Can Situations Help with Reusability of Software?

Hua Ming[1(✉)] and Carl K. Chang[2]

[1] Oakland University, Rochester, MI 48309, USA
ming@oakland.edu
[2] Iowa State University, Ames, IA 50010, USA
chang@iastate.edu

Abstract. Software reusability is an important concept, as well as a powerful tool, to achieve modular solutions in the design and implementation of modern software systems. There is a wide array of research studies conducted in this area ranging from conceptual level to software construction level. Despite all these good pieces of work, software engineers still need to face the complications that strictly separate design time activities from those carried out at software construction time, to shift their mental gear between high level specification properties and low level implementation details. To bridge this gap, we propose a unified approach to facilitate software reuse. We seek to carry out this enterprise surrounding an abstraction, namely *Situation*. More specifically, we have created a computing environment and, under its runtime support, a functional programming language called $Situ^f$ in which domain experts can capture the features of a software system in terms of functional expressions. For each $Situ^f$ program, declarations and functional expressions provide essential definitions of *Situations*. Some language constructs of $Situ^f$, such as the import and include directives, are designed to make it easier to compose new software features by reusing existing ones.

1 Introduction and Related Work

Software reuse [9,10] is a powerful concept and realistic technique in the design and construction of modern software systems. Often in a mutually promotive relationship with modular programming [20], software reuse advocates a systematic embrace for the ideal of fully exploiting existing well tested code towards building new software under reduced development time, increased productivity and reliability.

The support for software reuse from modern programming languages, component-oriented and framework-based technologies, middleware, as well as from the state-of-the-art of modern software construction practices, has steadily improved over the years. All these advantages, among other factors such as psychology of programming and human factor improvement, are good preparations leading software reuse to a certain degree of success [19]. On the flip side however, problems such as idiosyncrasies and heterogeneities between different software

M. Kurosu (Ed.): HCI 2016, Part I, LNCS 9731, pp. 598–609, 2016.
DOI: 10.1007/978-3-319-39510-4_55

applications and application domains have been constraining the growing impact and effectiveness of software reuse.

1.1 The Power of Abstraction and the Abstraction of Situation

Through the development and maturation of software engineering, as well as of programming languages, we have fully witnessed the power of abstraction [11,13,14]. By introducing abstractions, e.g., data abstractions, iteration abstractions, procedural abstractions etc., and in particular, by relating modularity to abstraction [12], the composing, understanding, as well as the debugging and maintenance of a gigantic computer program can be carried out in separate manageable pieces. Consequently, the need for literally going through all the coding details has thus been mitigated.

We propose to utilize an abstraction called **situation**, which is distilled from our previous work on *Situ* framework [5,15–18]. The concept of situation can be traced back to its root in mathematical logic [2,3], and thereafter applied to AI [21] and theoretical computer science [4]. Situation as a concept also marched into the realm of human computer interaction, contributing to the success of situation awareness technologies [8].

Extending from the situation abstraction, we come up with an infrastructure and a functional, domain specific programming language called $Situ^f$ implemented to secure the situation abstraction into concrete and practical software engineering circumstances.

2 Functional Style Situation

From *Situ* framework [5], the behavioral context of a situation represented by A as in $(d, A, E)_t$, refers to the interaction of a software system through its interface, usually a GUI, with its human user. The concept of situation intimately portrays integrated use case scenarios, including the features of the software system.

Functional programming paradigm [1] is about computing with values, where the control flow of the entire program is deeply akin to the evaluation of a mathematical function, with little, or no side effect [1]. It is more of a **what** rather than **how** process. Due to its high expressive power and elegant computational effect, in recent years functional programming paradigm keeps gaining momentum, and has already been absorbed and built into some high impact computing technologies or well known infrastructures. Google's MapReduce [6] for the processing of big data is such an example.

In this work, we argue that functional situations present a powerful abstraction to model software features. The motivation behind this paper is this: **if we can program functional situations, where runtime environment**[1]

[1] Figure 4 is such an environment.

links situations with software features, then software reusability can be translated to situation reusability.

$$map\ (\ (submit\_review.download),\ [paper_1,\ paper_2,\ \ldots\ paper_n]\ ) \qquad (1)$$

(1) above is an example functional expression:

- It features a functional expression, where higher order function **map** is used;
- For **map**, the composite function *submit_review.download* is one of its inputs;
- The composite function *submit_review.download* intrinsically reflects a temporal order: *download* goes first, *submit_review* second;
- (1) specifies a paper review situation, corresponding to a software feature like in MyReview software[2], shown in Fig. 1.

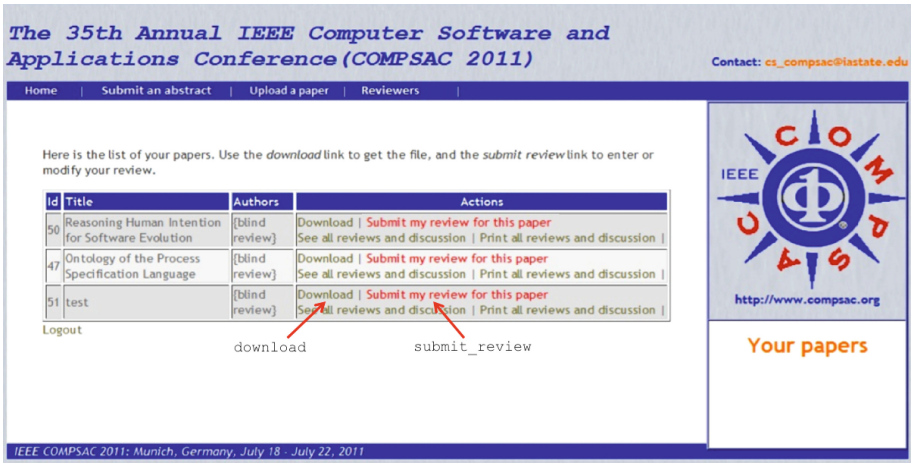

Fig. 1. Paper review situation vs. paper reivew software feature under MyReview software

The most important software features shown in Fig. 1 are marked by red arrows along with the names of the two functions in functional expression (1), i.e., "*dowload*" and "*submit_review*". Further, a moment's reflection reveals that the functional expression (1) captures the bareback essentials of the software features demonstrated in Fig. 1.

Indeed, functional expression (1) models the software features that allow the user, in this case a paper reviewer, to download papers and to submit reviews. It is a **situation** whose semantics is resulted by evaluating the functional expression (1).

It is a key observation from the preceding example that a situation may naturally take its form, or syntax, from extending a functional expression like (1).

[2] http://myreview.sourceforge.net.

In addition, a functional expression like (1) can semantically capture the essentials of the targeted software features. Situations thus proposed is a solid abstraction that carries both its syntax and semantics.

We further state that situation is an easy to use yet powerful abstraction for domain engineers. Using Fig. 1 again as an example: without the situation abstraction, Fig. 1 simply points to a bunch of software features for MyReview system; with the situation abstraction, Fig. 1 is simply one situation, namely paper review situation, for example. Its meaning is expressed via functional expression (1). The abstraction of situation is applied here in a natural and intuitive manner. On the following pages, we present a domain specific, functional programming language, named $Situ^f$ to further promote the abstraction of situations.

3 The Design of a Functional Domain Specific Language

To introduce $Situ^f$, we follow the reverse order: we will first present a program written in it, *i.e.,* Program 1 given in Fig. 2. Program 1 is a $Situ^f$ program for the paper review situation just discussed.

Program 1 A $Situ^f$ program for paper review situation

```
include GUI_Service_MyReview
import Context_Spec_MyReview

program paperReview
  data
    declare
      paper@129.186.93.0:/home/myreview/          \
            COMPSAC2011_Training/Review.php;
    declare
      Review@129.186.93.0:/home/myreview/          \
            COMPSAC2011_Training/Review.php;

  action
    declare
      download<None:paper>@129.186.93.0:/home/      \
          myreview/COMPSAC2011_Training/Review.php;
    declare
      submit_review<paper:Review>@129.186.93.0:/home/      \
          myreview/COMPSAC2011_Training/Review.php;

  situation
    map submit_review.download paper();
```

Fig. 2. A $Situ^f$ program for paper review situation

Program 1 defines context-oriented paperReview situation, following the original *Situ* framework, where all situations are based on behavioral and environmental contexts.

1. The notion of @ creates an IO channel in a $Situ^f$ program called *paperReview* to bind data and action to their real world counterparts: a paper can

be downloaded from Review.php, whose server-side url is specified; Review can be submitted and later on collected also through the same page. Each time a paper is downloaded or a review is submitted through Review.php, the contextual information will be captured by @ and sent back to program *paperReview*. @ is an I/O based language feature. Once declared, data and action can be used to construct a situation.

2. ( ) is another I/O based feature *Situ*^{f} offers. It is a data constructor: at runtime paper() returns a list of papers resulted by a series of paper downloading actions performed on Review.php of the deployed MyReview system. Figure 3 helps illustrate this point.
3. Closely related with SituIO and its @ operator is the <program_url>[3] defined in the attribute grammar of *Situ*^{f} at Table 1. This symbol specifies where *Situ*^{f} runtime is able to find the external counterpart that supplies contextual information to declared data, actions and situations defined in Program 1 (Fig. 2.) Situation services provides the implementation.

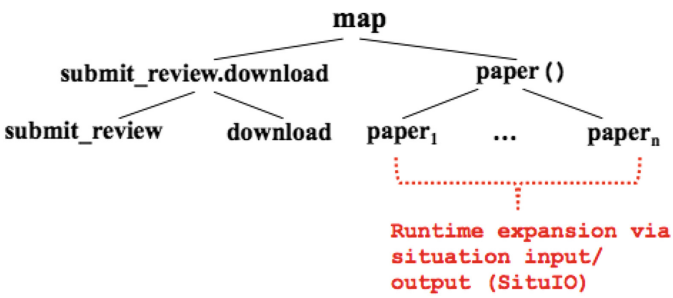

Fig. 3. Runtime expansion of "paper()"

3.1 Syntactical Features

The details of the attribute grammar for *Situ*^{f} are given in Table 1. The data and action declarations in Program 1 (Fig. 2) set up the data, as well as the action to construct a situation. @ operator connects data structures like paper and Review to their real world data source. For Program 1 (Fig. 2) the source of data for paper and review is the server-side *Review.php*. This simply means that each time the user downloads a paper through Review.php, the context data related to that paper such as author list, email contact and abstract etc. ... will be collected over the Graphical User Interface and sent back to Program 1 runtime. More concretely, through paper(), context information of all assigned papers

[3] <prog_url> denotes a program url which takes the form of server_IP_address:serverside_absolute_directory. For programs on your local machine, simply use 255.255.255.255; .

are captured incrementally one after another and are given as input to review action. When the user finishes reviewing that paper and generates a *Review*[4], the *Review* will be captured in terms of its context ensemble: an aggregation of review comments, review score, suggestions to the Program Committee, etc. The communication is carried out while all intermediate results are recorded through XML intermediate representation.

Situ^f provides four built-in functional patterns as situation constructors to propagate contexts, or in attribute grammar's terms: *attributes*, to the entire parse tree. These four built-in patterns are **map, filter, reduce and apply**. The *map* pattern is used in Program 1 (Fig. 2) in statement "map submit_review.download paper()" to describe a situation where a reviewer needs to download and then review every paper assigned to her/him. The *map* pattern, commonly found in functional programming paradigm, applies its first input, i.e. the temporally combined action of downloading and then reviewing ("submit_review.download") to its second input, which is a list of papers. Readers familiar with functional programming know that *map* is a higher-order function that applies the first argument it accepts, which is a function or a composed function, to its second argument, usually a sequence of data such as the paper list aforementioned. *Situ*^{f} introduces *map* pattern so that its first argument can be re-used for all members in its second argument. Overall, applying *map* pattern over a list is to transform the list to another by working on each and every member of the list according to its first argument; in Specification (1), a list of reviewed papers that are attached with review comments and scores etc. are the end result for the main success scenario for Specification (1).

3.2 Situ^f-based Environment

The situation model that *Situ*^{f} is built upon is context-oriented, where context data are derived from actions exerted by a user over a software system. However a software system itself does not provide extra functionality to support context data collection tasks. The design of *Situ*^{f} keeps that in mind and proposes a special *include* directive to "include" situation services that provide context collection capabilities. Situation services are programs with implementation to collect context information for different *Situ*^{f} programs.

With concrete examples, this section elaborates on the technical details of context specification, situation services, their relationship with XML, their affiliation to a *Situ*^{f} program and finally the active roles they play towards a *Situ*^{f}-based environment.

According to the grammar of *Situ*^{f} language, the major constituents of a situation are *data* and *action*s. In a *Situ*^{f} program, the situation constructors, i.e., map, reduce, filter and apply, are used to assemble data and actions declared into a meaningful situation. This means that the context information in a *Situ*^{f} program is classified into two categories: data context and action context. Action context is built on top of data context, as the input and output of each action

[4] the data type declared in Program 1 (Fig. 2.) .

Table 1. Attribute grammar for *Situ*^{f}

(1) <program>	→ [**include** <service_list>][**import** <situation_spec_list>] **program** <identifier> **data** <dataDeclList> **action** <actionDeclList> **situation** <SituStmtList> {$< SituStmtList >_{env} = < dataDeclList >_{env}$ $\cup < actionDeclList >_{env} \cup < service\_name >_{env}$ $\cup < situaion\_spec >_{env}$ }
(2)<identifier>	→ [$a \ldots \mid z \mid A \ldots \mid Z \mid \_]^+[\ 0 \mid \ldots \mid 9 \mid a \ldots \mid z \mid A \ldots \mid Z \mid \_ \mid \backslash]^*$
(3)<dataName>	→ **None** { $< dataName >_{env} = \phi$}
(4)<dataName>	→ <identifier> { $< dataName >_{env} =$ {$< identifier > .id$} }
(5)<dataDeclList>	→ **declare** <dataName>**@**<prog_url> { $< dataDeclList >_{env} = < dataName > .env$ $\cup$ {$< prog\_url > .id$} }
(6)<$dataDeclList^1$>	→ **declare**<dataName>**@**<prog_url>; <$dataDeclList^2$> { $< dataDeclList^1 >_{env} = < dataName > .env$ $\cup$ {$< prog\_url > .id$} $\cup < dataDeclList^2 >_{env}$ }
(7)<action>	→ **None** {$< action >_{env} = \phi$}
(8)<action>	→ <indentifier> { $< action >_{env} =$ {$< identifier > .id$} }
(9)<actionList>	→ <action> { $< actionList >_{env} = < action >_{env}$ }
(10)< $actionList >^1$	→ <action>,< $actionList >^2$ { $< actionList >^1_{env} = < action >_{env} \cup < actionList >^2_{env}$ }
(11)<input>	→ **None** { $< input >_{env} = \phi$ }
(12)<input>	→ <identifier> { $< input >_{env} =$ {$< identifier > .id$}}
(13)< $input >^1$	→ <identifier>,< $input >^2$ { $< input >^1_{env} =$ {$< identifier >_{id}$} $\cup < input >^2_{env}$ }
(14)<output>	→ **None** { $< output >_{env} = \phi$ }
(15)<output>	→ <identifier> { $< output >_{env} =$ {$< identifier > .id$}}
(16)< $output >^1$	→ <idnetifier>,< $output >^2$ { $< output >^1_{env} =$ {$< identifier >_{id}$} $\cup < output >^2_{env}$ }
(17)<actionDeclList>	→ **declare**<actionList>(< $input$ >:< $output$ >) **@**<**prog_url**> { $< actionDeclList >_{env} = < actionList > .env$ $\cup < input > .env \cup < output > .env$ $\cup$ {$< prog\_url > .id$} }
(18)<actionDeclList>	→ **declare**<actionList>(< $input$ >:< $output$ >) **@**<**prog_url**> **;**<actionDeclList> { $< actionDeclList >_{env} = < actionList > .env$ $\cup < input >_{env} \cup < output >_{env}$ $\cup < prog\_url > .id \cup < actionDeclList >_{env}$ }
(19)<situStmtList>	→ <situStmt> {$< situStmt >_{env} = < situStmtList >_{env}$}
(20)<situStmtList^1>	→ <situStmt>**;**<situStmtList^2> {$< situStmt >_{env} = < situStmtList >_{env}$ $< situStmtList^2 >_{env} = < situStmtList^1 >_{env}$}
(21)<situStmt>	→ **map** <actionList> <dataName>**()** {$map_{env} = < situStmt >_{env} \cup < actionList >_{env}$ $\cup < dataName > ()_{env}$}
(22)<situStmt>	→ **filter** <actionList> <dataName>**()** {$filter_{env} = < situStmt >_{env} \cup < actionList >_{env}$ $\cup < dataName > ()_{env}$}
(23)<situStmt>	→ **reduce** <actionList> <dataName>**()** {$reduce_{env} = < situStmt >_{env} \cup < actionList >_{env}$ $\cup < dataName > ()_{env}$}
(24)<situStmt>	→ **apply** <actionList> <dataName> {$apply_{env} = < situStmt >_{env} \cup < actionList >_{env}$ $\cup < dataName >_{env}$}

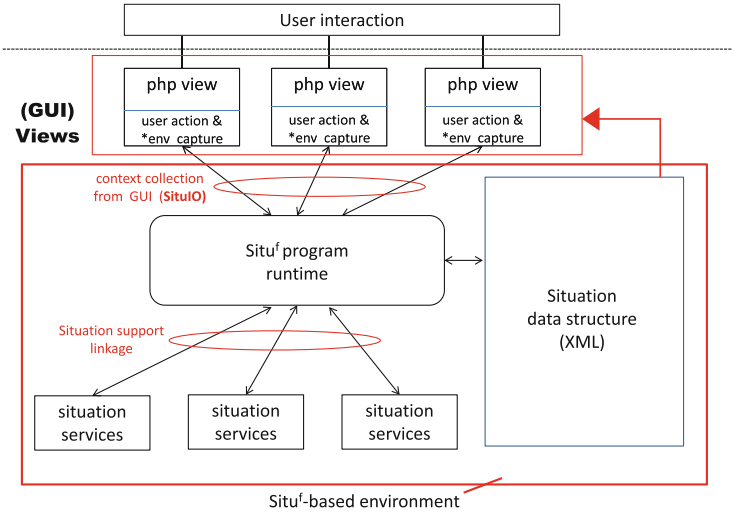

Fig. 4. *Situ*^{f}-based environment: the overview

come from data. We will concentrate on explaining data context, through which action context should seem easy.

In *Situ*^{f} environment, context information, either for data or for action, is represented and transmitted using XML format. We use XML Schema to configure "context" templates to synchronize the communication between a *Situ*^{f} program and the external context collection capabilities, i.e., situation services, under a *Situ*^{f}-based environment.

To provide concrete explanations and illustrations for key issues involved, let us revisit the paper review example given in Program 1 (Fig. 2.) The attribute grammar of *Situ*^{f} given in Table 1 requires that each declared *data*, represented by grammar symbol <dataName>, have an attribute called *env*, meaning *environment*. This is a composite attribute. Its runtime implication depends on the context specification the *Situ*^{f} program imports. In fact each paper declared in Program 1 (Fig. 2) contains the following attributes: *abstract*, *author_name*, *author_affiliation*, *email_contact*, *paperID*, *submitTime*, and *target_trackName*.

This detailed context information is generally beyond the concern or knowledge of a domain expert, but it is very important to answer the attribute grammar requests. *Situ*^{f}'s support of **separation of concerns** [7] bridges this gap. More concretely, *Situ*^{f} offers an *import* clause feature. As seen in Program 1 (Fig. 2), the "Context_Spec_MyReview" following the "import" directive is an instance of <situation_spec>, which is encoded as an XML Schema given in Fig. 5.

In fact, XML Schema enables *user-defined data types*, comprising *simple data types*, which cannot use elements or attributes, and *complex data types*, which can use elements and attributes [22]. Complex data types can also be defined from already existing data types. The XML Schema given in Fig. 5 essentially provides

```
<? XML version="1.0" encoding="UTF-16" ?>

<MyReview:schema xmlns:MyReview="http://www.w3.org/2001/XMLSchema"
version="1.0"s>
<MyReview:element name="paper" type="paperType">
    <MyReview:complexType name="paperType">
        <all>
            <element name="abstract" type="string" use="required"/>
            <element name="author_name" type='string" minOccurs="1"
                        maxOccurs="unbounded" />
            <element name="author_affiliation" type="string" minOccurs="1"
                        maxOccurs="unbounded" />
            <element name="email_contact" type="string" use="required"
                        maxOccurs="1" />
            <element name="paperID" type="integer" use="required" />
            <element name="submitTime" type="date" use="required" />
            <element name="targeted_trackName" type="string" use="required"
                        maxOccurs="1" />
            <element name="conference_name" type="string" use="required" />
        </all>
    </MyReview:complexType>
</MyReview:element>
</MyReview:schema>
```

Fig. 5. An XML schema-based context template for the paper data type

a template to help bind *paper*, a data variable declared in Program 2, and its closely related context. Note that Fig. 5 provides detailed attributes pertaining to the specific situations associated with the MyReview system. The associating power is further enhanced by the use of *namespace* MyReview in Fig. 5. That said, a paper under a different circumstance, such as the "EasyChair" software system, could involve completely different attributes, the use of which requires the importing of a different XML schema. Besides, the use of namespace in an XML Schema helps to disambiguate identical naming and to differentiate between separate situation domains, e.g., MyReview vs. EasyChair[5].

Upon the import of a context specification where relevant information for a paper is provided, the *Situ*^{f} compiler automatically executes the following action (**Note**: the initial *env* attribute of paper only includes its id information. To see that, from production (4) given by the attribute grammar in Table 1: $< dataName >_{env} = < identifier > .id$, when paper is declared, it replaces <dataName>.):

$$paper_{env} = paper_{env} \cup \{ \text{abstract, author\_name, author\_affiliation, email\_contact, paperID, submitTime, targeted\_trackName } \}$$

In Fig. 5, "paper" is defined as a new type, where abstract, author_name, author_affiliation, email_contact, paperID, submitTime, targeted_trackName and conference_name are its built-in fields. Each field, corresponding to the respective context of a "paper", is of a precisely defined data type, such as string, integer, etc. . . The diverse data types available in XML Schema make XML Schema powerful enough to specify highly diverse data different *Situ*^{f} programs may face.

[5] For more background information on namespace mechanism of XML schema, please consult [22].

Figure 6 is a direct instantiation of the XML Schema based context template given in Fig. 5. Given that Fig. 6 strictly follows the format prescribed by Fig. 5, the latter is hence named Context Template.

```
<?xml version="1.0" encoding="UTF-16 ?>

<paper MyReview:schemaLocation="rs.cs.iastate.edu/myreview/context-
      Spec_MyReview">
      <abstract> This paper describes a novel testing approach for ... </abstract>
      <author_name>John Schneider</author_name>
      <author_affiliation> Oakland University </author_affiliation>
      <email_contact>jschneidre@oakland.edu</email_contact>
      <paperID>215</paperID>
      <submitTime>2015-08-31</submitTime>
      <targeted_trackName>Software Testing</targeted_trackName>
      <conference_name>ACM/IEEE ICSE</conference_name>
</paper>
```

Fig. 6. A sample runtime collected context value stored in XML

Figure 6 presents a concrete runtime example of a data value traveling through SituIO. This XML element is a value for the data variable "paper" declared in Program 1 (Fig. 2). It is generated under the governing of "Context_Spec_MyReview" file, which contains the XML Schema given in Fig. 5. The XML context information shown in Fig. 6 for "paper" also presents itself as a sample value for *env* attribute of <dataName>, a grammar symbol instantiated by "paper," from *Situ*^{f}'s attribute grammar in Table 1. Figure 6 shows a concrete instance of context values.

3.3 The Inclusion of Situation Services

Situation services extend the capability of a *Situ*^{f} program that includes them. Situation services are either made by a third party provider and hosted on the cloud, or they can be hosted on the local machine. The default situation service for *Situ*^{f} is called "common_service_GUI". The default service offers the capability that, once deployed at the targeted url site, it can capture and record a software user's action information, which is then sent back through SituIO to where the *Situ*^{f} runtime is deployed. What is captured by the default service is *real time* behavioral and environmental contextual information, which is configured by the central *Situ*^{f} program that generally contains program url addresses.

The design and runtime support environment for *Situ*^{f} as introduced facilitate the domain experts, who have domain specific knowledge of existing software features, to compose new ones. Consider again the software features of MyReview shown in Fig. 1. The paper reviewers can use it to download the assigned papers for a conference and submit their reviews. The corresponding situation program in *Situ*^{f} named "paperReview", is found in Program 1 (Fig. 2.)

Program 2 A $Situ^f$ program for review reminder

```
import paperReview

program reviewReminder

 action
   declare
      email<Review:String>@129.186.93.0:/home/      \
          myreview/COMPSAC2011_Training/Admin.php;
   declare
      check_count<Integer:Bool>@129.186.93.0:/home/      \
          myreview/COMPSAC2011_Training/Util.php;

   declare
      count_words<Review:Integer>@129.186.93.0:/home/  \
          myreview/COMPSAC2011_Training/Util.php;

 situation
    map email (filter check_count.count_words paperReview);
```

Fig. 7. A $Situ^f$ program for review reminder

Now that there is a need to add a new feature named *reviewReminder* to the MyReview system, which aims to send a reminder email, after certain date, to the reviewers who have not finished their review assignments. The overall requirement for the *reviewReminder* feature is to go through all paper reviews and to count the number of words in the review comments, by which empty reviews bear zero word count. Below a certain count value, the relative reviews will be considered incomplete. Correspondingly, a reminder message is emailed to the related reviewers.

A good question to ask is how to take full advantage of, or, **re-use**, the existing system features to compose *reviewReminder*. To this end, being able to **expressively and immediately** compose the essential linkage, between existing features and **reviewReminder**, gives the software designer a leg up towards a high quality software construction. Using $Situ^f$ language and with relative ease, the domain experts can propose a short solution, i.e., a $Situ^f$ program, shown in Fig. 7.

References

1. Backus, J.: Can programming be liberated from the von neumann style? a functional style and its algebra of programs. Commun. ACM **21**(8), 613–641 (1978)
2. Barwise, J.: The Situation in Logic. Center for the Study of Langauge and Information. Stanford University, Stanford (1989)
3. Barwise, J., Perry, J.: Situations and Attitudes. MIT Press, New York (1983)
4. Barwise, J., Seligman, J.: Information Flow: The Logic of Distributed Systems. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge (1997)
5. Chang, C.K., Jiang, H., Ming, H., Oyama, K.: Situ: a situation-theoretic approach to context-aware service evolution. IEEE Trans. Serv. Comput. **2**(3), 261–275 (2009)

6. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: OSDI, p. 1 (2004)
7. Dijkstra, E.W.: On the role of scientific thought. Selected Writings on Computing: A Personal Perspective. Texts and Monographs in Computer Science, pp. 60–66. Springer, New York (1982)
8. Endsley, M.R.: Toward a theory of situation awareness in dynamic systems. Hum. Factors **37**(1), 32–64 (1995)
9. Frakes, W.B., Kang, K.: Software reuse research: status and future. IEEE Trans. Softw. Eng. **7**, 529–536 (2005)
10. Krueger, C.W.: Software reuse. ACM Comput. Surv. (CSUR) **24**(2), 131–183 (1992)
11. Liskov, B., Guttag, J.: Abstraction and Specification in Program Development. MIT press, Cambridge (1986)
12. Liskov, B., Guttag, J.: Program Development in JAVA: Abstraction, Specification, and Object-oriented Design. Pearson Education, New York (2000)
13. Liskov, B., Snyder, A., Atkinson, R., Schaffert, C.: Abstraction mechanisms in clu. Commun. ACM **20**(8), 564–576 (1977)
14. Liskov, B.H., Zilles, S.: Specification techniques for data abstractions. IEEE Trans. Softw. Eng. **1**, 7–19 (1975)
15. Ming, H.: Situ^{f}: a domain specific language and a first step towards the realization of situ framework. PhD Dissertation. Iowa State University. ProQuest Dissertations & Theses Global. UMI 3539397 (2012)
16. Ming, H., Chang, C.K., Yang, J.: Dimensional situation analytics: from data towisdom. In: 2015 IEEE 39th Annual Computer Software and Applications Conference (COMPSAC), vol. 1, pp. 50–59. IEEE (2015)
17. Ming, H., Chang, C., Oyama, K., i Yang, H.: Reasoning about human intention change for individualized runtime software service evolution. In: 2010 IEEE 34th Annual Computer Software and Applications Conference (COMPSAC), pp. 289–296, July 2010
18. Ming, H., Oyama, K., Chang, C.: Human-intention driven self adaptive software evolvability in distributed service environments. In: 12th IEEE International Workshop on Future Trends of Distributed Computing Systems 2008, FTDCS 2008, pp. 51–57, October 2008
19. Morisio, M., Ezran, M., Tully, C.: Success and failure factors in software reuse. IEEE Trans. Softw. Eng. **28**(4), 340–357 (2002)
20. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15**(12), 1053–1058 (1972)
21. Reiter, R.: The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. Artif. Intell. Math. Theor. Comput.: Papers in Honor of John McCarthy **27**, 359–380 (1991)
22. W3C: Extensible markup language (xml) (2003). http://www.w3.org/XML/