-functional requirements* mostly related to the representation of and possible interactions with the composition process, for example semi-automation [3], the use of metaphor [9] or abstraction, e.g. hiding technical details such as code [9, 21]. Other requirements included avoiding complex terminology and technical jargon, as well as providing proactive user feedback [21]. None of the preliminary requirements identified fit into the set of *structural requirements*. A complete list of the preliminary requirements can be found in “Appendix 1”.

3.1.2 Prototype specification and implementation

The second stage presented in SCRAM is the specification and development of the prototypical demonstrator, treated as a “script” with limited functionality and interactivity [17]. Sutcliffe states that better quality feedback is received from users when presenting an interactive prototype [22]. Furthermore, we felt that for SC it is important for participants to be able to experience the process of composing services and using the output. Hence, we decided that our prototype should, at the very minimum, provide the ability to compose services and execute the composite. After gathering the requirements for the prototype, these requirements needed to be translated into the design for a prototype. The functional requirements suggested three main areas in which the prototype must provide functionality:

1. Viewing, discovering and interacting with components.
2. Creating composites by using components in composition.

3. Viewing, discovering and interacting with composites (including iteration).

Within each of the main sections of the tool, various design decisions were made based on requirements derived earlier, as well as being influenced by the designs of currently available EUSC tools.

For example, one requirement recommended the use of templates in composition to simplify the process [10]. In a review of available tools, we saw that examples of templates could restrict composition to “*if [Trigger] then [Action]*” (e.g. IFTTT or Zapier), which is a particularly restrictive template. Another option (which we chose to follow) was to restrict composition to being linear, meaning that composition would allow “*[Component 1] then [Component 2] then ...*” (e.g. Automator or Tasker), which provides more freedom to the user. Other design decisions made at this stage were recorded and later presented to participants as part of the SCRAM sessions.

An important initial design decision was the platform on which the prototype should operate; in our case, the scenario prompted a mobile platform, and Android was selected. We named the Android app Composer, and a generic icon was selected from the provided images in the Android SDK.

To ensure that the prototype was presented at the same level as the other tools in the session, it was important to present an application that was as fully featured as possible and that did not necessarily look like a prototype.

Figure 1 shows the composition process in Composer for the composite in the main scenario for the sessions—a composite that notifies the user about delays on the tube. (The relevant scenario is presented later.) Figure 2 shows the list of components that are available to be composed in the prototype.

3.1.3 Scenario

The main scenario that participants were presented with as part of the introductory materials is shown below:

“Ben has a London Underground tube line status service for his smartphone that allows him to check the status of any tube line. He feels that it is too much hassle to check each of these manually every time he needs to get the tube and wants his phone to notify him when there is a problem using the in-built notification service in the OS. There is no option in the service itself to do this, so he decides to use EUSC to fix his problem. Using the Composer tool, he is able to compose the phone’s notification service onto the tube service, so that when a problem is reported on a particular line (which he can choose), he will be notified via a new item in the notification tray.

After using this service for a few days and being notified at strange times of day, he decides that he wants to receive

Fig. 1 The composition process in Composer

Fig. 2 The list of components provided by Composer

these notifications around the times he would normally be getting the tube. He chooses to edit the service and adds the device's clock service in between the tube service and the notification service. He sets the clock to only let the notifications through between 6–8 a.m., and 4–7 p.m.”

This scenario was created in an earlier technical meeting with a number of experts in services and SC.

3.2 Study sessions

We made a number of changes to the session method suggested in SCRAM, based on adaptations that we required because of the different context in which we were carrying out our study and drawing on comments made by Sutcliffe et al. [16, 17], regarding problems with the method in its original form.

Our sessions were split into three main parts, shown in Fig. 3. First, participants were presented with a set of introductory materials to describe SC, a glossary of terms, and a set of five introductory scenarios (as SCRAM). In the second stage, participants watched a composition task being performed with the demonstrator application following a scripted sequence, followed by a demonstration of a similar task on other EUSC tools, and were interviewed about the relevant section of the tool while having design decisions explained to them. This process was then repeated for different general functions within the tools. Finally, participants were invited to make any further comments on SC, the concepts involved or any of the EUSC tools that they encountered during the session.

The main part of the session was the run-through of the script with the prototypical demonstrator and other examples of EUSC tools, and subsequent interview questions (Task demonstration and interview in Fig. 3). The other EUSC tools were selected to present a diverse range of

Fig. 3 The sub-stages of our study sessions

design options to participants across two domains: mobile, Web, and desktop. The tools selected were as follows: Tasker (mobile), On{X} (web/mobile hybrid), IFTTT (Web), Yahoo! Pipes (Web mashup), Automator (desktop) and Quartz Composer (desktop). This part of the session was split further into three sections, which were chosen based on the three activities identified in the prototype specification: interacting with components, composition, and interaction with composites. For each of these activities, we performed the following tasks:

1. Demonstrate the use of the corresponding section of the tool in the Composer prototype. This demonstration followed a scripted sequence, and the demonstrated task was the same across all participants.
2. Perform a similar scripted task in the same section of each of the other EUSC tools selected: Tasker, On{X}, IFTTT, Yahoo! Pipes, Automator, and Quartz composer. Note that we were unable to perform precisely the same task in each tool due to the diverse nature of the chosen EUSC tools. However, the tasks were the same across all participants.
3. Interview the participant on the section of the tool using the probe questions and prompts derived from our initial categories. Questions were left open-ended but were followed up with probes if participants seemed unsure how to respond.

Each session lasted approximately 90 min and was video recorded.

Our method deviates from SCRAM in this part as we presented alternative fully fledged EUSC tools to participants rather than storyboard sketches of other designs. The aim of this change was to remove the bias towards the prototypical demonstrator over the other designs, which has been identified as a weakness of the original method [16, 17]. We feel that our change is beneficial as it presents the other designs on an equal footing with the prototypical demonstrator—especially given that our prototype is more fully featured than the prototype suggested in the original method. Furthermore, the original work on SCRAM presented design alternatives to participants using questions,

options, and criteria (QOC). This was found to be ineffective as users did not understand the representation, and Sutcliffe and Ryan [17] recommended using tables instead. We incorporated this design decision discussion with presenting the tools and were able to demonstrate the different design decisions made in each tool while it was being used.

To assess the effectiveness of our methodological changes, we carried out three pilot studies. When performing these pilot sessions, we found that they were running for upwards of 2 h, which became fatiguing for the participants and consequently detrimental to the effectiveness of the discussions. Sutcliffe and Ryan [17] indicate that session overloading is a problem that can occur with SCRAM and should be minimized wherever possible. Since we could not remove any of the earlier sections without compromising participants' knowledge, and hence, the requirements we would be able to gather, we instead moved the explicit requirements elicitation process from the sessions themselves and performed it post hoc following data analysis. Our discussions with participants of their answers to the probe questions still provided enough detail from which to generate a comprehensive set of requirements.

Within the sessions, we decided to have only a single requirements engineer. A pilot session with multiple requirements engineers showed that given the lightweight nature of compositions using the EUSC tools we were demonstrating, the second requirements engineer was redundant for most of the session. This also meant that we could reduce the number of participants per session, since removing the second requirements engineer meant that the session could remain balanced with one participant, who could still maintain ownership of the session [17]. Across three pilot sessions, we found that multiple participants led to fewer topics from which requirements could be derived compared with having a single participant discussing topics with the requirements engineer.

3.3 Post-study method

This part of our method is not related to SCRAM, since in the original work, Sutcliffe and Ryan's requirements were

elicited in the session whereas ours are not, and they provide little guidance as to how this is achieved in those sessions [16, 17]. The output of our study sessions is a transcript of the discussion that the participant had with the requirements engineer, which needs to be analysed to elicit requirements. Our approach was first to codify the transcriptions in order to identify the topics of the discussions and then further to elicit requirements. The approach we used for this analysis and elicitation is directed content analysis (DCA). DCA uses a more structured approach than other content analysis approaches [23]. This is achieved by identifying aspects from prior research in the area as initial coding categories [23]. DCA involves going through the transcript and assigning codes to topics/concepts relating to the initial categories. Any topics that do not relate to the initial categories are put to one side to be analysed later. These latter topics are then coded and grouped together into a set of subsequent categories. It is useful in an interview situation where participants are asked a series of questions that follow the topics of the predetermined categories, since they are primed to respond about those topics, and DCA is robust to priming. Data coded in this manner should not be compared using statistical tests of difference, so ranking and frequency comparisons may be used instead [24]. However, when performing these ranking and frequency analyses, it is important to take into account the bias/priming incurred by the initial categories compared with those of the codes and categories derived from the data.

In contrast, conventional content analysis is used when there is relatively little existing theory or research in a particular area, where codes and categories are derived directly from the data itself [23]. This approach relies on open-ended questions with high-level probes [23], not suitable for use with SCRAM.

We identified seven initial categories for DCA from domain research. Composition flow [2] is the type of flow in the composition—control, data, etc. (reported in Sect. 2.2). Composition—connections and compatibility [25] are the connections between the components in the composition process. Metaphor [26] is the metaphor used to abstract the representation of the composition. Templates/examples [2, 10] refer to the use of templates/examples to assist the user. Component type is the “type” of components that are supported, e.g. triggers in IFTTT. Discovery/acquisition of components refers to how users discover and acquire new components [6]. Attributes are the attributes that the component or composite presents to the user [10].

There were obvious omissions from the set of categories above with respect to SC, so we reviewed the design of currently available SC tools to gather others. Components and composites are general references to components and composites, respectively. Inputs/outputs are the display and

use of inputs and outputs of components (if data passing is supported). Testing is the ability of the user to test a composition as she is making it. Grouping refers to how users might want to group collections of components or composites together. Aesthetic refers to how visual the SC tool is. The categories identified in the preliminary work for DCA were used as the topics for the probe questions in the sessions.

4 Results

This section describes the stage at which the data from the sessions were transcribed and broken down into codes that could subsequently be analysed in order to derive requirements.

We carried out 10 sessions with 1 participant per session: 5 male, 5 female, with a mean age of 27.8 (SD = 10.18). Five participants were students: 4 were employed and 1 was self-employed; 3 had a background in Computer Science, 2 in Physics, 2 in Beauty, 1 in Engineering, 1 in Psychology, and 1 in Geography. All were owners of smartphones: 5 iPhone, 4 Android, and 1 Blackberry. Transcriptions of the sessions were coded using directed content analysis based on the initial categories listed in Sect. 3.1. Each category had a number of codes associated with it, and each code had a total number of occurrences across all participants: **O**; and a number of participants who used the code (out of 10): **P**. For example, within the category “Attributes – components”, three of the identified codes were as follows: [*Description*, 24**O**, 5**P**], [*Number of uses*, 8**O**, 5**P**], and [*Cost (free)*, 8**O**, 5**P**].

According to the directed-coding process, codes that did not fit into initial categories were categorized in a bottom-up manner, yielding a set of 6 subsequent categories. *Tool feature* referred to general features of the tool. *Social* indicated any connections with social media/friends. *Assistance* reflected any extra assistance that the tool provides to the user. *Specific tool/function* indicated that there was a reference to a specific tool or feature that a tool has. *Accessibility* referred to accessibility features provided by the tool. *Comparison with non-SC tool* indicated that the participant compared an aspect of composition with something outside the domain of SC.

Given a set of codes and occurrences derived using directed content analysis, we are able to use quantitative comparisons to apply a rank ordering to the codes that were identified [23]. When considering the two types of category separately, this gives us a rough idea of how popular the different codes within the categories were. However, this is a very simplistic approach and as such cannot be used to make any grand statements about the popular features within particular areas.

For instance, Table 2 shows the five most popular codes in the initial categories, and Table 3 shows the five most popular codes in the subsequent categories. In this case, popularity was judged based first on the number of different participants with whom that code was identified, followed by the total number of occurrences across all participants. Occurrences of codes were only identified if they were explicit references to the code made by the participant; just because a participant was asked about a topic did not mean that a code was recorded at that point.

While this method of ranking codes by their frequency of use shows which codes were popular across participants (with or without priming/bias), it does not reflect all of the interesting findings of the study, given that the most interesting topics were usually identified by a single participant. These “interesting” codes were identified across both sets of categories—both initial and subsequent. A selection of the interesting codes is listed below:

- *OS integration*—integration of SC with the home screen of the phone instead of an app.
- *Composing pervasive services*—services in the environment can be discovered.
- *Multiple icons to show components/function*—the composite should be represented by a collection of icons showing the components within it.

Table 2 Most popular initial codes

Code	P	O
<i>Input/output</i> —inputs and outputs of components	9	28
<i>Examples</i> —examples of components used in composites	9	15
<i>User ratings</i> —some user ratings for services (e.g. star ratings)	8	32
<i>Group by function</i> —grouping services by their function	8	14
<i>Name not representative of function</i> —the name of the component is not representative of the function it performs	8	14

Table 3 Most popular subsequent codes

Code	P	O
<i>Components: input-only versus output-only</i> —the difference between components that only have an input and those that only have an output.	5	10
<i>Terminology confusion</i> —the participant was confused by the terminology used in the tool.	5	8
<i>Choosing components is like choosing apps</i> —the component selection process is similar to the app discovery process.	5	5
<i>Like</i> —the participant liked some aspect of the tool.	4	12
<i>Two versions of tool: high-tech and low-tech</i> —different user technical abilities require different levels of support from the tool.	4	7

- *Making a description for a composition means you can make a composition from a description*—if the process of composition can create a single plain text description of the output, then the process of composition could be powered by a single text description and the process reversed.
- *Automatic composition identification*—the tool should “watch” the user’s actions and automatically identify compositions.
- *Infinite composition*—the output of composition could also be used as the input.

5 Analysis and requirements derivation

This section describes the process by which the codes that were gathered from the requirements gathering sessions were analysed in order to derive a set of requirements.

To outline how requirements were derived from the codes generated in content analysis, we will select a small set of codes to illustrate how the corresponding requirements were identified. The codes that we will present fall into the following categories:

1. Those found in prior work that our work has validated, e.g. control flow, data flow.
2. Concepts that are present in current SC tools but have not been identified as requirements in prior work, e.g. metaphor, composition templates.
3. Concepts not identified in prior work on SC but are visible in related, e.g. user ratings (considered within the domain of mobile apps).
4. Concepts that are not within prior SC work or found in other related domains, e.g. automatic composition identification, infinite composition, and pervasive composition.

Control flow and *Data flow* were codes assigned to participants’ responses when they were asked to describe what a given composition representation and were identified based on how participants phrased their response, (in a similar way to [2]): “The tube is looking up stuff and then it’s notifying you and then it’s the end”—P1. “The tube component *passes something* to the notification”—P6; “You also need to understand how the *data* moves in this on”—P2. Specifically, we were interested in the participants focus on either the ordering of the composition (represented by control flow), or the data being passed between the components (represented by data flow). This analysis yielded two requirements, both of which have been identified in prior work by Mehandjiev and De Angeli [2]:

R1. The flow of control between components should be represented in composition.

The tool should present the order that the components execute in.

Rationale: users need to be able to identify the order in which components are executed in the composition

Category: non-Functional—Operability—

Appropriateness recognizability

Criticality: high

Risk: low

Associated requirements: R44, R44.2, and R44.3

Source: [Control flow]

Prior identification: [2]

COTS: Atooma, Tasker, AutomateIt, IFTTT, Zapier, Automator

R2. The flow of data between components should be represented in composition, if it is present.

The tool should show how the data are passed between the components in the composition.

Rationale: users should be able to identify what data are being passed between components in the composition

Category: non-Functional—Operability—

Appropriateness recognizability

Criticality: medium

Risk: low

Associated requirements: R44, R44.1, and R44.3

Source: [Data flow]

Prior identification: [2]

COTS: Yahoo! Pipes, and Quartz Composer

There are other concepts that can be present in the design space of EUSC tools but have not yet been used as requirements for EUSC. Two examples we identify for this are *Testing* and *Sharing*. Testing of composites is the mechanism provided by most EUSC tools for the user to verify that the composite they have created will execute correctly and performs the task that they intended. Sharing allows users to share what they have created with other users, as well as discovering composites that other users have created. Neither testing nor sharing was identified in the preliminary requirements capture.

R1. The tool should allow users to share services.

The tool should allow users to share services that are created using the tool with other users of the tool.

Rationale: once a user has created and used a composite, they might want to share it with others.

Category: Functional—Management—Monitoring and Adaptation

Criticality: high

Risk: low

Trigger: the user indicates they want to share the composite.

Preconditions: there is a composite to share.

Post-conditions: the composite is shared to a shared composite repository.

Failure effects: the composite is not shared.

Associated requirements: R18 and R23

Source: [Sharing/publishing of composites]

COTS: Atooma, IFTTT, Yahoo! Pipes, AutomateIt, and On{X}

R2. The tool should allow users to test composites.

The tool should allow users to test their composites-in-progress while they are creating them.

Rationale: users need to ensure that composites they create function as intended.

Category: Functional—Verification

Criticality: critical

Risk: low

Trigger: user indicates that they want to test the composite.

Preconditions: the composite contains some components.

Post-conditions: the composite executes in its current state.

Failure effects: the component does not execute as intended

Associated requirements: R15–R17

Source: [Testing]

COTS: Yahoo! Pipes, Quartz Composer, Automator, Tasker, Yahoo! Pipes, and On{X}.

The third type of requirement identified from our study is those relating to concepts not found in current EUSC tools but present in other, linked domains. *User ratings* were suggested as being useful for users to determine the quality of either components or composites that they can discover: “But obviously having ratings for them all would be quite cool too”—P1. “But then ratings would be more important for composites”—P5.

R3. The tool should allow users to rate services.

Users should be able to rate services to convey their opinion on the quality of the service to other users of the tool.

Rationale: users should be able to provide feedback to the creators of components and composites.

Category: Functional—Management—Monitoring and Adaptation

Criticality: medium

Risk: low

Trigger: the user indicates they want to rate the service.

Preconditions: there is a service to be rated.

Post-conditions: the service’s current rating gets

aggregated with the new rating.

Failure effects: the rating is not applied.

Associated requirements: R25

Source: [User ratings]

COTS: AutomateIt, IFTTT, Yahoo! Pipes, and On{X}

The final type of requirement that we will consider is new concepts that the participant would be unlikely to have encountered in this context. The first of these is *Automatic composition identification*: “It would be quite cool for it to be able to identify things for you that you might not think about automating. Like for examples if it watched things you do and suggested compositions for you”—P5. The second is *Infinite composition*: “You might want to use them again”—P3; “And then build them up as well if you can”—P8. This yielded two further potential requirements for such a system:

R4. Potential compositions could be identified automatically.

The tool should be able to monitor the activities of the user and identify tasks that they perform regularly that could be adapted to form a composite. Rationale: if the tool were able to automatically identify potential compositions, it would reduce user burden in deciding what compositions to create.

Category: Functional—Specification

Criticality: low

Risk: high

Trigger: the user performing a manual task repeatedly

Preconditions: the tool is installed on the user’s device

Post-conditions: a composite is created that performs the repetitive task

Failure effects: the composite is not created

Associated requirements: none

Source: [Automatic composition identification]

R5. Composition should be “infinite”.

The tool should allow users to take composites that have been created using the tool and use them as components to be composed in a new composite.

Rationale: composition is a considerably more powerful concept if the user can reuse composites they have created in new compositions.

Category: Structural—Composition structure

Criticality: medium

Risk: low

Associated requirements: none

Source: [Infinite composition]

The categorization we use in this section also highlights the inverse of the first kind of requirement we identified: requirements identified in prior work that were not

validated by our work. The only example of this type of requirement relates to the security of services. This represents a deficiency in our requirements and is discussed in the Requirements Evaluation section.

Following the analysis, there were 7 codes for which we were unable to generate requirements. Codes were unsorted for one of two reasons: they were a generic reference to an aspect of composition, such as the user liking or disliking something; or they were too specific, e.g. the participant commented on the function of a particular component.

6 Requirements

In this section, we describe a subsection of the requirements that we gathered, based on the codes that were gathered from the requirements engineering sessions. Following this, we present an evaluation of the set of requirements as a whole, based on completeness, consistency, and correctness.

We gathered a total of 139 requirements, which, like the preliminary requirements, needed to be organized into groups. First, we separated the requirements into functional and non-functional requirements [13]. Further categorization of requirements was performed based on the model-based validation process which is discussed in the Requirements Evaluation section.

A number of the requirements gathered could be used in the design of an EUSC tool in their current state. There are also a number that require further investigation before they could be used in the specification of an EUSC tool. We will discuss three of the codes on which these requirements were based, all of which relate to some aspect of automation in the tool: *Automatic composition identification*, *Automatic description generation*, and *Automatic composition generation from description*.

- *Automatic composition identification*—Participants suggested this code as a feature where the tool would “watch” the tasks that the user performs and is able to create a composite service to perform a task that the tool “saw”. While this would be an interesting requirement for an EUSC tool, high levels of automation in composition is something that generally is not recommended [27].
- *Automatic description generation* means that the tool is able to automatically generate a description for a composite based on the components that are in the composition. A number of current tools allow the user to enter their own description for composites that they create, but none are able to automate this process.

- *Automatic composition generation from description* is effectively the inverse of *Automatic description generation* in that the tool should be able to automatically create a composite based on a description of the required composite. Hence, it would first be necessary to enable automatic description generation before tools would be able to utilize this method of creating a composite.

6.1 Requirements evaluation

To demonstrate their applicability and usefulness, our requirements must first be evaluated. There are three main properties of requirements against which we can evaluate them: completeness, correctness, and consistency [28].

6.1.1 Completeness

Completeness is an important property of the requirements for a system given that incompleteness of requirements has been identified as one of the most common causes of system failures and accidents [28]. It is also one of the more difficult problems to detect in a requirements specification [28].

Ensuring requirements completeness can be done in a number of ways including: model-based evaluation, individual evaluation, requirements metadata, comparison with repositories, and creation of a specification document [29]. Given the resources available to us, we chose to individually evaluate each requirement, as well as evaluating both our functional and non-functional requirements against a model. Model-based evaluation also provided a useful categorization for our requirements.

There are a number of types of model against which functional requirements can be evaluated, including formal models, event models, process models, data models, etc. [29]. In the Background section of this paper, we identified a process model based on a combination of two suggested EUSC life cycles [6, 7] that could be used to evaluate our functional requirements for completeness. We categorized our functional requirements against the stages of this model, identifying the number of requirements present at each stage. Figure 4 shows the breakdown of requirements within each of these stages.

All stages of the model contain requirements, although the distribution is far from uniform. We suggest that this is based on the user's perception of and involvement in each of the stages. That is, the specification/request stage contained few requirements because of the difficulty of separating specification/request from discovery—it is hard to imagine how the user would specify to the EUSC tool what she wanted without discovering components to do so.

Fig. 4 Distribution of requirements across the stages of the life cycle model

Deployment and annotation were another stage with few requirements, and we believe that this is due to: (1) this is a stage that would have little involvement from the user and (2) it was not the subject of any of the probe questions that were part of our sessions. So, the two requirements were both elicited from unprompted comments made by participants.

The model against which we evaluated our non-functional requirements is the quality model presented in ISO/IEC 25010 [4]. This model contains 12 categories, each with a number of sub-categories: functional suitability, reliability, performance efficiency operability, security, compatibility, maintainability, transferability, usability, flexibility, and safety. When we applied this model to our non-functional requirements, we found that the vast majority (76) were classified within the category of operability, 6 within maintainability, and none in any of the other categories. A further 12 did not fit within any of these categories. The breakdown of requirements in the sub-categories of operability is shown in Fig. 5. Within the maintainability category, 5 requirements were associated with changeability and one with modularity.

Fig. 5 Distribution of requirements within the operability category

We believe that our participants identified a large number of requirements within operability because this is a category that is associated with the use of the tool, whereas a number of the quality aspects were ones which the user may not associate with their use of a piece of software—such as transferability or safety. Furthermore, none of the probe questions in our sessions referred to quality aspects, which explains the very uneven distribution of requirements across these categories. The majority of the requirements within appropriateness recognizability related mainly to the attributes of services that the user could use to recognize whether they were appropriate to their needs or not.

The remaining non-functional requirements that covered topics not relating to quality were grouped into a further three categories: representation/UI (4), interaction (1), and architectural requirements (7).

Following the assessment of our requirements against the models, we evaluated each requirement individually against a number of properties as well as metadata. For each requirement, we identified the following: rationale, source, category, criticality, risk, prior identification of that requirement, associated requirements, software that implements that requirement, as well as any clashes or conflicts with other requirements. Further, for each of the functional requirements, we identified the following: trigger event, preconditions, post-conditions, and failure effects. A full list of the requirements with these properties included can be found in “Appendix 2”.

6.1.2 Consistency

Consistency of requirements relates to conflicts between requirements. In particular, they should not contradict one another [28]. It can also refer to consistent use of terminology.

One conflict we identified was in the requirements relating to the grouping of services within the tool. The potential conflict was between one requirement stating that services should not be grouped (R5.8), and several others saying that they should and specifying how they should be grouped (R5, R5.1–R5.7). To address this potential conflict, we took a different view on the requirement that stated that grouping was not required. A number of requirements relating to grouping of services or components were focused on metrics that could be used to group the services. Thus, the tool could apply a view on the components where they were not organized into groups and were instead presented as a single list, and the potential conflict is resolved. This solution also validates one of our other requirements, stating that grouping should be customizable (R60).

The amount of freedom or restriction within the composition process was another area in which we identified

conflict. Specifically, one requirement said that composition should be free and unrestricted (R35), and another said that the process should be restricted (R59). The solution to this conflict was identified in another requirement relating to providing templates for composition (R34): multiple templates could be provided, some which restrict the possibilities of composition, and others that allow it to be completely “free”. This solution also addresses another potential conflict, identified where composition should be both simple (R50) but at the same time comprehensive, complex and hence more powerful (R11), which can also be facilitated by providing different templates.

6.1.3 Correctness

Correctness of requirements has been defined as the interplay between completeness and consistency [28], although practically, correctness of requirements relates to the mapping of the requirements to the actual needs of the users of the final system [28]. Since we do not have a single business customer, and the preferences and backgrounds of our potential users are likely to be wide and varied, it is difficult definitively to say that all requirements are “correct” for all potential users.

Given that our requirements were elicited after the sessions, we were unable to evaluate the correctness of our requirements with these participants. We also felt that given the varying opinions of these participants, we would be unlikely to get a definitive answer. Instead, we first identified where requirements had been identified previously: either in other sets of requirements for EUSC tools, or implemented in EUSC tools themselves. Roughly half of our requirements were identified as being correct in this way.

We used peer review to evaluate the correctness of the remaining requirements [30]. Thus, we invited researchers knowledgeable in the domain of SC and EUSC to review the requirements individually and as a set, and they found that none of the requirements was incorrect. We have also shown that our requirements are functionally complete and consistent, providing further corroboration of the overall correctness of the requirements.

6.1.4 Requirements validity

We have been able to demonstrate that our requirements are consistent, functionally complete, and correct. The overall completeness of the non-functional requirements was reduced by our use of end-users as the source of these requirements, as they did not consider aspects that a developer or business customer might identify. This is of course unavoidable when end-users are used as the source of the requirements.

Our requirements have not yet been validated directly by, for example, being used to influence the design of an EUSC tool. However, we can provide arguments for the validity of our requirements set based on comparisons with prior work, and the method used to gather them. The two examples of prior work aimed at gathering requirements for EUSC tools yielded two distinct sets of requirements: one which is high level and looking for requirements for SC in general [3], and one very specific, focusing on a particular aspect of composition, i.e. flow [2]. Our requirements contain both high level and specific requirements, including an overlap with the flow-based requirements from prior work. Some requirements were more present in some areas rather than others because of the use of probe questions as part of our sessions, and the limitations of this will be discussed in the next section. We can see from their abundance that this did not inhibit the generation of more general, high-level requirements. Furthermore, the topics of the probe questions were derived from prior work across a wide range of EUSC literature and tools.

7 Limitations and future work

The main limitation in our set of requirements is the lack of coverage of non-functional requirements. There are a number of categories that are not covered by our requirements, and the requirements are not distributed uniformly across those categories that are covered. This limitation is caused by a combination of factors. First, our decision to have requirements motivated by potential end-users meant that they were less likely to identify the various types of non-functional quality-based requirements. Secondly, our probe questions and prompts focused on aspects that the user would interact with directly, rather than aspects such as maintainability, security, and safety.

To address this limitation in future work, we suggest that a section of each session be explicitly dedicated to the discussion of quality-based non-functional requirements of the application being developed. This would help ensure that participants are prompted about the topics for which we have not been able to gather any requirements and should present a more even distribution of requirements across these categories.

Another limitation of the requirements is the disparity between the relative maturity of the requirements. Some are usable immediately and are applicable across all types of EUSC tools. Some present various options that the designer could choose between, or choose all of. There are also a number of requirements that require further investigation before they could be used to motivate the design of an EUSC tool. Thus, further validation of our requirements is an important next step. Validating these requirements

could take a number of different avenues: EUSC tool creation, investigation of particular concepts (e.g. automatic description generation for composites), or further investigation of the design space for EUSC tools. Given the apparent popularity of design space research in EUSC and mashup tools [25, 31–33], this seems to be a promising avenue for requirements validation.

Determining the relative priority of the requirements we gathered is a potential next step. This is particularly important given the size of the set of requirements derived, as well as the disparity in how the requirements could be used. We do not believe that basing priority on the relative popularity of the codes used to derive them is a sufficiently robust method for establishing priority. However, determining relative priorities could be part of our future work on the design spaces of EUSC tools.

After relative priorities have been determined for the requirements, our future work could move on to using these requirements to generate potential designs for EUSC tools, as well as being the basis upon which an EUSC tool can be built.

When detailing the analysis and requirements gathering process earlier in this paper, we identified several groups into which requirements could fit, based on our prior requirements (“Appendix 1”) and the design of current EUSC tools:

1. Those found in prior EUSC requirements gathering studies that our work has validated.
2. Those found in the design space for EUSC tools but not in prior EUSC requirements gathering studies.
3. Those not found within the design space for EUSC tools, but are found in related domains.
4. Those not within the design space for current EUSC tools or related domains.

There are two groups that we have not yet considered:

- Those present in the design space for EUSC tools that were not identified in our study.
- Those present in prior EUSC requirements gathering studies that were not identified in ours.

The former category is unsurprising, as the design space for EUSC tools could contain a theoretically infinite number of design decisions and options and hence a theoretically infinite number of potential requirements [34]. We can also identify one example of the former: security. Security of component services was identified by Namoun et al. [3], but was not identified as being a category in its own right, and hence was not the subject of a question in our study. Furthermore, none of the composition tasks within our study sessions required the user to consider their personal information. It is likely that if one of the tasks had required their personal information (or some other sensitive

Fig. 6 Method overview

information), then participants would have been more aware of security as a requirement.

The omission of security of a requirement highlights a limitation in the method used to gather requirements: priming of participants on selected topics. Participants were much more likely to identify requirements for those topics that had already been identified by the requirements engineer and subsequently used as probe questions. This problem could be mitigated by generating new probe questions from the subsequently identified categories and using these probe questions for later sessions. However, this approach would also run the risk of overloading participants in the later sessions.

Within the SCRAM sessions, we were not able to perform precisely the same composition task using each of the EUSC tools due to their different domains and contexts of use, although the same tasks were performed across all sessions. This meant that as well as the design of the tool varying, the task being performed also varied somewhat, although the nature of the task—SC—remained constant across all tools within the session. We felt it more important to introduce participants to the diversity of SC and hence mitigate the bias towards our prototypical demonstrator rather than to perform precisely the same tasks across each tool.

Our method was designed to prime participants on certain topics within EUSC to ensure that they were familiar with them and could provide requirements in those areas. This priming was reflected in the analysis by grouping the categories into two: initial (primed topics) and subsequent (un-primed). On average, 85.1 % of identified codes represented initial codes (range 78.9–88.3 %, $\sigma = 2.8$ %). Given the priming deliberately built into the method, it is very likely that using different participants would not greatly affect this, and a majority of participants would still refer to topics within initial categories.

Our method was very demanding both in terms of work required in the sessions and the analysis of the data to gather the requirements. After 10 participants, we had enough data from which to derive a large, robust set of requirements. Restricting the sessions to 10 participants minimized the overlap of content between participants, and also the work required to analyse these data and produce requirements. In our study, participant selection required only that

participants were familiar with the platforms upon which the EUSC tools are running, as the sessions provided an introduction to EUSC itself. This approach could work similarly in other domains, as outlined in the following section.

7.1 Method generalizability

The primary focus of our work is to gather requirements for EUSC tools, and to do this we extended and made a number of changes to an existing requirements gathering method: SCRAM. Our proposed method can easily be adapted for use in other domains by making a number of small changes. An overview of the stages within the method is given in Fig. 6.

To adapt the method to a new domain, the preliminary work would need to be applied in the new domain in order to gather requirements for design and implement a prototypical demonstrator in that domain. Our initial requirements capture was based on prior studies on gathering requirements, and a review of applications that have already been created within the domain. This part of the method could therefore be carried out similarly in any domain for which prior requirements gathering has been done or in which existing tools can be profiled. For domains in which such resources are not available to the requirements engineers, we suggest falling back on the recommendations made by Sutcliffe and Ryan in SCRAM: performing conventional interviews and fact-finding [16]. If end-users are used as interviewees in the requirements gathering interview sessions, then this fact-finding could easily take the form of questionnaires which can be distributed widely, for example by on the Web, rather than one-on-one interviews.

Given an initial set of requirements derived through the process above, the next stage, i.e. design and implementation of the prototypical demonstrator, could be carried out using standard techniques and tools.

Once the demonstrator has been implemented, the next step is to research topics in the domain that can be used as probe questions and hence as initial categories for the content analysis of the sessions' outputs. Concurrently, other tools in the domain should be identified. From this research and tool identification, a scripted task could be created to present the various features of the demonstrator, other available tools, along with probe questions. The

sessions themselves would then run in the same way as reported here, while being recorded.

After the sessions, transcription and content analysis could proceed in exactly the same way as described here by performing directed content analysis using the initial categories identified in the background research. Following the transcription and analysis, requirements elicitation could then proceed based on the codes identified.

8 Conclusion

The work reported here has two main contributions:

1. The creation of a comprehensive set of 139 requirements for EUSC tools, which we have demonstrated to be valid.
2. A thorough, generalizable, and repeatable method used to generate these requirements.

The set of requirements gathered here is a larger and more comprehensive set than any gathered in prior work covering a broad range of topics within EUSC. We evaluated our requirements for completeness, consistency, and correctness. Previous approaches covered only one topic within EUSC: flow. Our requirements covered a number of topics, necessitating further categorization of our functional requirements based on aspects of the EUSC life cycle. The non-functional category covered aspects of quality-based requirements such as operability and maintainability but lacked discussion of aspects such as security and performance. A full, categorized list of the requirements gathered can be found in the Appendices.

One contribution of these requirements is to validate some of those gathered in prior EUSC requirements gathering approaches. The validity of these requirements is evidenced by the breadth of topics they cover, as well as the number of requirements we were able to gather within these topics. This is particularly useful when we consider the small number of requirements that have been gathered previously. More evidence of the validity of our requirements is the robust method used, which is based on an already-established method for requirements gathering.

Reflecting further on the methodology, we have suggested how it could be applied in other domains with relatively little modification. We believe the adaptations and extensions made to SCRAM were positive, in that we were able to adapt the basic method to cater for gathering requirements from end-users, as well as incorporating a mechanism for eliciting requirements from the output of each session without imposing a burden on participants. We made modifications to every stage of SCRAM as expressed in its initial specification [16, 17] as well as providing insight into how requirements can be elicited from the output of the interview sessions, something which is not part of the original method.

Of the requirements we present, a number are directly applicable to EUSC immediately: in their current state, they can be applied to the design process for future tools, evidenced by a number of our requirements being used in currently available EUSC tools. No single current EUSC tools implements more than half of the requirements generated (the majority implement considerably fewer than half), suggesting that tools implementing these requirements could significantly improve current designs. The requirements themselves are demonstrably consistent, functionally complete, correct, and can be used by designers of EUSC tools to inform and inspire the design of future EUSC tools, and in refining those that already exist.

Acknowledgments This research was funded by the Mobile VCE and Engineering and Physical Sciences Research Council grant EP/G06167X/1.

Open Access This article is distributed under the terms of the Creative Commons Attribution License which permits any use, distribution, and reproduction in any medium, provided the original author(s) and the source are credited.

Appendix 1: Preliminary requirements list

Preliminary requirements from literature

These requirements were identified in previous studies, either explicitly to gather requirements for EUSC/SC, or those that have specified requirements before performing work on SC.

Functional

PR1 The tool should support each stage of the EUSC life cycle.

All of the stages of the EUSC life cycle.

Source: [6, 7]

PR1.1 Request/Discovery.

The tool should allow users to request and discover components that they can then use in composition to create composites.

Source: [6, 7]

PR1.2 Composition.

The tool should allow users to coordinate components to create a composite.

Source: [6, 7]

PR1.3 Verification/Validation.

The tool should allow users to verify that the composite they have created executes successfully and completes the task that they intended.

Source: [6, 7]

PR1.4 Annotation/Deployment.

The tool should allow users to add information to the composites they create and then deploy them so that they can be executed.

Source: [6, 7]

PR1.5 Execution.

The tool should allow users to execute composites that they have created.

Source: [6, 7]

*Non-functional***PR2 The tool should display services by their user interfaces.**

Services in the tool should be represented by their user interface during the composition process.

Source: [3, 9]

PR3 The composition canvas should be large.

The composition “canvas” should be large enough to allow users to interact with the services on it easily.

Source: [3]

PR4 Services should be secure.

Services used by the tool that require personal information should be secure.

Source: [3]

PR5 Feedback to users should be continuous and proactive.

Users should be provided with feedback throughout the composition process. This feedback should not have to be prompted.

Source: [3]

*Structural***PR6 Sets of data should be treated as a single item.**

If a service returns a list of the user’s friends, it should only indicate that it returns a single item.

Source: [2]

PR7 Users should be assisted in resolving problems with dependencies between the invocation of services.

If there are problems with control flow in the composition, e.g. the execution of one service requires the execution of another service that will never be executed, then the user must be assisted with resolving this.

Source: [2]

PR8 The UI of the composite service being created should be represented.

The user of the tool needs to be able to see what the output of the composition process will look like. It should also be linked with the sections in the composition that are presenting that information. Note that this is only applicable to presentation layer composition.

Source: [2]

PR9 Automate repetitive tasks.

Any tasks in the tool that the user has to complete over and over again should be automated where possible.

Source: [10]

PR10 Users should be able to modify composition templates by removing optional tasks or rearranging the flow of the composition.

If the SC tool provides users with templates that they can fill in, they must also be able to edit the order of the components in this template, or remove optional tasks from the template.

Source: [10, 34]

PR11 Structured flow options should be provided.

The flow in the structure of the composition should provide structured elements such as looping and branching.

Source: [35]

PR12 Allow the configuration of the composition at run-time.

The tool should allow the user to configure the composition at runtime as well as at design time.

Source: [35]

PR13 Allow multiple instances of the same composition to run at once (with different parameters).

Once the user has created a composite, they should be able to “instantiate” it multiple times with different parameters. For example, they might want to have the same composite that operates across multiple accounts for a social network, so a separate instance would be needed for each account.

Source: [35]

PR14 The result of the composition must be usable immediately

Once the user has created a composite, they must be able to execute it immediately.

Source: [11]

PR15 The outputs of composition should be extremely diverse.

SC tools should allow the user to create a large number of diverse composites.

Source: [11, 18]

PR16 Binding to services should be automatic and switch depending on the availability of services.

Services that need to be bound (i.e. those in the environment) should do this automatically based on their availability. For instance, if the user moves location then the tool should bind with services in that location automatically.

Source: [12]

PR17 Remote and local services should be represented in the same way.

Services should be represented in the same way regardless of their location.

Source: [12]

PR18 Abstraction layers should be used to hide complexity.

The inherent complexity of the composition process should be hidden from users by the use of abstraction layers.

Source: [9]

PR18.1 Code should be hidden from the end-user.

The code that underpins the composition process should be hidden from the user.

Source: [9]

Appendix 2: Requirements list subset

Due to the length of the entire requirements list, we only present a subset of the requirements gathered in our study here. A full list of requirements can be found at: http://www.cs.bath.ac.uk/pervasive/publications/2014/EUSC_Requirements_List.pdf.

Functional requirements*Specification***R1. Potential compositions could be identified automatically.**

The tool should be able to monitor the activities of the user and identify tasks that they perform regularly that could be adapted to form a composition.

Rationale: if the tool were able to automatically identify potential compositions, it would reduce user burden in deciding what compositions to create.

Category: Functional—Specification

Criticality: low

Risk: high

Trigger: the user performing a manual task repeatedly

Preconditions: the tool is installed on the user's device

Post-conditions: a composite is created that performs the repetitive task

Failure effects: the composite is not created

Associated requirements: none

Source: [Automatic composition identification]

R2. The tool should be able to generate a composite from a text description.

Assuming it is possible for the tool to generate a description based on the components that are in the composite, this process should be reversible

Rationale: users may not want to have to manually connect together the components, instead they may just want to describe what they want and have the system create a composite based on this text description.

Category: Functional—Specification

Criticality: low

Risk: medium

Trigger: the user entering a description

Preconditions: the tool is installed

Post-conditions: a composite is created

Failure effects: the composite is not created

Associated requirements: none

Source: [Composition to Description → Description to composition]

Subsequent

*Discovery***R3. Users should be able to discover and acquire components.**

Users should be able to acquire to acquire new components (either directly or indirectly) through the tool.

Rationale: in order to perform composition, users need to first discover the components that they are then able to compose together.

Category: Functional—Discovery

Criticality: critical

Risk: low

Trigger: none

Preconditions: tool installed

Post-conditions: components discovered

Failure effects: no components discovered; wrong components discovered

Associated requirements: R4.1

Source: [0, R9, Acquiring components]
 Prior identification: [6, 35]
 COTS: AutomateIt, Atooma, IFTTT, Zapier, Yahoo! Pipes, and On{X}

R4. Users should be able to search for services.

The tool should allow users to search for services within the tool.

Rationale: to find either components to be composed, or composites created by others that they can use.

Category: Functional—Discovery

Criticality: high

Risk: low

Trigger: user enters a search term

Preconditions: tool running, connected to component repository, connected to composite repository.

Post-conditions: a service is returned.

Failure effects: no service is returned, the wrong service is returned.

Associated requirements: R4.1, R4.2

COTS: Quartz Composer, Automator

R4.1. Users should be able to search for components.

The tool should allow users to search for services within the tool.

Rationale: if the user is looking for a particular component, they need to be able to search for it.

Category: Functional—Discovery

Criticality: high

Risk: low

Trigger: user entering a search term.

Preconditions: tool running, connected to component repository.

Post-conditions: a component is returned.

Failure effects: no component is returned, the wrong component is returned.

Associated requirements: R4.1.1 and R4.1.2

Source: [Search by function]

COTS: Quartz Composer, Automator

R4.1.1. Search for components by function performed.

Users need to be able to search through components as part of the discovery process.

Rationale: if the user has an idea of the function that they want from a component, but not the name of the component, they should be able to search for that function.

Category: Functional—Discovery

Criticality: low

Risk: low

Trigger: user enters a search term.

Preconditions: tool installed, connected to component repository, component functions documented.

Post-conditions: component returned.

Failure effects: no component returned and wrong component returned

Associated requirements: none

Source: [Search by name]

COTS: Quartz Composer, Automator

R4.1.2. Search for components by the service provider.

Users should be able to search for components by the application in which the component is provided.

Rationale: if the user knows they want a component provided by a particular service, but does not know its name, they should be able to search for this service provider.

Category: Functional—Discovery

Criticality: low

Risk: low

Trigger: user enters a search term.

Preconditions: tool installed, connected to component repository, component service providers documented.

Post-conditions: component returned.

Failure effects: no component returned and wrong component returned.

Associated requirements: none

Source: [Search by service provider]

R4.2. Users should be able to search for composites shared by others.

Users should be able to search through composites that have been created and shared by other users.

Rationale: rather than “reinventing the wheel”, the user should be able to search for a composite that may have been created by another user to perform the task that they want to perform.

Category: functional—Discovery

Criticality: medium

Risk: low

Trigger: user enters a search term

Preconditions: tool installed, connected to composite repository.

Post-conditions: composite returned.

Failure effects: no composite returned and wrong composite returned.

Associated requirements: R4

Source: [Search for shared composites]

COTS: IFTTT, AutomateIt, Yahoo! Pipes, and On{X}

R5. Services should be sorted into groups, or ranked.

Lists of services in the tool should be separated out into groups, or ranked based on some property of the service.

Rationale: Users should be able to browse through components if they do not know exactly what they are looking for, and a grouping/ranking mechanism would make the browsing process easier.

Category: Functional—Discovery

Criticality: High

Risk: Low

Source: [Grouping]

Prior identification: [12]

COTS: Tasker, Atooma, IFTTT, Zapier, Yahoo! Pipes, Automator, and On{X}

Associated requirements: R5.1–R5.8

R5.1. Services should be grouped by their function.

Services should be grouped based on the function that they perform. For example, all of the components involving interactions with social media could be in a social category.

Rationale: Services can be grouped by their function in order to collect similarly functioning services together.

Category: Functional—Discovery

Criticality: Medium

Risk: Low

Source: [Grouping by function]

COTS: Tasker, Automator, Yahoo! Pipes, and On{X}

Associated requirements: R5, R5.1–R5.8

R5.2. Components should be grouped by their location.

Components should be put into groups based on where they can be found. For example, components on the device might be in one group, and components from the Web in another.

Rationale: The user may want to distinguish between components in a local location over those in a remote location.

Category: functional—Discovery

Criticality: low

Risk: low

Source: [Grouping by component location]

Associated requirements: R5, R5.1–R5.8

R5.3. Components should be grouped by the apps that provides them.

Within their location, components should be grouped by the application that provides them.

Rationale: the user may want to distinguish between components provided by apps that they already have over those that they do not.

Category: Functional—Discovery

Criticality: low

Risk: low

Source: [Grouping by the containing application]

Dependency: Composer component distribution paradigm

COTS: Automator

Associated requirements: R5, R5.1–R5.8

R5.4. Components should be grouped by their cost.

Components should be grouped by their cost, or the cost of the application that provides the component.

Rationale: Users may want to distinguish between components that will cost them money and those that would not.

Category: Functional—Discovery

Criticality: Low

Risk: Low

Source: [Grouping by cost]

Associated requirements: R5, R5.1–R5.8

R5.5. Components grouped by their rating.

Components should be grouped by their rating—top rated services should be shown in a separate section.

Rationale: Users may want to distinguish between components whose quality has been assessed as being higher or lower than others.

Category: Functional—Discovery

Criticality: Medium

Risk: Low

Source: [Top Rated]

Associated requirements: R5, R5.1–R5.8

R5.6. Components should be grouped by the service provider.

Components should be grouped by the provider of the service, or the developer of the service.

Rationale: If browsing for a component provided by a particular service, the user may want to find other components that are also provided by that service.

Category: Functional—Discovery

Criticality: Medium

Risk: Low

Source: [Grouping by service provider]

COTS: Atooma, IFTTT, and Zapier

Associated requirements: R5, R5.1–R5.8

R5.7. Components should be ranked by previously used.

Components should be grouped to show those that have been used previously by the user.

Rationale: If a user has previously used components, they may want to find these components again with little effort.

Category: Functional—Discovery

Criticality: Medium

Risk: Low

Source: [Previously used components, recently used]

Associated requirements: R5, R5.1–R5.8

R5.8. The tool should not allow users to group services.

Services should not be grouped.

Rationale: Users may not want to have any sort of grouping imposed on their browsing of services.

Category: Functional—Discovery

Criticality: Medium

Risk: Low

Source: [No grouping]

COTS: Quartz Composer

Associated requirements: R5, R5.1–R5.8

R6. Services should be grouped into groups and subgroups.

Grouping should allow for services to be put into groups and then subgroups.

Rationale: Grouping metrics may have more than one level. Grouping should reflect this.

Category: Functional—Discovery

Criticality: Low

Risk: Low

Source: [Groups and subgroups]

Associated requirements: R5, R5.1–R5.8

R7. Services should be “tagged” with keywords to represent their function.

Instead of being grouped, services should be assigned tags describing their functionality.

Rationale: Grouping can be restrictive, tagging is more flexible.

Category: Functional—Discovery

Criticality: Low

Risk: Low

Source: [Tagging]

COTS: Atooma

Associated requirements: None

Composition: Design and construction

R8. The tool must allow users to compose services.

Users must be able to create composite services from component services using the tool—the primary function of an EUSC tool.

Rationale: This is the main function of a EUSC Tool

Category: Functional—Composition

Criticality: Critical

Risk: Low

Trigger: The user chooses to create a composite.

Preconditions: The tool is installed.

Post-conditions: A composite is created.

Failure effects: A composite is not created.

Associated requirements: R9

Source: [Composition]

COTS: Atooma, Tasker, IFTTT, Zapier, Yahoo! Pipes, Quartz Composer, Automator, and On{X}

R9. Users should be able to edit the order of the components in composition.

If the components in the composition are positioned in a particular way, the user should be able to modify this initial order.

Rationale: The user may position the components in the wrong order initially, or change their mind about what order they want the components to execute in.

Category: Functional—Composition

Criticality: High

Risk: Low

Trigger: The user moves a component.

Preconditions: There are components in the composition.

Post-conditions: The components in the composition are in the new order.

Failure effects: The components in the composition are in their original order.

Associated requirements: R8

Source: [Editing composition, Customization]

COTS: Tasker, Yahoo! Pipes, Quartz Composer, and Automator

R10. Composition must not involve coding.

The composition process should not involve the user having to write any code

Rationale: EUSC is designed for users who are not skilled enough in programming to create the composite manually.

Category: Functional—Composition

Criticality: Critical

Risk: Low

Trigger: The user indicates they want to create a composite

Preconditions: The tool is installed

Post-conditions: A composite is created

Failure effects: A composite is not created

Associated requirements: None

Source: [Composition—no coding]

Prior identification: [3, 9, 21, 35, 36]

COTS: Atooma, Tasker, IFTTT, AutomateIt, Zapier, Yahoo! Pipes, Quartz Composer, and Automator

R11. The tool should facilitate complex compositions to give the user more power.

The tool should allow the user to create complex compositions so that they can create a wider range of more powerful composites.

Rationale: EUSC seeks to allow end-users to create applications/services that they would otherwise have to ask a developer to make for them. SC should allow such complexity in the composition.

Category: Functional—Composition

Criticality: Medium

Risk: Low

Trigger: The user indicates they want to create a composite

Preconditions: The tool is installed

Post-conditions: A composite is created

Failure effects: A composite is not created

Associated requirements: None

Source: [Complexity is power]

R12. The composition process should be semi-automated.

The process of composition should be semi-automatic to assist the user during composition.

Rationale: Users may need assistance with matching of inputs and outputs in the composition process.

Category: Functional—Composition

Criticality: High

Risk: Low

Prior identification: [2, 3]

COTS: Atooma, AutomateIt, IFTTT, Zapier, Yahoo! Pipes, and Quartz Composer

R13. Composition should work with components whose data types do not match.

Composition of components that have data types that do not match should be allowed since in some cases the execution of a subsequent service does not rely on the data being passed to it by a preceding service.

Rationale: There may be instances where the data are not being used, so the type of that data does not matter.

Category: Functional—Composition

Criticality: High

Risk: Low

Source: [Using components that don't match]

Prior identification: [10]

COTS: Automator and On{X}

Associated requirements: R31.1

*Verification and validation***R14. The tool should allow users to test composites.**

The tool should allow users to test their composites-in-progress while they are creating them.

Rationale: Users need to ensure that composites they create function as intended.

Category: Functional—Verification

Criticality: Critical

Risk: Low

Trigger: User indicates that they want to test the composite.

Preconditions: The composite contains some components.

Post-conditions: The composite executes in its current state.

Failure effects: The component does not execute as intended

Associated requirements: R15–R17

Source: [Testing]

COTS: [Yahoo! Pipes, Quartz Composer, Automator, Tasker, Yahoo! Pipes, and On{X}]

R15. Components should have a test mode.

A “test mode” should be provided with components that interact with external entities. Rationale: some components might have an action that operates on an external entity, but while being tested, the user might not want the component to have any interaction with that external entity. For instance, sending a SMS.

Category: Functional—Verification

Criticality: Medium

Risk: Low

Trigger: User indicates that they want to test the composite.

Preconditions: The composite contains some components.

Post-conditions: The composite executes in its current state.

Failure effects: the component does not execute as intended

Associated requirements: R15.1–R17

Source: [Test mode for components]

R15.1. Testing of triggers should be simulated.

Compositions that are created with triggers to initialize them would only normally be executed when that trigger is fired. Even though this can be done manually in some cases, for testing purposes, users should be able to simulate that this trigger has occurred. Rationale: Triggers (components that execute when a particular event occurs) need the event to occur before they can be executed—restricting when they can execute. This allows them to be tested.

Category: Functional—Verification

Criticality: High

Risk: Low

Trigger: User indicates that they want to test the composite.

Preconditions: The composite contains some components, the first of which is a trigger.

Post-conditions: The component executes in its current state.

Failure effects: the component does not execute.

Associated requirements: R15

Source: [Simulate testing of triggers]

R15.2. Dummy data should be provided for testing.

Some services can acquire data from an external data source; when being tested, these services should provide “dummy” data.

Rationale: components may need to look up data in order to pass it to other components—the user should be able to verify that this will work without having to commit to looking up the data.

Category: Functional—Verification

Criticality: Medium

Risk: Low

Trigger: User indicates that they want to test the composite.

Preconditions: The composite contains some components.

Post-conditions: the component executes in its current state.

Failure effects: the component does not execute.

Associated requirements: R15

Source: [Dummy data for testing]

R15.3. Components should indicate their execution process while they are being tested.

While a component is being tested, it should report the task that is being performed to the composition tool so that the user can better determine where errors occur.

Rationale: the user may need more information regarding what is happening within a component—for instance, if it fails they would be able to see where it failed.

Category: Functional—Verification

Criticality: Medium

Risk: Low

Trigger: User indicates that they want to test the composite.

Preconditions: the composite contains some components.

Post-conditions: the component executes in its current state.

Failure effects: the component does not execute.

Associated requirements: R15

Source: [Execution progress within component]

COTS: Automator

Subsequent

R16. Composition should be debuggable.

The tool should allow the user to debug the composition while they are creating it.

Rationale: If testing the composition fails, the user should be able to step through the execution to find out where it failed.

Category: Functional—Verification

Criticality: Medium

Risk: Low

Trigger: User indicates that they want to debug the composite.

Preconditions: The composite contains some components, the first of which is a trigger.

Post-conditions: the component executes in its current state.

Failure effects: the component does not execute.

Associated requirements: R15

Source: [Debugging]

COTS: Yahoo! Pipes

R17. Testing should only be needed when the composition is sufficiently complicated.

Testing is only needed in a composition tool where the composites being created are sufficiently complex.

Rationale: if a composite is very straight forward, it should not need to be tested.

Category: Functional—Verification

Criticality: Low

Risk: Low

Associated requirements: R15

Source: [Testing complexity]

COTS: Yahoo! Pipes versus IFTTT

*Annotation and deployment***R18. The tool should integrate with users' social networks.**

Users should be able to log into the tool through one or more social networks and connect with the friends that they have on these networks.

Rationale: users may want to share their creations with their friends (or discover friends' creations).

Category: Functional—Annotation and deployment

Criticality: Medium

Risk: Medium

Trigger: the user enters their social network details.

Preconditions: the tool is installed, the user is registered on the social network.

Post-conditions: The user is connected to the social network through the tool.

Failure effects: the user and tool are not connected to the social network

Associated requirements: R23

Source: [View friends' composites, Sharing/publishing of composites on Social Networks]

COTS: Atooma

R19. Descriptions of composites should be generated automatically.

The tool should be able to generate descriptions for composites based on the descriptions of the components that make them up and the logical operations that combine them.

Rationale: the user may not want to think of descriptions for the composites they create, so the tool should provide a facility to do this for them.

Category: Functional—Annotation and Deployment

Criticality: Low

Risk: Medium

Trigger: the user saves their composite.

Preconditions: the user has created a composite containing various components.

Post-conditions: the composite then gets an automatically generated description.

Failure effects: the composite does not get a description.

Associated requirements: R45.2

Source: [Automatically generate descriptions for composites]

*Monitoring and adaptation***R20. The tool should allow users to delete composites.**

The tool should allow users to delete composites that they have acquired or created.

Rationale: once a composite has been used and is no longer required, the user should be able to delete it.

Category: Functional—Management—Monitoring and Adaptation

Criticality: High

Risk: Low

Trigger: the user selects delete.

Preconditions: there is a composite to delete.

Post-conditions: the composite is deleted.

Failure effects: the composite is not deleted.

Associated requirements: None

Source: [deleting entities]

COTS: Atooma, Tasker, AutomateIt, IFTTT, Zapier, Yahoo! Pipes, Automator, and On{X}

R21. The tool should allow users to execute composites.

The tool must be able to execute composites that they have acquired or created.

Rationale: composites need to be executed.

Category: Functional—Management—Monitoring and Adaptation

Criticality: Critical

Risk: Low

Trigger: the user chooses to run the composite, or the trigger in the composite is activated by an external event.

Preconditions: a composite has been created.

Post-conditions: the composite is executed.

Failure effects: the composite is not executed.

Associated requirements: None

Source: [Testing]

COTS: Atooma, Tasker, AutomateIt, IFTTT, Zapier, Yahoo! Pipes, Quartz Composer, Automator, and On{X}

R22. The tool should allow users to share services.

The tool should allow users to share services that are created using the tool with other users of the tool.

Rationale: once a user has created and used a composite, they might want to share it with others.

Category: Functional—Management—Monitoring and Adaptation

Criticality: High

Risk: Low

Trigger: the user indicates they want to share the composite.

Preconditions: there is a composite to share.

Post-conditions: the composite is shared to a shared composite repository.

Failure effects: the composite is not shared.

Associated requirements: R18 and R23

Source: [Sharing/publishing of composites]

COTS: Atooma, IFTTT, Yahoo! Pipes, AutomateIt, and On{X}

R23. The tool should allow users to share composites on social networks.

Users should be able to share what they create on social networks so their friends can interact with them.

Rationale: Users may want to show off their creations with their friends on social networks.

Category: Functional—Management—Monitoring and Adaptation

Criticality: Medium

Risk: Low

Trigger: the user indicates that they want to share the composite.

Preconditions: there is a composite to share, the tool is connected to the social network.

Post-conditions: the composite is shared to the user's social network.

Failure effects: the composite is not shared.

Associated requirements: R18 and R22

Source: [Sharing/publishing of composites on Social Networks]

COTS: IFTTT

Subsequent

R24. The tool should allow users to rate services.

Users should be able to rate services to convey their opinion on the quality of the service to other users of the tool.

Rationale: users should be able to provide feedback to the creators of components and composites.

Category: Functional—Management—Monitoring and Adaptation

Criticality: Medium

Risk: Low

Trigger: the user indicates they want to rate the service.

Preconditions: there is a service to be rated.

Post-conditions: The service's current rating gets aggregated with the new rating.

Failure effects: the rating is not applied.

Associated requirements: R25

Source: [User ratings]

COTS: AutomateIt, IFTTT, Yahoo! Pipes, and On{X}

R25. The tool should allow users to interact with composites created by other users.

Users should be able to interact with the services that other users create and share.

Rationale: users should not need to “reinvent the wheel” and also be able to provide feedback on these composites.

Category: Functional—Management—Monitoring and Adaptation

Criticality: Medium

Risk: Low

Trigger: the user indicates that they want to browse through created composites.

Preconditions: there are composites available to be browsed through.

Post-conditions: none

Failure effects: none

Associated requirements: R22

COTS: IFTTT, Yahoo! Pipes, Atooma, AutomateIt, Zapier, and Yahoo! Pipes

Source: [Sharing]

R25.1. Users should be able to interact with composites created by their friends.

Users should be able to view the composites that their friends have created, and interact with them in the same way they can interact with others.

Rationale: users may want to interact with composites created by people they know, rather than all users of the tool.

Category: Functional—Management—Monitoring and Adaptation

Criticality: Low

Risk: Low

Trigger: the user indicates that they want to browse through created composites.

Preconditions: there are composites available to be browsed through.

Post-conditions: none

Failure effects: none

Associated requirements: R23

Source: [View friends' composites]

Dependent: [R16]

R26. Services should be customizable by editing their parameters.

Users should be able to customize services by editing their parameters.

Rationale: Components and composites may need to be customized after they have been created.

Category: Functional—Management—Monitoring and Adaptation

Criticality: Medium–High

Risk: Low

Trigger: The user chooses to set the parameters of the service.

Preconditions: There is a service to set the parameters of.

Post-conditions: the parameters of the service are set.

Failure effects: the parameters are not set.

Associated requirements: R26.1–R26.3

Source: [Parameters, Component customization], [R59.1–R59.4]

COTS: Yahoo! Pipes, Quartz Composer, Tasker, IFTTT, Atooma, Automator, AutomateIt, Zapier, and On{X}

R26.1. Parameters should be editable during and after composition time, and up to and including at run-time.

Users should be able to edit the parameters of a composite after the initial composition process.

Rationale: the user might want to change the parameters at any time.

Category: Functional—Management—Monitoring and Adaptation

Criticality: Medium

Risk: Low

Trigger: the user changes the parameters.

Preconditions: the composite is in an executable form.

Post-conditions: the composite has its parameters set.

Failure effects: the parameters are not set.

Associated requirements: R26, R26.2, R26.3

Source: [Set parameters after composition, Set parameters at runtime, Set parameters at composition time]

COTS: Yahoo! Pipes, Quartz Composer, Tasker, IFTTT, Atooma, Automator, AutomateIt, and On{X}

R26.2. Parameters should be set to default values at composition time and be editable later.

Users should be able to set default values for parameters at composition time, which can then be edited later.

Rationale: the user should be able to run the component without having to explicitly set the parameters.

Category: Functional—Management—Monitoring and Adaptation

Criticality: Medium

Risk: Low

Trigger: the user runs the composite.

Preconditions: the composite has been created.

Post-conditions: the composite executes with default parameters.

Failure effects: none

Associated requirements: R26, R26.1, and R26.2

Source: [Set default parameters at composition time, Parameters—don't ask again]

COTS: Tasker, IFTTT, Atooma, AutomateIt, Zapier, Quartz Composer, Automator, and On{X}

R26.3. Setting of parameters should use the composite description.

Setting the parameters of the composite should involve changing the description of the composite.

Rationale: using the description to modify the parameters is intuitive, and the description is updated at the same time.

Category: Functional—Management—Monitoring and Adaptation

Criticality: Low

Risk: Low Medium

Trigger: the user modifies the description.

Preconditions: the composite has been created.

Post-conditions: the description is changed, the parameters are changed.

Failure effects: the description and parameters are not changed.

Associated requirements: R26, R26.1, R26.2

Source: [Parameter setting in On{X}]

COTS: On{X}

R27. Components should only need to be activated once.

If the user needs to activate the component, i.e. entering usernames and passwords, this should only need to be done once no matter how many composites into which the component is composed. Rationale: it would be annoying for the user to have to activate components every time they had to use them rather than just once.

Category: Functional—Management—Monitoring and Adaptation

Criticality: high

Risk: low

Trigger: the user activates the component.

Preconditions: the component needs to be activated.

Post-conditions: the component is activated.

Failure effects: the component is not activated.

Associated requirements: none

Source: [Component activation]

COTS: Atooma, Tasker, AutomateIt, IFTTT, Zapier, Yahoo! Pipes, Quartz Composer, and Automator

References

- Laga N, Bertin E (2012) Widgets and composition mechanism for service creation by ordinary users. In: Communications Magazine IEEE, pp 52–60
- Mehandjiev N, Namoun A, Wajid U et al (2010) End user service composition: perceptions and requirements. In: Web services (ECOWS), 2010 IEEE 8th European conference, pp 139–146
- Namoun A, Nestler T, De Angeli A (2010) End user requirements for the composable web second international work light integra web ICWE 2010
- ISO/IEC (2008) ISO/IEC CD 25010: Software engineering—Software product Quality Requirements and Evaluation (SQuARE)—software and quality in use models
- Tosic V, Mennie D, Pagurek B (2000) Dynamic service composition and its applicability to E-business software systems—the ICARIS experience. Object-Oriented Business Solutions ECOOP
- Silva E, Lopez JM, Pires LF, van Sinderen M (2008) Defining and prototyping a life-cycle for dynamic service composition. In: Second international workshop on architectures, concepts and technologies for service oriented computing, pp 79–90
- Mehandjiev N, Angeli A De (2012) End user mashups: analytical framework. In: Proceedings of the second international workshop adaptation services future internet 6th international workshop on web APIs and Services Mashups, pp 36–39
- Paternò F, Santoro C, Spano LD (2011) Engineering the authoring of usable service front ends. J Syst Softw 84:1806–1822. doi:10.1016/j.jss.2011.05.025
- Nestler T, Namoun A, Schill A (2011) End-user development of service-based interactive web applications at the presentation layer. In: Proceedings of the 3rd ACM SIGCHI symposium on engineering interactive computing system ACM, Pisa, Italy, pp 197–206
- Mehandjiev N, Lecue F, Wajid U, Namoun A (2010) Assisted service composition for end users. In: Proceedings of the 2010 eighth IEEE European conference on web services, IEEE Computer Society, pp 131–138
- Albinola M, Baresi L, Carcano M, Guinea S (2009) Mashlight: a lightweight mashup framework for everyone. MEM 2009 WWW 2009
- Bottaro A, Gerodolle A, Lalanda P (2007) Pervasive service composition in the home network. Advanced information networking and applications 2007. AINA'07. 21st international conference, pp 596–603
- Sommerville I (2011) Software engineering, 9th edn. Pearson/Addison Welsey, Harlow, England, New York
- Potts C, Takahashi K, Anton A (1994) Inquiry-based requirements analysis. Softw IEEE 11(2):21–32
- Sutcliffe AG, Maiden NAM, Minocha S, Manuel D (1998) Supporting scenario-based requirements engineering. Softw Eng 24:1072–1088. doi:10.1109/32.738340
- Sutcliffe A (2003) Scenario-based requirements engineering. In: Proceedings of the 11th IEEE international requirements engineering conference 2003, pp 320–329
- Sutcliffe AG, Ryan M (1998) Experience with SCRAM, a scenario requirements analysis method. In: Proceedings of the third IEEE international conference on requirements engineering 1998, pp 164–171
- Aghaee S, Pautasso C (2012) End-user programming for web mashups open research challenges. Current trends in web engineering, pp 347–351
- Geyer L (2000) Feature modeling using design spaces. In: Proceedings of the 1st German Workshop on Software Product Lines
- Baum L, Geyer L, Molter G (1998) Architecture-centric software development based on extended design spaces. In: Development and evolution of software architectures for product family, pp 197–204
- Cappiello M, Matera M, Picozzi M, Sprea G, Barbagallo D, Francalanci C (2011) DashMash: a mashup environment for end user development. In: Web engineering, pp 152–166
- Sutcliffe A (1998) Scenario-based requirements analysis. Requir Eng 3:48–65. doi:10.1007/bf02802920
- Hsieh H-F, Shannon SE (2005) Three approaches to qualitative content analysis. Qual Health Res 15:1277–1288. doi:10.1177/1049732305276687
- Curtis JR, Wenrich MD, Carline JD et al (2001) Understanding physicians' skills at providing end-of-life care. J Gen Intern Med 16:41–49. doi:10.1046/j.1525-1497.2001.00333.x
- Aghaee S, Nowak M, Pautasso C (2012) Reusable decision space for mashup tool design. In: Proceedings of 4th ACM SIGCHI Symposium on engineering interactive computing systems—EICS'12. ACM Press, New York, New York, USA, pp 211–220
- Danado J, Paternò F (2012) Puzzle: a visual-based environment for end user development in touch-based mobile phones. In: Human-centered software engineering, pp 199–216
- Vulcu G, Bhiri S, Hauswirth M, Zhou Z (2008) A user-centric service composition approach. In: Meersman R, Tari Z, Herrero P (eds) Move to meaningful internet systems OTM 2008 Work. Springer, Berlin, pp 160–169
- Zowghi D, Gervasi V (2003) On the interplay between consistency, completeness, and correctness in requirements evolution. Inf Softw Technol 45:993–1009. doi:10.1016/S0950-5849(03)00100-9
- Firesmith D (2005) Are your requirements complete? J Object Technol 4:27–43
- Westfall L (2009) The certified software quality engineer handbook. ASQ Quality Press, Milwaukee
- Minhas SS, Sampaio P, Mehandjiev N (2012) A framework for the evaluation of mashup tools. In: 2012 IEEE 9th international conference on services computing IEEE, pp 431–438

32. Na L, Patel A, Latih R et al (2010) A study of mashup as a software application development technique with examples from an end-user programming perspective. *J Comput Sci* 6:1406–1415
33. Grammel L, Storey M (2010) A survey of mashup development environments. In: *The smart internet*, pp 137–151
34. Cappiello C, Matera M, Picozzi M, Sprega G (2011) DashMash: a mashup environment for end user development case study: mashups for sentiment analysis, pp 152–166
35. Albreshne A, Pasquier J (2011) A template-based semi-automatic web services composition framework (unpublished)
36. Cappiello C, Daniel F, Matera M et al (2011) Enabling end user development through mashups: requirements, abstractions and innovation toolkits end-user development. In: Costabile M, Dittrich Y, Fischer G, Piccinno A (eds) *End-user development*. Springer, Berlin, pp 9–24