2012

situ-f: a domain specific language and a first step towards the realization of situ framework

Hua Ming
Iowa State University

Situ^f: A Domain Specific Language

and

A First Step Towards the Realization of *Situ* Framework

by

Hua Ming

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:

Carl K. Chang, Major Professor

Johnny Wong

Simanta Mitra

Jin Tian

Ting Zhang

Iowa State University

Ames, Iowa

2012

DEDICATION

I would like to dedicate this dissertation to my families and friends, both in Beijing China and in Iowa United States.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Situ proposes a human centered, dynamic reasoning framework for domain experts to evolve their software. It formally models the relationship between externally observed situation sequences and the rapid evolution of that software system, using real-time usage information from users and contextual capturing on the behavior of a software system relative to its runtime environment.

Situ^f is a continuing effort under *Situ* framework [1]. In this effort, a domain specific, functional programming language named *Situ*^{f} is presented from its design, semantics and a feasibility test through theoretical validation. The targeted users of this language mainly include domain experts and engineers who are versed in the major concepts and paradigms regarding human-centric situations. As argued there, human-centric situations are vitally important to infer a user's intention and therefore, to drive software service evolution. *Situ*^{f} is designed particularly to encourage domain experts and engineers to think and work with *situation*s. An attribute grammar based approach is developed so that through *Situ*^{f}, relevant real-time contexts can be systematically aggregated around situations. A computational semantics is offered to precisely describe the runtime behavior of a *Situ*^{f} program. While the *Situ*^{f} language serves as the critical centerpiece of this work, its functioning necessarily requires environmental support from *Situ* elements outside the language itself, such that altogether they give rise to a *Situ* oriented system. This environment, named *Situ*^{f}-based environment, is also introduced.

Keywords: *Situ*-framework, *Situ*^{f}-environment, situation, human intention, software evolution, domain-specific programming language, functional programing language, structural operational semantics

CHAPTER 1. INTRODUCTION

1.1 Review of *Situ* framework

Human intention has long played important roles in both cognitive reasoning and creative software development [2, 3, 4]. How intention can be connected with the software maintenance and evolution process has not been adequately studied [1]. *Situ* framework was developed to bridge this gap towards a rapid, automated software service evolution process [1, 5]. An important concept brought to light by *Situ* framework was that of *human intention* - which is defined as a temporal sequence of *situations* observed towards achieving a common goal. The goal there is meant to be in the context of *system goals* directly related to the goal model used in Goal-Oriented Requirement Engineering [6, 7]. Under *Situ*'s definition, each *situation* at a particular time instant must encapsulate a user's behavioral as well as environmental context from which a user's desire can be served [1]. In order to enrich the concept of *Situation* with more expressive power for software and service evolution, Ming et al., [5, 8] continued to expand this research spectrum under *Situ* framework with new concepts, including *Situ-module*,*Situ-morphism* and *Situ-channel*. While in need of more refined and sustainable efforts, the conceptual cornerstone for the *Situ* framework has been provided by which robust upper-level structures are made possible.

1.2 Declarative situations

In computer programming, the declarative paradigm distinguishes itself from all other paradigms, such as its popular imperative counterpart, in its emphasis to minimize or even eliminate side effects by describing *what* the program should do rather than how to accomplish it. This defining characteristic of the declarative paradigm stands out especially with regard

to the way *functions* are created in both their syntax and, more importantly, their semantics. Popular declarative languages include SQL, ML and Haskell. Declarative programming is of particular interest recently, in both the research and the industry due to the fact that eliminating side effects can greatly simplify the writing and debugging of parallel programs.

While it is very impressive that the declarative programming paradigm goes a long way for simplifying the writing of correct parallel programs, especially in this age of multi-core and multi-processors, it is of special interest for the designer of a domain specific programming language under *Situ* framework. First of all, the description of a *situation*, in its very nature, is a **what** rather than how process.

Let us consider a concrete example. A domain expert trying to schedule a meeting for various parties to attend wants to accommodate as many requested meeting times and locations as possible. To start with, she wants to compile everyone's time and location by first filtering out "impossible" situations for which there are no viable chances, and then reduce the remaining situations to one that would allow most of the intended parties to participate.

The solution that comes out most naturally to help the meeting scheduling expert can be specified as follows:

- the situation for each party is represented as a pair of available time and location;
- the collection of all the availabilities consist in a list of available time and location pairs;
- apply a *filter* function on each party's situation to remove the inviable ones, and;
- *synthesize* all remaining situations into a final situation that works the best to accomodate each remaining situation.

A direct translation from the above scheme is as follows:

$$synthesize \rightarrow (filter \rightarrow [(t_1, loc_1) \ldots (t_n, loc_n)])^{1} \tag{1.1}$$

A declarative language promotes the most straightforward solution leading to a simple computer program. Note that should such a declarative language be available, the domain

[1] This is also known as *map reduce*, a well known scheme with its root from functional paradigm, a subcategory under declarative paradigm.

expert will only need to worry about *what* the filter has to do on each situation, enjoying the complete insulation from *how* the filter is going to be applied iteratively from one situation to the next throughout the list. A similar case occurs in the *synthesis* step as described in the above scheme.

More interestingly, should there be an appropriate declarative language support, such a solution can also serve as an executable high level specification targeting strategic design objectives.

Second, a declarative paradigm is consistent with *Situ* defined *situation*s. According to the definition by *Situ*, each situation carries a time stamp by which situations can be collected and sequenced into specified time intervals. A key observation is that once a situation is observed and added to a temporal sequence, it becomes immutable - in a sense similar to historical data. Indeed, this impiles that *no side effects* are allowed under the proposed domain specific language over situations. Once they are generated as function outputs, they are final.

In particular, this kind of immutability is well supported by functional paradigm where no update assignment is allowed.

1.3 A functional paradigm

Functional paradigm emphasizes computing with values rather than with actions. The computation is a direct, explicit description based on *what values* to use and to generate.

To illustrate the charm of the simplicity intrinsic to a functional paradigm, let us consider the following example using Haskell[2].

> *Problem: Find the summation of squares of natural numbers up to a particular number.*
>
> A typical imperative program might solve the problem with the following code:
>
> sumSq := 0;
>
> i := 0;
>
> while i ¡ n do

[2]A popular functional programming language

```
    i := i + 1;
    sumSq := i * i + sumSq;
end
```

The variable *sumSq*, which holds the summation value of the squares of natural numbers under consideration, is changed repeatedly during program execution, as is the *count* variable i. The effect of the program can *only* be seen by following the sequence of changes made to these variables by the commands in the program.

In contrast, the following is usually what a professional Haskell programmer will write to solve the very same problem:

$$sumSq :: Int \rightarrow Int$$
$$sumSq\ n = sum\ (map\ square\ [\ 1\ .\ .\ n\ ]\ )$$

In this program, a list is used to store numbers from 1 to n, then each of them is squared before being summed up to give the result. This Haskell code snippet uses neither control flow, found in imperative programs, nor recursion, and serves as a good example of a functional style program with elegant simplicity. Note that *square* is a Haskell function that another Haskell function *map* applies as its argument[3] to every member of a list before the *sum* function adding every number in the list to give the summation. The underlying *data-directed* programming style is added value of the functional paradigm. As compared with the meeting scheduler example under section 1.2, the functional skeleton structure demonstrated in the Haskell code snippet above, especially the *map* function, can be used immediately to implement the meeting scheduler design in (1.1) with a functional paradigm.

A program written in a functional programming language, such as the Haskell code snippet we saw earlier, can also be read as *executable specifications of behaviors*[9]. A domain expert, who oftentimes goes beyond simply being a programmer, would like to embrace an executable behavior specification language to edit, compile and test the design specification by actually

[3]therefore, map is called the higher order function.

running it on the source code level. In ensuing sections, we will showcase in detail how a *Situation* oriented domain expert can benefit from such a functional, *Situ* framework specific programming language.

Functional paradigm has already been used to serve various domain specific engineering purposes. For example, functional reactive programming (FRP) paradigm[10] in which continuously evolving behaviors and discrete events are used to model systems, has been adopted and implemented in applications as diverse as robotics, animation and real-time programming. As to concrete language, Fran[11] is a FRP domain-specific language with its root from Haskell for building reactive animations, using simple vector graphics, text, (animated) bitmaps and sound. In the area of multimedia reactive authoring research, FRP languages are also proposed to catch hold of varying values rendered by runtime animation tasks[12].

1.4 *Situ*^{f}-based Environment

As the following diagram shows, the central components within *Situ*^{f}-based environment revolve around serving the *Situ*^{f} program. We model the underlying capturing mechanism for context information centered around a situation by *SituIO*, which by nature is to stream captured context information. SituIO connects internally specified situations written in *Situ*^{f} language and the externally observed context variables bound with those situations. In *Situ*^{f}, each situation can be constructed, either independently or combinatorially, by four built-in functional patterns: *map, reduce, filter and apply.*

The precise meaning of SituIO is described using computational semantics in 3.6.3.

1.5 A glance view of the *Situ*^{f} environment

No computer program can run in a vacuum. Ever since Brian Kernigan and Rob Pike's classic *The Unix Programming Environment* hit the shelf, generations of programmers have been deeply aware that the secret of Unix's huge success is in a big part due to its unusually rich and productive programming environment. At its heart, it is the relationships among programs rather than the programs themselves that produce the core power of a system [13].

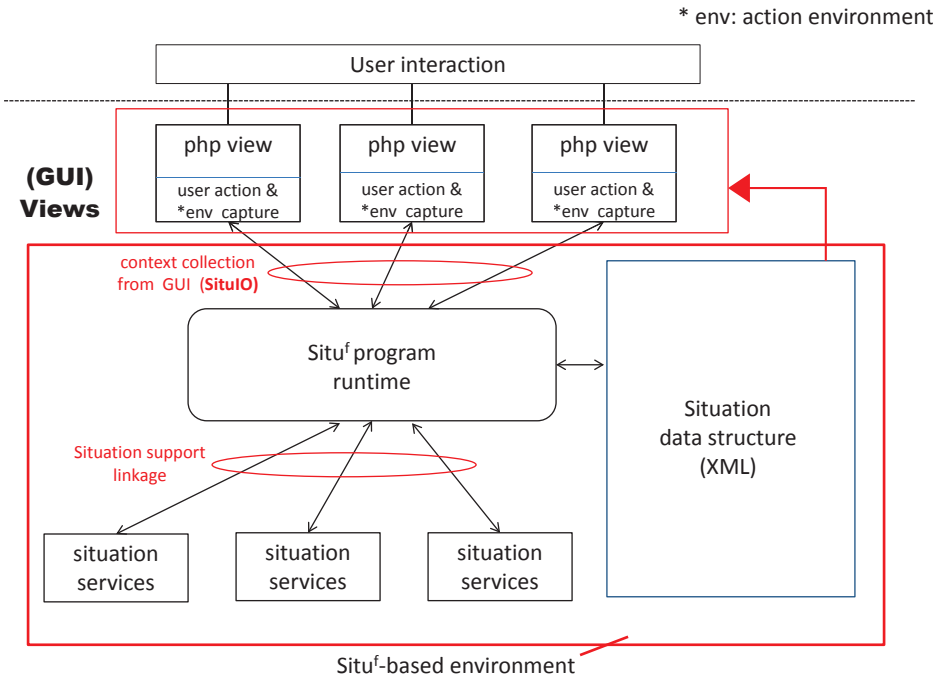
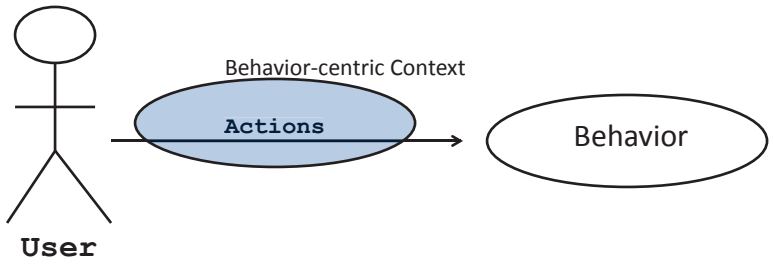


Figure 1.1 $Situ^f$-based environment: the overview

One of the design objectives of $Situ^f$-based environment is to provide such a setting in which domain experts can take advantage of a variety of elements centered around the abstraction of a human-centered situation [1].

Situ framework envisions and blueprints a rapid software evolution paradigm. $Situ^f$-environment targets the implementation of this vision by first focusing upon graphical user interface level transitions and commands. From situation services, situation data structures, to the underlying $Situ^f$ runtime, $Situ^f$-based environment is proposed to embrace all of the above as its core interest, contributing towards a situation-oriented, automated software adaptation paradigm. This design is composed to highlight the human-centric nature of situations under *Situ* framework. A user's context, especially, a user's behavioral context, falls into the category of *internal* dimension of context as opposed to external dimension of context. The latter has been more sufficiently addressed than the former in the context-awareness research community [14] . The reason, as least in part, is due to its being *external* and therefore easier to observe, for example location, temperature, time, lighting levels, proximity to other objects and so on. However,

to dig deeper into the context, especially to more accurately capture those regarding human factors like user's goals, tasks, work context, etc. . . , *internal* context needs to be attende d[14] equally well.



Behavior-centric Context

Figure 1.2 A simple diagram of the behavior-centric context for a user

The internal context, i.e., user's behavioral context, surrounds the actions of a user and it only comes into existence when a sequence of actions leading to a behavior are being performed. The internal context is created at the outset of an action sequence performed by the user and ends when the behavior is concluded.

An important observation is that behaviors vary in *scope* ranging from the very general to the very specific. General behaviors contain one or more specific activities. In this sense, we can think of a behavior as a *container* in which all the sub-behavioral activities at various levels compose a hierarchical structure. To give a concrete example of this context hierarchy, let us consider the login activity inside MyReview–a real world example featuing a web-based paper review system.

In general, a behavior, coupled with its internal context within which it exists, gives rise to a

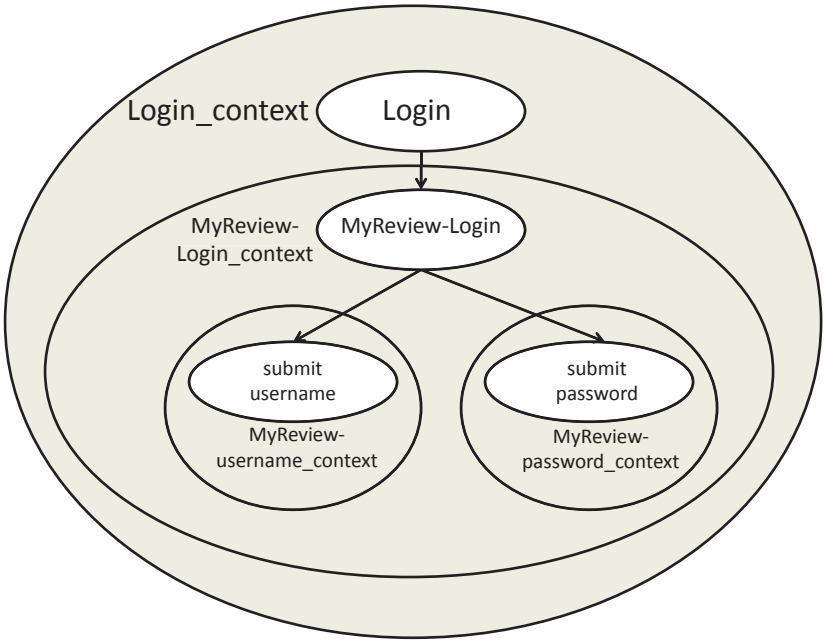


Figure 1.3 The cascading structure of the context for MyReview login

structural context diagram as shown in Figure 1.2. Note that *Login_context* influences not only login behavior, but also $MyReview - Login$, MyReview username and password submission behavior.

The picture is incomplete when we only look into the context surrounding the receiver of a user's action within MyReview without investigating the action provider's functional role. In other words, the user's profile also creates an important part of the internal context. Suppose that the user uses the MyReview system to organize a workshop under IEEE COMPSAC to meet academic researchers and industrial practitioners to discuss emerging methodologies and techniques to enhance software security. The context goes beyond interacting with the MyReview system, though they are closely related. In order to run a successful workshop, to attract and evaluate scholar paper submissions is indispensible. This activity cannot isolate itself from the context: a security workshop organizer usually is an expert in the area of computer security, which in turn is part of her career path: maybe she is a professor of computer security from computer science department in a University, currently working on several NSF-

funded research projects, one of which is to test the applicability and scalability of certain security theory as being extended to the industry partners. Organizing a workshop is an excellent task to support the project, for which MyReview is used as a tool. The user's profile is hierarchized through Figure 1.4:

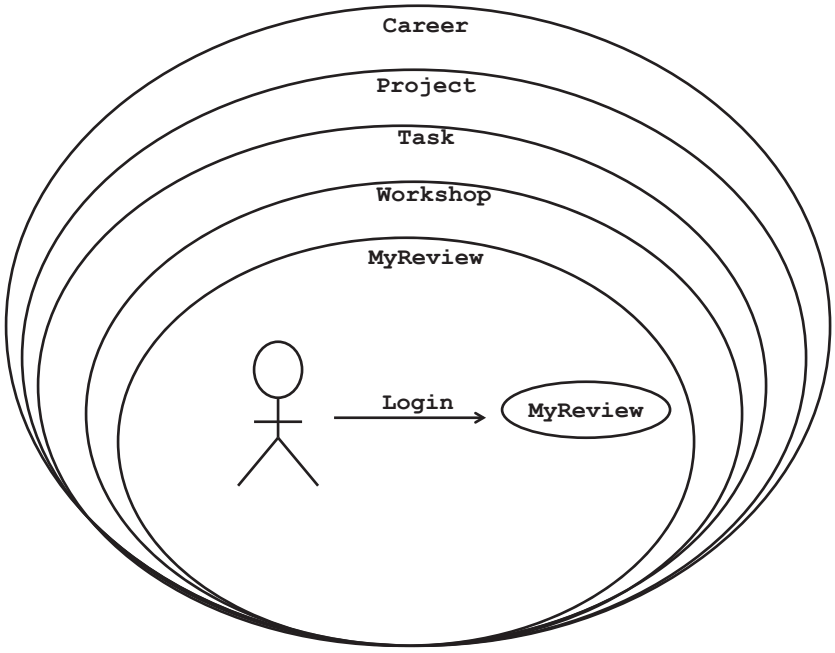


Figure 1.4 A more complete picture of the MyReview login context integrating user's profile

While a user's internal context can mirror her expectation, the challenge is to identify *a priori* the information that exists within the context, especially the internal context: how do you identify the information regarding the behaviors being performed even before the behavior is performed? A side question that comes along is how to exclude irrelevant information from contextual considerations in order to recognize only legitimate situations relative to a user's goal. *Situ*^{f} is designed to set up the environment that serves to answer the first question; we propose to answer the second question by the binding mechanism of context variables built inside the *Situ*^{f} language. More detailed discussion is offerd in the next chapter.

The ordering of contexts can be adjusted by events. Events are also signals sending off information about switching of context that often suggests transitioning between situations.

The mechanism of *events-passing* between contexts serves to link context-oriented situations. At any given time, only one *internal* context is active; as a result, a user's behavior will be recognized only when it is supported by the active internal context. The diagram in Figure 1.5 shows a concrete example of an active internal context.

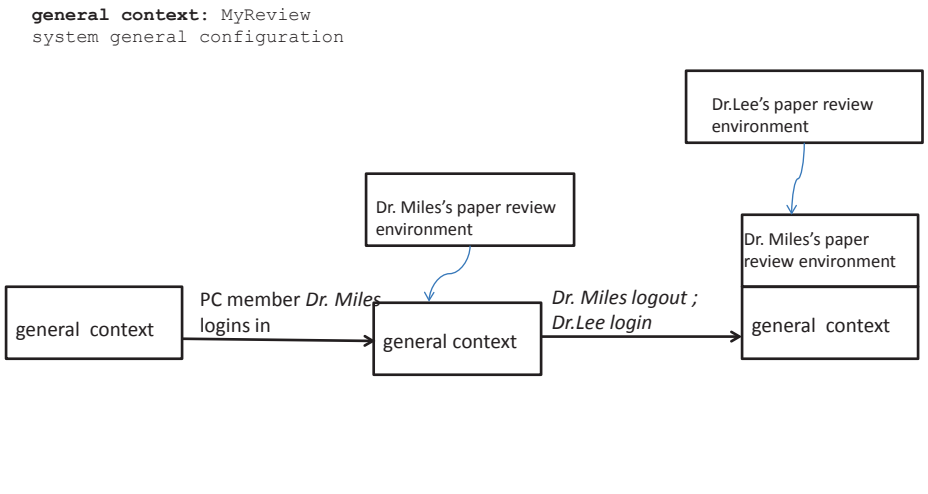


Figure 1.5 Context stack

Situ^f-environment supports context propogation: when the whole system is under a certain active context, all its sub-system will automatically inherit that active context. After a paper reviewer's successful login, for example, each GUI gadget such as buttons and links can be conceived as carrying the context of that specific reviewer's profile. This view reflects the human-centered characteristic of *Situ*^{f}-environment. When an event occurs in a certain context, all changed contextual elements will be published to all its sub-contexts. This is achieved combinatorially by *Situ*^{f}'s attribute grammar based context handling machinery and its interplay with *Situ*^{f}-based environment.

Situ framework is context-oriented [1]. Each situation contains information regarding its environmental context. In this work, an attribute grammar based context handling approach

is proposed to define the *Situ*^{f} language. The *Situ*^{f} language and the correlated environment based on *Situ*^{f} together bring home a situation programming model.

Existing approaches mostly remain focused on a user's external context level. In order to better address issues involving a user's cognitive activities, our approach will focus on capturing and using the context information surrounding the behavioral performance by a user. To this end, the designer of the *Situ*^{f} language pays particular attention to craft the language such that through built-in context binding, monad-based SituIO streaming and pattern-based situation constructing mechanism, situation specifications written in *Situ*^{f} intrinsically revolve around a user's behavioral performance. Having a behavioral-centric context implies a firm and consistent step forward towards the model of human-centered situation proposed by the original *Situ* framework.

The diagram in Figure 1.6 shows a typical usage scenario that engages a *Situ*^{f} program runtime. A domain engineer specifies situations in *Situ*^{f} code. Due to the high level situation-oriented perspective, the domain engineer, for different software modules that serves the end user, imports different context specifications to correctly bind contexts information in her *Situ*^{f} program. In the meantime, appropriate situation services are included to assist the real-time, context collection task.

1.5.1 A retargetable environment

Underlying software modules vary from domain to domain. The flexibility of allowing the *plug and play* of domain specific software modules into *Situ*^{f}-based environment makes it easily retargetable to other software modules whose situational interplay with an end user interests a domain engineer to write situation specification code in *Situ*^{f}. Besides, one domain expert's situation specification written in *Situ*^{f}, once made public under appropriate circumstances, can be imported to assist another domain expert's situation specification effort using *Situ*^{f}. Indeed, the central design objectives of the *Situ*^{f} language include promoting situation re-usability.

The contribution of *Situ*^{f} is that through language features and its built-in support, *Situ*^{f} allows domain experts to think and code in terms of **situations**. Lower level details regarding specific software modules through which an end user interacts with, as well as the specific

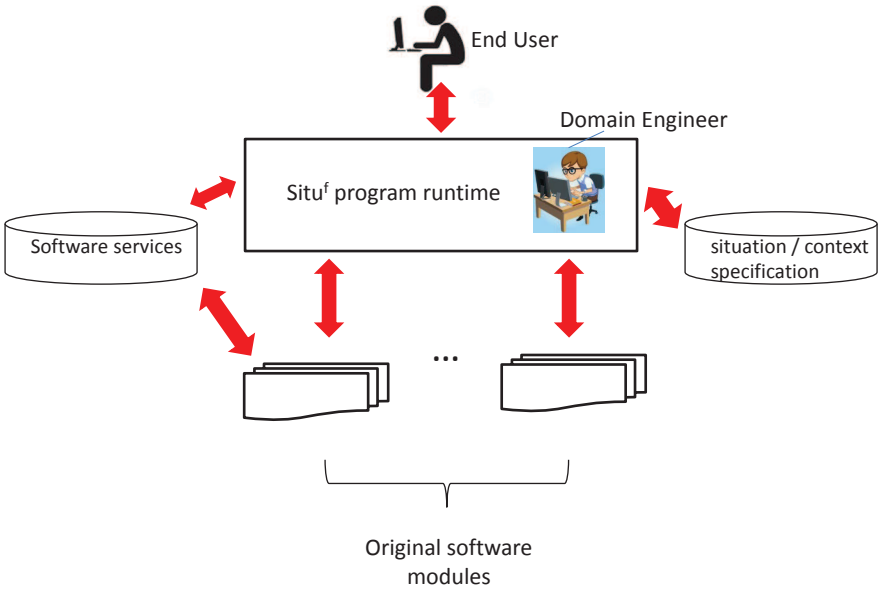


Figure 1.6 A *Situ*^{f}-based environment in action

services and context specifications are well encapsulated to promote the reusability of the code. Although currently only graphical user interface based user interaction is fully supported by the prototype, future extension can well cover the ground of remote sensory interaction and multi-modal interaction between a human user and a computing device.

1.6 My contribution

This work is the first attempt to realize the conceptual model of *Situ* framework [1]. The objective is to create a programming model with the following specific aspects:

1. bridging the concept of *Situation* over to realistic computing circumstances with clear software engineering realizations;

2. creating the *Situ*^{f}-based environment is created to drive the evolution of graphical user interface based commands and transitions - a subcategory of software evolution proposed by the original *Situ* framework;

3. utilizing a *Situation* as a basic building block into a functional paradigm specification language called *Situ*^{f};

4. developing an attribute grammar-based approach to formalize contexts surrounding a situation as *attributes*, with the purpose of rigorously specifying situations in *Situ*^{f};

5. linking situation data structures specified in *Situ*^{f} scripts to existing software by the language features and attribution rules built inside *Situ*^{f}'s attribute grammar;

6. carrying out experiments in MyReview and Java JFrame mechanism to showcase the feasibility of the *Situ*^{f}-based environment, which in turn provides evidence of the rigor and practicality of the original *Situ* framework.

1.7 Organization

This work is organized as follows: first *Situ*^{f}'s underlying data exchange model is introduced, built on top of an XML-based *intermediate representation*. Then, a series of examples are given revolving around the concepts of *Situation* contexts, especially action-oriented behavioral context and environmental context. After that, a rigorous definition of the domain specific, functional language *Situ*^{f} is given.Our focus centers around the attribute grammar model used in *Situ*^{f}, which combines the syntax and static semantics of the concept of *situation* under the grammar production rules, each one of which is decorated by a set of attribute equations. Our approach to model contexts as attributes in situations receives particular emphasis.

It is through a situation specification written in *Situ*^{f} that a *Situ*^{f}-based environment can be initiated, set up and finally established. Overall, the big picture that creates and runs such a *Situ*^{f}-based environment is entirely revolving around *situations*. Given that context data collection necessarily requires I/O support, SituIO, which finds its root in the *Situ*^{f} language itself, is emphasized and precisely described using computational semantics, a.k.a small-step operational semantics. Finally, an evaluation of the approach is given by a feasiblity test, followed by conclusion and future work.

CHAPTER 2. OVERVIEW

2.1 Background information on situation and human intention

Among the flurry of research on human intention cutting across Philosophy [2], cognitive science [15], and artificial intelligence [16, 3], two well referenced opinions that directly relate to the purpose that motivates this research stand out.

First, Bratman described intention as mental states motivating actions [2]. His opinion has been adopted and turned into the supporting theory, known as BDI logic, for agent planning research. Targeting Rational agents, a lot of research work has been done in the area of Artificial Intelligence (AI) and Robotics, to infer the mental states of agents. Approaches involving planning theory [17, 18], ontologies [19, 20], closed world mathematical logic such as Kripke semantics [21, 22] are well developed. However, this is not enough for inferencing **human** intentions. The reasons are found both due to the efficiency issues and in terms of the practical concerns under current state of art. The essential challenge comes from the highly fluid and intangible nature of human's mental states. To see the gap more clearly, let us use epistemic formulas from Kripke semantics[1] [23] as an example.

The key construct under Kripke semantics is what is so called *Kripke structure* $\mathbb{M}$, defined as a tuple ¡ S, π, R_1, ... R_m ¿ where:

(i) S is a non-empty set of states,

(ii) $\pi : S \rightarrow$ (P $\rightarrow$ { $\mathbb{t}$, $\mathbb{f}$}) is the truth assignment to propositional atoms per state,

(iii) $R_i \subseteq S \times S$ ($i = 1, \cdots, m$) *are* so-called *accessibility relation.*

[1] also known as *possible world semantics.*

As is shown above, the set S of states has to be determined **before** one can start reasoning on the truth value of an *Epistemic Formula* [23]. The Epistemic Formula generally takes the following form[2]:

$$(\mathbb{M},\ s)\ \vDash\ K_i\phi\ \Leftrightarrow\ (\mathbb{M},\ t)\ \vDash\ \phi\ for\ all\ t\ with\ (s,t) \in R_i$$

Despite its logic form, $K_i\phi$ expresses the meaning of "Agent i acknowledges ϕ". The underlying implication is that in order to "understand" a certain epistemological state of agent i's knowledge ϕ in world $(\mathbb{M},\ s)$, it is sufficient and necessary to "understand" if that knowledge ϕ still holds in all worlds agent i considers possible. Note that $(\mathbb{M},\ t)$ can be any of such a possible world relative to a given state s due to state t's "*for all*" condition. Two key issues stand out when applying the Kripke semantics to human-centered domain:

- If set S can not be solidly decided, or can never be easily stabilized mostly due to the human factors involved, the above epistemic reasoning can be seriously hampered;
- Even if set S is decidable but it is very large a set, as is usually the case even for a standalone computer program [24], not to say the case where state changes are made by human beings' instantaneous decision, the efficiency of reasoning under Kripke semantics-based logic system can be quite a daunting task [25, 23]. As shown from the diagram below, to verify agent i's knowledge ϕ at state s, all possible worlds have to be checked. Those possible worlds are derived from all accessible states sanctioned by agent i's accessbility relation R_i.

These restrictions generally apply as long as Kripke semantics serves as a key component in the underlying theoretical foundation, which is at the time of this writing prevalently true in AI for knowledge representation, practical knowledge reasoning [25, 23], as well as ontologies such as those based on description logics [19, 20]. More concrete examples include the well-known LORA [26] system, a derivative and further extension from Rao and Georgeff's original BDI logic system that allows the representation and reasoning about beliefs, desires, intentions, and actions of agents within a system, and how these beliefs, desires, intentions and actions change over time [26, 27].

[2]for the sake of brevity, we omit the purly propositional counterparts.

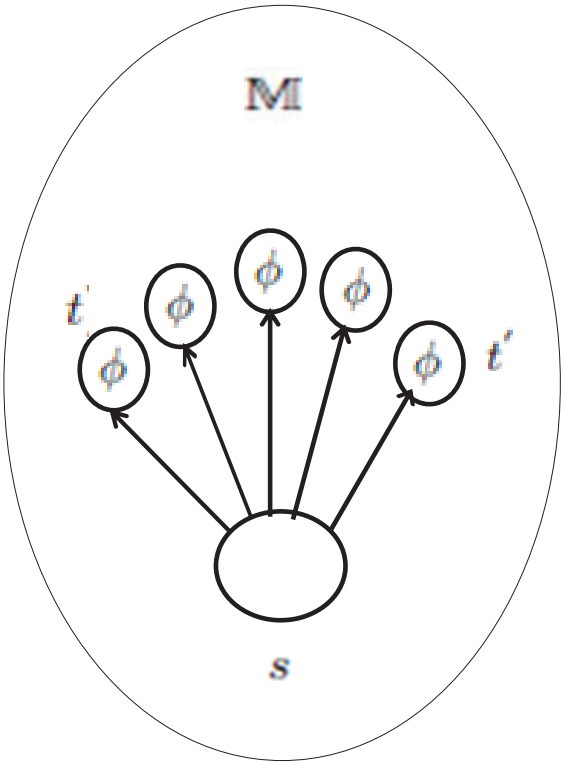


Figure 2.1 A Kripke Structure

In response to Bratman's theory, Scheer initiated the second school of opinion regarding the definition of human intention. He pointed out that intention should be considered as a course of actions [28]. In Scheer's opinion, direct mental states modeling should be avoided by way of capturing action sequence.

Both Bratman and Scheer seem to agree on the point where human intention can be analysed through the observation of action sequences. In addition, sensor-based approaches [29, 30] have recently gained momentum, through which collected information on temporal contexts, closely related user-centric data[3], can be timely captured and analysed to improve human life style [31]. It's worth mentioning that *XML-based* sensor language helped to seamlessly move sensor data onto the interconnected World Wide Web, to the platform of mobile computing [29] and even to serve the construction of human-robot interfaces [32].

[3]Examples include user's geographical locations, etc.

We use XML to describe the structure of sensory data collected and the structure of situations drawn from those data. XML serves as the intermediate form of *Situ*^{f} program and prepares the discussion for further analysis of situation modeling that motivates the invention of *Situ*^{f} domain specific language.

The XML-encoded Data (or documents) are ordered, labeled tree structures. Among others, XML's intrinsic hierachical structure offers enormous flexibility and in the meantime has nurtured a flurry of research in computer science on XML itself, ranging from XML type system analysis based on DTD or XML schema to XML-based document processing techniques; Good examples include statically typed XML prcessing devices such as XDuce [33] and RELAX NG [34]. By design, *Situ*^{f} adopts XML to serve as an intermediate meta-language to capture and represent contextual information attached to each situation. Underpinning its wide-spread applications on sensory data is XML's semi-structured data model, which is enbodied by the form of a mark-up language.

XML can also comfortably serve the purposes of describing the sensory data and of depicting the structure of situations drawn from these data. The intuitive and syntactic nature of XML serves to prepare the discussion of further analysis and modeling of situations under *Situ* which nutures the birth of *Situ*^{f} domain specific language and *Situ*^{f}-based environment.

To strike a brand new paradigm featuring rapid and automated software evolution, *Situ* framework [1] developed the concept of minimal intention that is built from a sequence of situations[4] with respect to a goal[5]. We take minimal intention as our default definition of intention to avoid terminology confusion. Each situation snapshots software user's behavioral and environmental contexts as well as the predicted user's desire based on those contexts.

As part of the big picture, rather than exposing the hierarchical context directly to the domain experts and engineers, we propose that contexts be captured and internally processed inside *Situ*^{f}-based environment, which is set up by *Situ*^{f} program written by domain experts. More concretely, we abstract the key processig power of *Situ*^{f}-based environment as an abstract machine, whose native language is a situation stream language as will be explained later on.

[4] where each situation is defined as a triple $\{\mathrm{d},\mathbb{A},\mathbb{E}\}_t$, it is human-centric since $\mathbb{A}$ and $\mathbb{E}$ are human factors; see [1] for details.

[5] referring to system goals as discussed in Goal-Oriented Requirement Engineering [6, 7]

This abstract machine is noted as *Situ*^{f}-AM. *Situ*^{f}-AM sets up and maintains its internal states, geared by captured contexts.

2.2 A motivating example

Let us first consider a concrete example.

MyReview, a paper review system in use for conference organization has three types of users: paper author, paper reviewer and conference organizer. Each author is given login access to her paper once the initial submission has been completed to the system so that she can keep updating her submission until the deadline is hit. Paper reviewers can login to review those papers assigned by conference organizers following a double blind review policy, and conference organizers once login, can utilize the administration tools such as assign papers to reviewer, batch email to all program committee members etc. A typical scenario of interest is the login situation.

Figure 2.2 A MyReview Example

Our discussion is based on the assumption that a user's computer has been equipped with a sensory touch detector, basically a situation service shown in Figure 1.1, that can record all the mouse clicks and key strokes over button, links and textboxes etc. In fact, this is the basic setup *Situ*^{f}-based environment requires in order for it to fulfill its mission. Our *Situ*^{f} prototype provides this capability as a default service.

Given that in MyReview, the login interface for users (author, paper reviewer and conference organizer) may seem to provide identical visual effect, a less experienced paper reviewer accidentally hits the conference organizer's login entrance. *She types in her username and password*(S_1) (see footnote [6]), *clicked the login button*(S_2). The following picture visualizes these two situations with regard to the login interface:

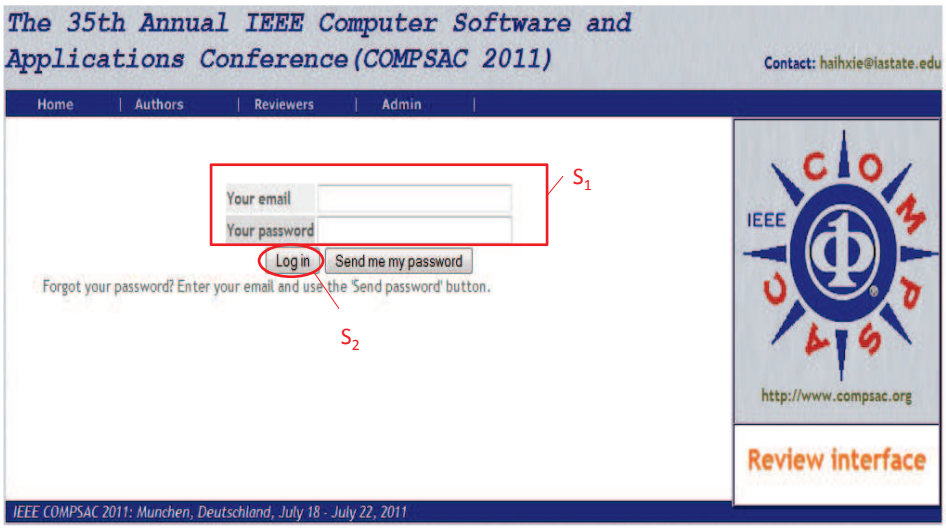


Figure 2.3 Situations S_1 and S_2

The minute the login button is clicked after wrong username and password information is typed in, the user sees a login fail page saying "invalid password!" which signals her to relogin (S_3).

Each situation such as S_1 is intrinsically timestamped, such as $S_1^{t_1}$. Several times around, the user eventually gets to login successfully; then she *clicked and downloaded one of the four papers assigned to her and started reviewing*($S_4^{t_4}$), *uploaded her comment and review score into the system*($S_5^{t_5}$). Following a similar vein, she moves on and reviews the next paper ...

[6] S_1 corresponds to a node in XML format, which is used as the intermediate representation to encode captured situation sequences. Same thing for S_2,S_3,S_4, ...

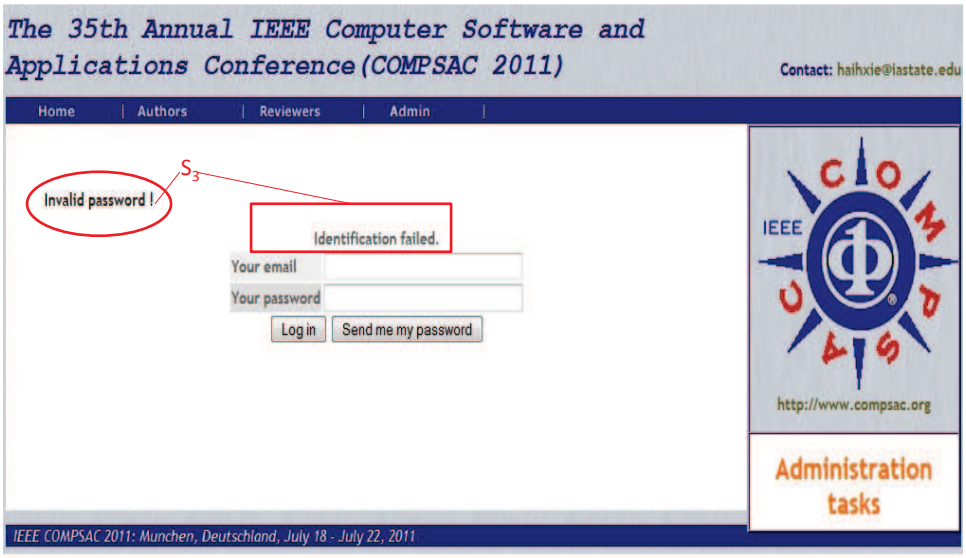
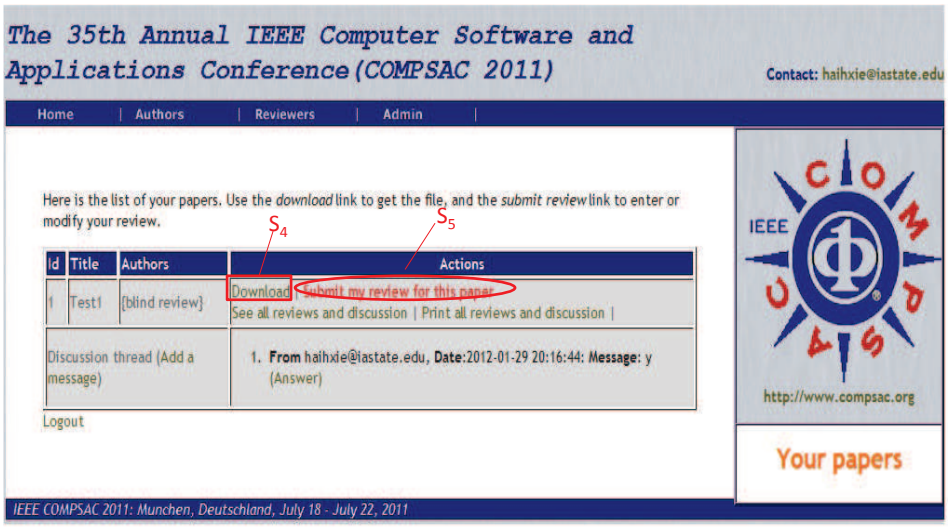


Figure 2.4 Login Fail Situation S_3

For concrete syntax offered by *Situ*^{f} that a domain engineer can use to write code to create the *Situ*^{f} environment, please refer to Chapter 3. We still use XML to explain the machinery, given that it is a *Situ*^{f} syntax neutral, intermediate representation. This arrangement strives to emphasize on the semantic transformation critical for any computing environment, including *Situ*^{f}-based environment.

Next, we assume that a domain engineer's program written in *Situ*^{f} is already successfuly interpreted. This assumes the completion of the of proper establishment of a *Situ*^{f}-based environment so as to effectively capture the runtime situation sequence centered around a use case scenario. The context mediates the entire process. The aforementioned MyReview system example is such a concrete case in point.

The particular situation sequence describing the above scenario, after being captured by the mechanisms employed under *Situ*^{f} environment, is sequentially represented as follows:



$$S_1^{t11}, S_2^{t21}, S_3^{t31}, S_1^{t12}, S_2^{t22}, S_3^{t32}, S_1^{t13}, S_2^{t23}, S_3^{t33}, \ldots, S_1^{t1n}, S_2^{t2n}, S_4^{t41}, S_5^{t51} \tag{2.1}$$

Note that in this situation sequence, since the last login action given by the paper reviewer must be a successful login action, there is no S_3^{t3n} to represent a *Login Fail Situation* to follow S_2^{t2n} in the sequence. By nature, a *Login Fail Situation* corresponds to an event which is passed back to the $Situ^f$-based environment, which is formalized as $Situ^f$ virtural machine. This event is analogous to an IO interrupt on a real machine, whereas under $Situ^f$-based environment the communication is between user's action imposed on a specific software system (such as MyReview's graphical user login for paper review interface just demonstrated) and the external environment, generally imagined as the $Sifu^f$ virtual machine. The handling of this event inside $Situ^f$ virtual machine will change the state of the virtual machine correspondingly.

The XML intermediate representation reflecting the capturing of temporal situation sequence is as follows:

$<S_1$ timestamp="t_{11}" $>$

<action target=“login_text_box” src=“login.php “ >

<output>

<context target=“text_box1“> text1 </conext>

<context target=“text_box2“> text2 </context>

</output>

</action>

</S_1>

<S_2 timestamp=“t_{21}“ >

<action target=“login_button” src=“login.php “ >

<input>

<context target=“text_box1“> text1 </context>

<context target=“text_box2“> text2 </context>

</input>

<output>

<context> username </context>

<context> password </context>

</output>

</action>

</S_2>

<S_3 timestamp=“t_{31}“ >

<effect> Fail </effect>

<source> S_2 </source>

</S_3>

<S_1 timestamp=“t_{12}“ >

<action target=“login_text_box” src=“login.php “ >

<output>

<context target=“text_box1“> text1 </conext>

<context target=“text_box2“> text2 </context>

</output>

</action>

</S_1>

<S_2 timestamp="t_{22}" >

<action target="login_button" src="login.php " >

<input>

<context target="text_box1"> text1 </context>

<context target= "text_box2" /> text2 </context>

</input>

<output>

<context> username </context>

<context> password </context>

</output>

</action>

</S_2>

<S_4 timestamp="t_{41}" depend_on="S_1" >

<action target="paperd_download_button" src="review.php " >

<input>

<context> username </conext>

<context> password </context>

</input>

<output>

<context> paper_S_4 </context>

</output>

</action>

</S_4>

<S_5 timestamp="t_{51}" depend_on="S_4" >

<action target="review_upload_button" src="review.php " >

<input>

<context> paper_S_4 </conext>

</input>

</action>

</S_5>

We define an event to be a special situation that semantically splits a sequence of situations into sub-sequences. It is the boundary that delimits scopes of contexts. An event is closely related to the immediate goal of a sub-sequence of situations. In the example of $LoginFailEvent$, the goal of the sub-sequence of situations up until S_3 is the negation of the event message, that is, to login successfuly.

Event passing inside $Situ^f$ virtual machine suggests the following pattern:

$$((S_1, S_2, S_3)^*(S_1, S_2))(S_4, S_5) \tag{2.2}$$

To see that, the repeated situation sequence is (S_1, S_2, S_3): with the event S_3 trailing the sequence; This creates $(S_1, S_2, S_3)^*$. The rest is non-repeated situation sequence excluding S_3, (S_1, S_3) since no exceptional situations, or events, occur there.

This pattern is generated over captured situation sequence (2.1). Prototypical forms of situations are used in pattern (2.2),where contextual information is taken off. For example $S_1^{t_k}$ and $S_1^{t_{k+1}}$ both have the same **prototypical form** S_1, which stands for generic login situation - let alone certain contexual differences such as {username,password}. Two failed logins, between one and the other, must have different context variables such as their time stamp, input texts of username and password by the user etc. $S_1^{t_k}$ and $S_1^{t_{k+1}}$ represent two concrete, context-annotated logins at time instant t_k and t_{k+1} respectively. Indeed, we can view t_k, the temporal tag decorating a prototypically formed situation S_1, as a symbolic annotator implying all associated contexts for S_1 at time instant t_k.

2.3 The Environment Model of *Situ*^{f} language

In *Situ*^{f} language, variables containing context information are stored in places called *locations*. The set of locations is noted as **Loc**. Let l denote an arbitrary location of **Loc**. Given that a machine all has countably many storage locations, we assume that **Loc** = $\mathbb{N}$, meaning locations are natural numbers.

The environment for *Situ*^{f} context variables is a function that maps each context variable to a storage location. We can imagine a variable environment as a symbol table. More formally, the set of context variable environments is the set of partical functions from context variable to *locations*:

$$EnvV = Var \ \cup \ \{next\} \ \rightharpoonup Loc$$

We use env_V to denote an arbitrary member of $EnvV$. The $\rightharpoonup$ represents a partial function. Moreover, we model the allocation of memory location for a new variable by assuming the existence of a function:

$$new : Loc \rightarrow Loc$$

the *new* function above returns a successor for each location. That is done whether this successor location is available or not.

Since we are assuming that $Loc = \mathbb{N}$, we can think of it in our settings as:

$$new \ l = l + 1$$

The special context variable *next* is used to point to the next available location to be assigned to a variable.

$$next = new \ l$$

given that current variable location in our natural number modeling proceeds to l. The next diagram provides a more intuitive illustration.

The following introduces a notation for the introduction of a new variable, which when bound to a location will produce a new environment. Suppose that the old environment is

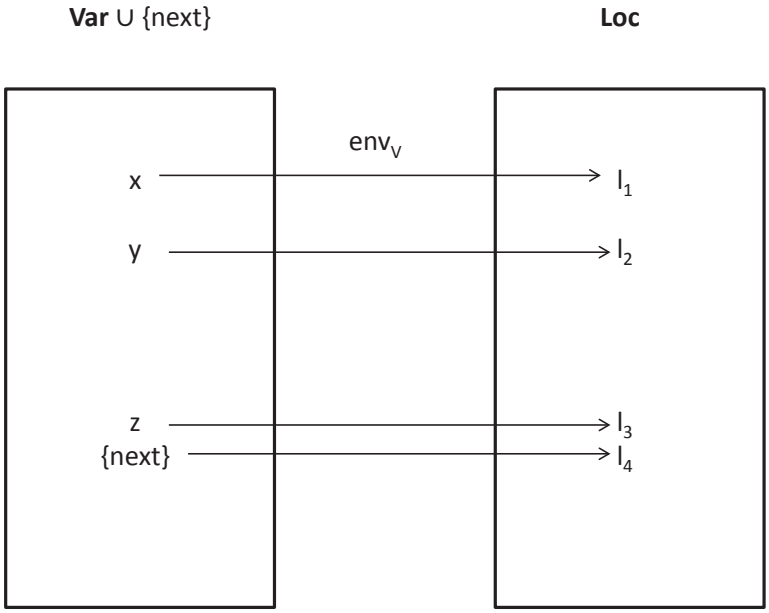


Figure 2.5 Example of context variable environment

env_V, now a new context variable x is bound to location l, i.e. $env_V[x \mapsto l]$; then the new environment $env_V^{'}$ is:

$$env_V^{'}y = \begin{cases} env_V y & \text{if } y \neq x \\ l & \text{if } y = x \text{ and x is unbound in } env_V \\ error & \text{if } y = x \text{ and x is already bound in } env_V \end{cases}$$

The following figure shows that we have three context variables x, y and z. env_V is the function noted by the arrows between **Var** $\cup\{next\}$ and **Loc**. It shows clearly that x is bound to location l_1, y is bound to location l_2 and z is bound to location l_3. The next free location is l_4. No other context variables are bound to any locations.

Note that since $Situ^f$ programs are computing in functional paradigm and no update assignment is allowed. Every context variable will be assigned one exclusive location and each

location will not change its value once defined. Therefore, in our environment model each location is one-to-one corresponding to a value. This provides notational convenience since we only need to keep track of the binding location of a context variable to understand its semantics.

The set of stores is also defined as a set of partial functions from locations to values. Given that all values in the domain of *Situ*^{f} applications are Graphical User Interface based transitions and commands, the value of a context variable can be encoded as a string. Each string is a linear combination of characters($\mathbb{C}$), each *Store* function is a mapping from set of locations to $\mathbb{C}^*$. Therefore the set of *Store* functions is represented as follows:

$$Store = Loc \rightharpoonup \mathbb{P}(\mathbb{C}^*)$$

in which, $\mathbb{P}$ refers to power set.

Situ^f, a functional language that is formally defined in the next chapter, strives to provide necessary means to allow a domain expert to set up the *Situ*^{f}-based environment complying with the original *Situ* model [1]. In this dissertation, we focus on Graphical User Interface based commands and transitions which appears to be typical in present web-based applications. Domain expert's vision and expertise injected into the environment through *Situ*^{f} program largely determines the effectiveness of the environment. Moreover, context variables under *Situ*^{f} environment, whose close correlation with situations are proposed through an attribute grammar based approach, which allows one to look at situations as context-oriented structures [1].

XML's semi-structure feature and its wide application across multiple domains, especially in the realm of sensory data representation to support many pervasive computing purposes, makes it an excellent choice to serve as an intermediate form to construct and represent situations under *Situ*^{f}-based environment. XML is used in this work to capture and illustrate the internal workings of the underlying *Situ*^{f}-based environment. Furthermore, we model *Situ*^{f}-based environment as a virtual machine, designed to handle situation flows and context variables from a computer programming point of view.

This section provides a foundation for the efforts later on to formally define the semantics of *Situ*^{f} language, especially in 3.6.2.

2.4 Context variables under the environment model

Let us again consider the paper review system example from 2.1. Due to unfamilarity with the system, the paper reviewer might have accidentally run into *unrelated situations* such as clicking a link and being transferred to paper author's paper submission page, rather than paper reviewer's review submission interface. This action leads to an erroneous situation since the username and password required to enter authors's paper submission page is different from the current username and password that are taking effect, namely the one that records a paper reviewer's identity. An event (S_7) is resulted from this error and will be passed.

$$(S_1, S_2, S_3)^*(S_1, S_2)(S_6, S_7) \tag{2.3}$$

The XML intermediate representation for Situation sequence (2.3) is as follows:

<S_1 timestamp="t_{11}" >
<action target="login_text_box" src="login.php " >
<output>
<context target="text_box1" > text1 </conext>
<context target="text_box2"> text2 </context>
<output>
</action>
</S_1>
< S_2 timestamp="t_{21}" >
<action target="login_button" src="login.php " >
<input>
<context target="text_box1"> text1 </context>
<context target="text_box2"> text2 </context>
</input>
<output>
<context> username </context>
<context> password </context>

</output>

</action>

</S_2>

<S_3 timestamp="t_{31}" >

<effect> Fail </effect>

<source> S_2 </source>

</S_3>

<S_1 timestamp="t_{12}" >

<action target="login_text_box" src="login.php " >

<output>

<context target="text_box1" > text1 </conext>

<context target="text_box2" > text2 </context>

</output>

</action>

</S_1>

<S_2 timestamp="t_{22}" >

<action target="login_button" src="login.php " >

<input>

<context target="text_box1"> text1 </context>

<context target= "text_box2"> text2 </context>

</input>

<output>

<context> username </context>

<context> password </context>

</output>

</action>

</S_2>

<S_6 timestamp="t_{61}" >

<action target="paper_submit_button" >

<input>

<context> username </conext>

< *context* > password </context>

</input>

<output redirect="submitpaper.php " />

</action>

</S_6>

<S_7 timestamp="t_{71}">

<effect> Fail </effect>

<source> S_6 </source>

</S_7>

Each event is directly from a human action imposed on the software system; an immediate system goal exists with regard to the action [1], and the occurence of an event from within $Situ^f$ environment is closely linked to a user's desire.

Indeed, event is an appropriate mechanism to realize the interaction between the user (feedback) and the software system to better understand the user's instantaneous desire. A good question to ask in the case of situation sequence (2.3) is: does the user desire to submit a paper, or does she/he simply commits an operational mistake? The latter implies that the user is still committed to her/his original desire: to review paper. The $Situ^f$-based environment will inject action around events to more accurately capture the user's desire.

The passing of event (S_7) works as an interrupt between the user and the $Situ^f$ virtual machine. It interrupts the output generation of (S_7), namely successful redirection to the paper submission page for paper authors. The internal working of this event is based on the *environment model* upon which the virtual machine is built. Let us go into certain length of detail of how the *environment model* employed by $Situ^f$ works to facilitate its event passing machinery.

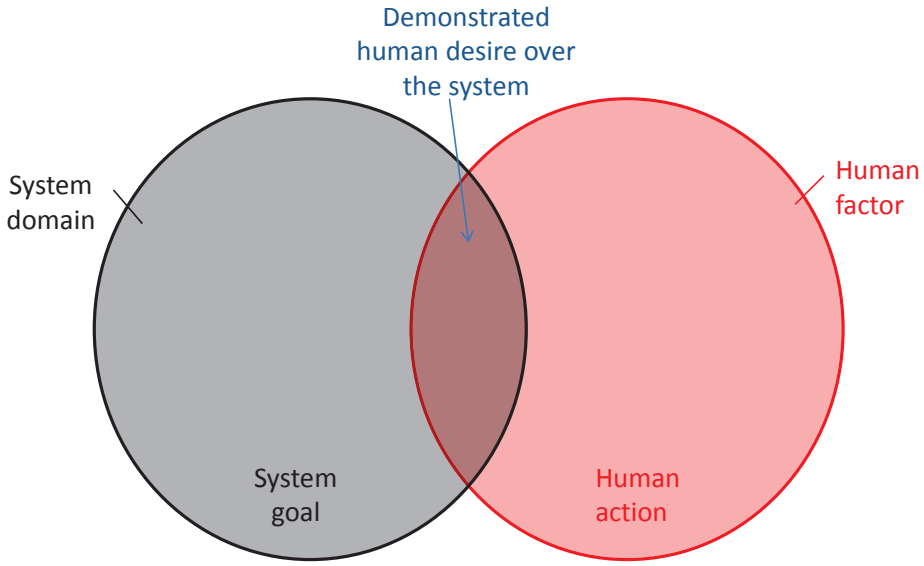


Figure 2.6 System user's demonstrated desire

2.5 Event passing under *Situ*^{f}'s environment model

Under *Situ*^{f}'s environment model, there are three kinds of errors detected by the environment that can trigger an event.

- A runtime error raised by the software system
- A tendency to use undefined (unbound) context variables or functions
- A tendency to conduct update assignment to a variable location already bound to a context variable

Let us take a look at the events raised in the MyReview system example again, especially the situation sequence at (2.3):

$$(S_1, S_2, S_3)^*(S_1, S_2)(S_6, S_7)$$

The XML intermediate representation for the above situation sequence is:

<S_1 timestamp="t_{11}" >

<action target="login_text_box" src="login.php " >

<output>

<context target="text_box1" > text1 </conext>

<context target="text_box2"> text2 </context>

<output>

</action>

< /S_1>

<$S_4$2 timestamp="t_{21}" >

<action target="login_button" src="login.php " >

<input>

<context target="text_box1"> text1 </context>

<context target="text_box2"> text2 </context>

</input>

> <output>

<context> username </context>

<context> password </context>

</output>

</action>

</S_2>

<S_3 timestamp="t_{31}" >

<effect> Fail </effect>

<source> S_2 </source>

</S_3>

<S_1 timestamp="t_{12}" >

<action target="login_text_box" src="login.php " >

<output>

<context target="text_box1" > text1 </conext>

<context target="text_box2" > text2 </context>

</output>

</action>

</S_1>

<S_2 timestamp="t_{22}" >

<action target="login_button" src="login.php " >

<input>

<context target="text_box1"> text1 </context>

<context target= "text_box2"> text2 </context>

</input>

<output>

<context> username </context>

<context> password </context>

</output>

</action>

</S_2>

<S_6 timestamp="t_{61}" >

<action target="paper_submit_button" >

<input>

<context> username </conext>

<context> password </context>

</input>

<output redirect="submitpaper.php " />

</action>

</S_6>

<S_7 timestamp="t_{71}">

<effect> Fail </effect>

<source> S_6 </source>

</S_7>

The two events are situations S_3 and S_7. Their happening is due to the actions occured in their immediate previous situations - S_2 and S_6 respectively. For event S_7, the user's action to click on the paper submission button triggers MyReview system to internally check the cached username and password. The username and password are modeled as context variables by $Situ^f$'s environment model.

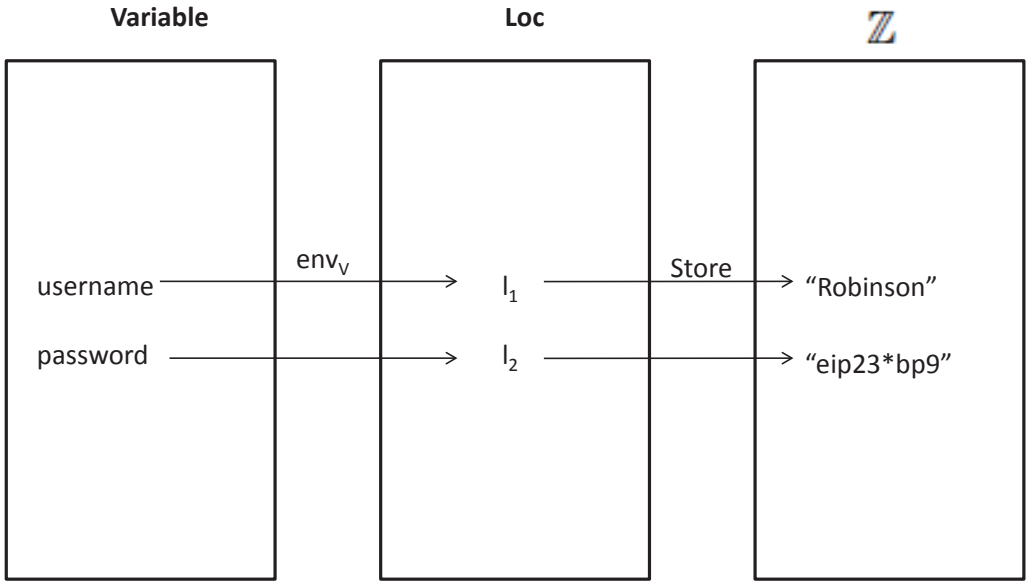


Figure 2.7 Environment model: a working example

Applying the env_V function to *username* and *password* gives the memory locations in *Situ*^{f} virtual machine for *username* and *password* to store the data; afterwards, applying *Storage* function on those two locations returns the current value of *username* and *password*. In other words, the value v of a context variable x can be found by $v = store \circ env_V(x)$, where $\circ$ stands for functional composition.

MyReview system looks these values up in its internally maintained credentials, and delivers a rejection error afterwards. This is a runtime error. An important point to notice is that the context variable of *username* and *password* are established into the environment by the last S_2, since the previous one was interrupted by an event, that therefore led to an abortion of the desired context variables.

Note the difference between the MyReview system and *Situ*^{f}-based environment. The environment model is employed by *Situ*^{f}-based environment, not MyReview system. The latter happens to be an example under discussion, whereas *Situ*^{f}-based environment can be set on top of different Graphical User Interface based software system. To tailor the *Situ*^{f}-based environment to a specific Graphical User Interface based software system, a domain expert needs to use *Situ*^{f} language, which will be introduced with a concrete example in the next section.

By comparing (2.3) with (2.2), it is clear that these two patterns are not the same, and even less, neither is compatible with the other. The question is: is event S_7 an accident, or is it the user's real intention? By writing *Situ*^{f} script, a domain expert can choose either interactive mode or default mode to resolve this issue. By default mode, the *Situ*^{f} virtual machine will reason based on its contextual information.

Under default mode, with respect to a goal, for example, (S_4, S_5), (2.3) can be thought of as compatible with (2.2), therefore S_7 is an accident. The reason is that S_6 is a noise with regard to (S_4, S_5) - since none of S_6's context data is found in S_4, S_5 - *Situ*^{f} virtual machine internally replaces S_6 with ϵ, a *least* situation, which is a constant situation compatible with any situation type (including situation sequence rendered meta-situation, like (S_1, S_2)), so that (S_6, S_7) will be subsumed into $(S_1, S_2, S_3)^*$, which reduces (2.3) to $(S_1, S_2, S_3)^+(S_1, S_2)(S_4, S_5)$.

After being equipped with the above example that provides a concrete "insider's" view of *Situ*^{f}'s intermediate meta-level to expose how situations are represented and analysed inside *Situ*^{f}-based environment through contextual information, let us now take the domain engineer's[7] role to see how to write a script in *Situ*^{f} domain specific language to set up the *Situ*^{f} environment in order to capture those situations via relevant contexts. Before that, an emphasis on understanding of situation from a *Situ*'s point of view as a human centric construct is necessary [1].

2.6 Human-centric Situations

A critical new ingredient injected into the concept of *situation*, around which the entire *Situ* framework [1] is built, is that all situations are *human-centric* situations. *Situ*'s perspective on situation strikes a brand new vision in which a human's dimension is added as an indispensable component.

In the last section, the paper review system example was discussed. From this section on, while still using the same example, our emphasis will be switched to the machinery a *Situ*^{f}-based environment offers to facilitate the capturing of user's information which is eventually built into a situation, thus the name human-centric situation.

In addition to providing some intuitive background for those interested in exercising situation programming in *Situ*^{f}, this example also serves the following purposes:

- it elaborates the concept of *behaviorial context* and how it relates to *situations*;
- it elaborates the concept of *Situ-environment* and how it integrates *Situation* and *a real world system*;
- it introduces *Situ*^{f}'s built-in support for situation composition patterns that a domain expert can benefit from.

[7] In this writing, we use domain engineer and domain expert interchangeably.

2.7 An introduction to *Situ*^{f} language and examples

Situ^f is a functional specification language. Its central langauge craft revolves around the idea of a function, from a function name, its inputs (arguments), outputs to more sophistecated techniques like functional composition, partial function, currying, etc... In *Situ*^{f}, a name is either a function or static data. *Situ*^{f}'s function models an action or compound action, representing a behavioral context within a situation [1]. The real novelty is the way that *Situ*^{f} is proposed, which combines functional paradigm with attribute grammar to model situations for domain specific purposes within the boundary of *Situ* framework.

2.7.1 Attribute-Grammar model of *Situ*^{f}

Attibute grammar [35] can be conceived as context-free grammar with an additon of attached context-sensitive conditions and semantics-oriented attribute rules. More precisely, it extends the context-free grammar by attaching attributes to the nonterminal symbols of the grammar and by supplying attribute equations to define attribute values. Each production in attribute grammar has a set of associated attribute rules known as attribute equations to specify the relationships between the attributes of terminals and nonterminals in the production. For the following production p:

$$p : X_0 \rightarrow X_1 \dots X_k$$

each $X_i, (0 \leq k)$ denotes an occurence of a grammar symbol, and associated with each nonterminal occurence is a set of attribute occurrences, denoted as $A(X_i)$ which includes all nonterminal's attributes.

Each production in an attribute grammar usually has a set of equations, each of which defines the attribute values. In essence, those equations are indeed tantamount to *attribute-definition functions*. The attributes of a nonterminal are divided into two disjoint classes: *synthesized* attributes, denoted $S(X_i)$ and *inherited* attributes $I(X_i)$, where $A(X_i) = S(X_i) \cup I(X_i)$. Briefly, *synthesized* attributes are used to pass information **up** a syntax tree; in contrast, *inherited* attributes are used to pass information **down** a syntax tree. In particular:

- Terminals may have only *synthesized* attributes;
- Nonterminals may have both *sythesized* and *inherited* attributes.

Figure 2.8 in the following section illustrates these two important terms.

2.7.2 Synthesized attributes, inherited attriutes and functional dependency

Synthesized attributes and inherited attributes are two key components for an attribute grammar to propagate attribute values through its derivation tree. Moreover, a dependency graph further enhances a derivation tree by adding functional dependency relations among attribute occurences to visualize the direction of the propagation flow of the attribute values for the attribute grammar. A handy side effect coming out of the dependency graph is that it serves as a convenient tool to allow intuitive judgement upon circular versus non-circular attribute grammars, without the need of stepping into full length formal proof. This section brings together these concepts and their closely related formalisms, such as *Function Dependency* (FD), *Dependency Graph*, etc. . . to facilitate further discussion.

As a running example, we use a simple programming language called *SimpleL*, which does not have type expressions in variable declarations. The only statement the language supports is the assignment statement. The context-free grammar for *SimpleL* is defined in Table 2.1. For brevity, we do not show productions that can be derived from the nonterminal identifiers and expressions).

(1) program	→	**program** identifier **var** declList **begin** stmtList **end**
(2) declList	→	**declare** identifier
(3) declList	→	**declare** identifier ; declList
(4) stmtList	→	stmt
(5) stmtList	→	stmt ; stmtList
(6) stmt	→	identifier := exp
(7) stmt	→	**begin** stmlList **end**

Table 2.1 A context-free grammar for *SimpleL*

It is easy to see that this context free grammar for *SimpleL* language depicts program scheme shown in Program 1.

Program 1 Program scheme for *SimpleL* language

```
program p
var
    declare q;
    declare r;
begin
    stmt;
    stmt;
    stmt;
end
```

The attribute annotated grammar, i.e. attribute grammar for *SimpleL*, is in Table 2.2.

(1) program	→	**program** identifier **var** declList **begin** stmtList **end** $stmtList.env = declList.env$
(2) declList	→	**declare** identifier $declList.env = \{identifier.id\ \}$
(3) declList	→	**declare** identifier ; declList $declList_1.env = \{identifier.id\} \cup declList_2.env$
(4) stmtList	→	stmt $stmt.env = stmtList.env$
(5) stmtList	→	stmt ; stmtList $stmt.env = stmtList_1.env$ $stmlList_2 = stmtList_1.env$
(6) stmt	→	identifier := exp $exp.env = stmt.env$
(7) stmt	→	**begin** stmtList **end** $stmtList.env = stmt.env$

Table 2.2 Attribute grammar for *SimpleL*

Functional dependencies (FD) among attribute occurrences in a production p can be represented by a directed graph, called *dependency graph*, denoted by D(p). The in-depth definition of *dependency graph* is :

1. For each attribute occurrence b in an attribute grammar G, the graph contains a vertex b'.

2. If attribute occurrence b appears on the right-hand sie of the attribute equation that defines attribute occurrence c, the graph contains an edge(b', c'), directed from b' to c'.

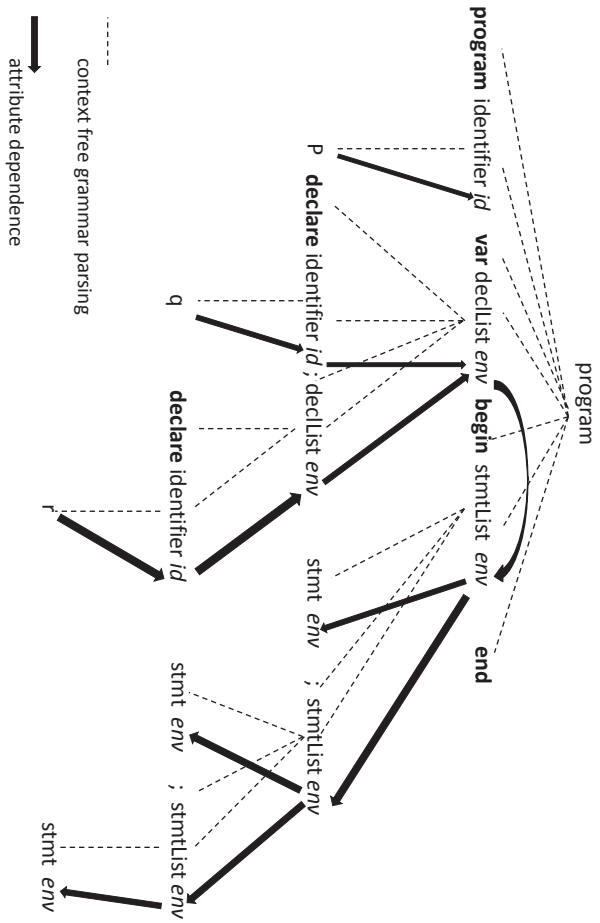


Figure 2.8 Parse tree and attribute dependency graph

In Figure 2.8, the dotted lines show the parsing of Program 1 against the context free grammar specified in Table 2.1. A solid arrow shows the flow of attribute values. It also

shows the dependence relation that echoes the attribute flow. Note that since there is no cycle formed by the solid arrows in Figure 2.8, the attibute grammar for *SimpleL* is a non-circular attribute grammar. In this research, we confine our attention to *noncircular* grammars only. Theoretically, a grammar is noncircular if it is not possible to build a derivation tree in which attributes are defined circularly. Besides, circularity issues of attribute grammars have already been well solved by Knuth in 1971. He showed that circularity is a decidable property of attribute grammar [36], and proposed an algorithm to test circularity [36]. In 1975, Jazayeri et al. showed that such algorithm is of inherently exponential complexity [37]. We will take full advantage of the intuition a dependency graph offers, such as that in Figure 2.8: after transforming an attribute grammar to a dependency graph, if it is acyclic, then it is sufficent to say that the original grammar is a noncircular attribute grammar. We directly use this conclusion for the rest of this work. Another default convention we follow is that we deal only with *well formed* attribute grammars where each production has exactly one attribute equation for each of the lef-hand-side nonterminal's synthesized attribute occurrences and the right-hand-side nonterminals' inherited attribute occurrences.

Intuitively, variables such as p and q have to be declared before they are being used. The declaration records a variable by supplying its id which will be stored in the symbol table, which is the global environment drawn from the environment of the declaration list.

Attribute grammar shares a great deal in common with functional paradigm, or more broadly, declarative paradigm-based programming models. Some researchers even argued that an attribute grammar per se is a delarative functional language [38, 39, 40, 41]. Indeed, our proposed $Situ^f$ domain specific language follows functional paradigm to allow a domain expert to specify situations. In order to do that, we find that all important contexts surrdounding a situation, both environmental contexts and behavioral contexts, fit well into an *attribute grammar model*. They serve as the *attributes* for each production to specify situations. The attribute grammar provides a good modeling tool so that we can combine the syntax rules and static semantic rules for $Situ^f$ under one formalism. More interestingly, attribute grammars have several desirable qualities as a model for specifying the intrinsic relations between a situation and its contexts in $Situ^f$. A good example is that it supports modular specification of

situations through contexts, as each attribute equation under attribute grammar is local to only one grammar production. Any attribute to each equation can be thought of as a placeholder to an attribute equation, just as parameters to a function in many mainstream languages such as C++, Java etc. In addition, although propagation of attribute values through the derivation tree is not specified explicitly by attribute grammar, it is implicitly defined by the equations of the grammar and the form of a tree. The functional situation specification language *Situ*^{f} is based on attribute grammar formalism.

In general, for each situation scenario to be specified in *Situ*^{f}, the domain expert needs to define production-like grammar rules, with a corresponding set of attribute equations which take each context variable as an attribute. The ensemble of context variables reflects domain specific vocabulary particularly pertaining to the situation domain. While the main success scenario is specified as a functional based production rule in *Situ*^{f}, the distribution of context variables over that particular situation is captured by the machinery built in the associated set of semantic rules.

Based on the online paper review scenario, the following example is composed to draft a real *Situ*^{f} program. In it, the perspective of a domain expert is taken and various language features are picked along the way. Towards the end, the reader is expected to have an overall feeling and some tangible hands-on experience. Having learned this example, readers will be ready to see a complete syntactic and semantic description of the *Situ*^{f} language - a main task for the next chapter.

2.7.3 Paper review example

Returning to the paper review example, let us assume a domain expert at work who specializes in the web-based paper review process. She is also trained and understands the fundamentals of *situation theory* proposed in [1]. First of all, she sketches the main success scenario of a general online paper review scenarion.

$$goal \ \rightarrow \ map \ (\ (download.review),\ [paper_1,\ paper_2,\ \ldots\ paper_n]\ ) \tag{2.4}$$

The parse tree depicted in Figure 2.9 reveals the intrisic structure of (2.4).

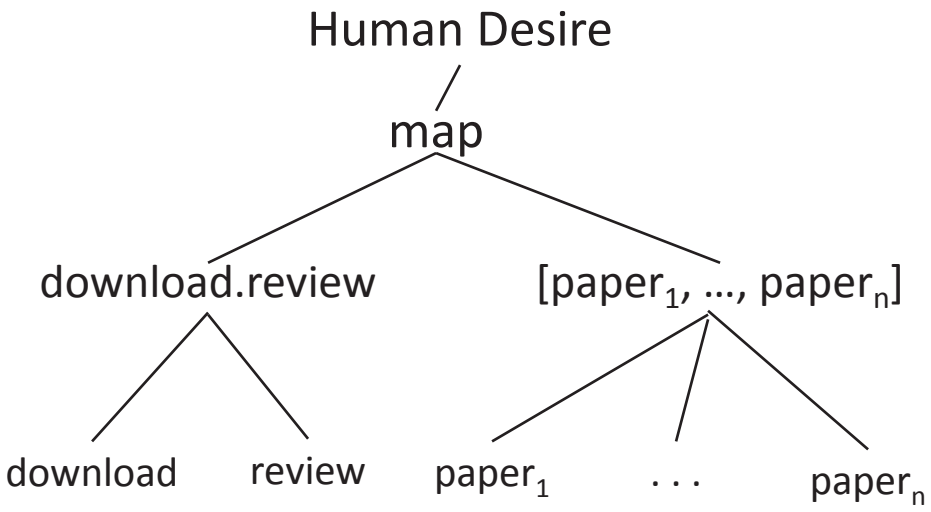


Figure 2.9 Parse tree for paper review situation

Note that given *Situ*^{f} a domain specific language, the vocabulary used in *Situ*^{f} script are all domain vocabulary. For example, (2.4) is specific to the online paper review domain where paper, download, review, etc. . . , are frequently used. The main actions captured are modeled as names of functions, e.g. *downoad*, *review* etc. The "." notion used in the expression of *download.review* refers to a composed action incorporating *download* and *review*, in which the output of *download* is pipelined to the input of *review*. Indeed, being a functional language, *Situ*^{f}'s situation specification particularly reflects this characteristic.

Each situation defined under *Situ* framwork [1] contains a temporal tag defining the time instant for its being. The temporal order within a situation sequence is recorded through these temporal tags. Note that in (2.4), the download action happens temporally *before* review as each paper is concerned. Figure 2.10 offers a pictorial explanation. Indeed, the functional composition notion lends itself well to temporal sequencing representation. Specification (2.4) simply reads: for each paper *first* download it, and then review it.

Since each situation lives within its correlated context, under *Situ* framework a situation

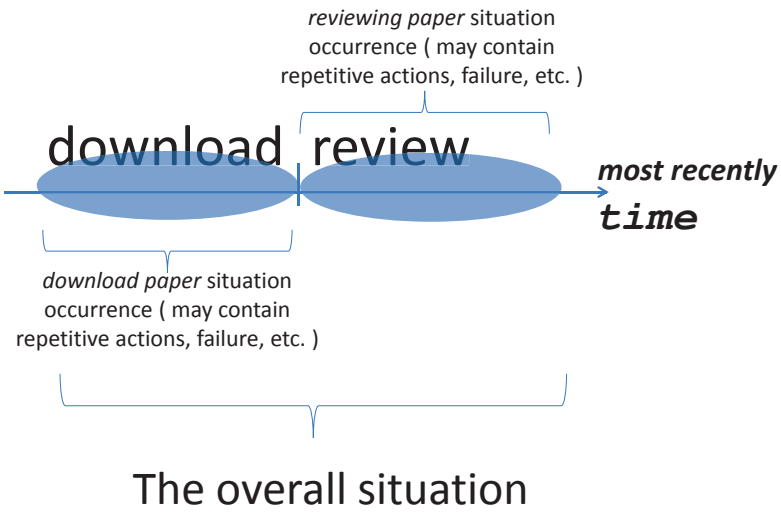


Figure 2.10 Temporal ordering of situations

is conceptually defined to include environmental as well as behavioral context. To capture the context surrounding a situation, a domain expert usually needs to focus first on the data source. This translates to a series of problems. Particularly in this paper review example:

- Most often than not, the domain expert would not be able to know the accurate number of papers assigned to a reviewer. These details, which are tied to specific circumstances, are beyond the knowledge as well as the concern of a domain expert;
- The concrete software support for downloading and reviewing paper in MyReview system is not shown in (2.4). To connect situations with their working circumstances in real world, this information is indispensible.

To solve these problems, *Situ*^{f} provides certain features which are illustrated through Program 2. Program 2 can be considered *Situ*^{f}'s implementaion of (2.4).

There are several critical points demonstrated through Program 2:

Program 2 A *Situ*^{f} program for paper review situation

```
include common_service_GUI
import contextSpec_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
      review<paper:Review>@129.186.93.0:/home/      \
          myreview/COMPSAC2011_Training/Review.php;

  situation
    map download.review paper();
    // other temporally ensuing situations
    . . . . . .
```

- First, 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 page 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 page, the contextual information will be captured by @ and sent back to program *paperReview*.

- Notice that review and Review are different program entities in Program 2: the former is the action whereas the latter is the result coming out of that action thus declared as data in Program 2. This example shows the effect of variables naming in *Situ*^{f}.

- @ is an I/O based language feature. Once declared, data and action can be used to

construct a situation. Another point worth attention in Program 2 is by the use of (), which follows declared data paper. **()** is another I/O based feature *Situ*^{f} offers. It is a data constructor: *paper()* returns a list of papers resulted by a series of paper downloading actions performed on Review.php page of the deployed MyReview system. The **end** of such a situation, that is the moment *paper()* stops constructing papers is when the user leaves Review.php page or simply logs out. User's leaving triggers an internal end-of-situation event **EOS** inside *Situ*^{f}-based environment.

- Closely related with SituIo and its **@** operator is the ¡program_url¿[8] defined in *Situ*^{f} attribute grammar, which will be introduced in the next chapter. This symbol specifies how *Situ*^{f} runtime is able to find the external counterpart that supplies contextual information to declared data, actions and situations defined by a domain expert's *Situ*^{f} program. Figure 2.11 illustrates it using a concrete example.

By nature, "**@**" is a monad type which should be familiar to readers who are experienced in Haskell, in particular Haskell's I/O mechanism. A monad helps to bind side-effect with a purely functional return value to form a new return type. It really is a sequencing mechanism:

1. to perform an I/O operation;
2. to return the retrieved value through I/O.

Having monad helps to keep a purely functional paradigm while still being able to combine impure side effects. Some computer scientists consider monad *an imperative sublanguage inside a purely functional language* [42]. Monad has its root in category theory and has already been well studied by logicians and theorists; therefore we do not step into the theoretical side of monad much. Rather, our focus is to precisely specify the semantics arising from monad-based *SituIO* mechanism so as to well establish the link between situation structures derived from a *Situ*^{f} program and the correlated contextual information gathered externally. Further, we

[8] ¡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;

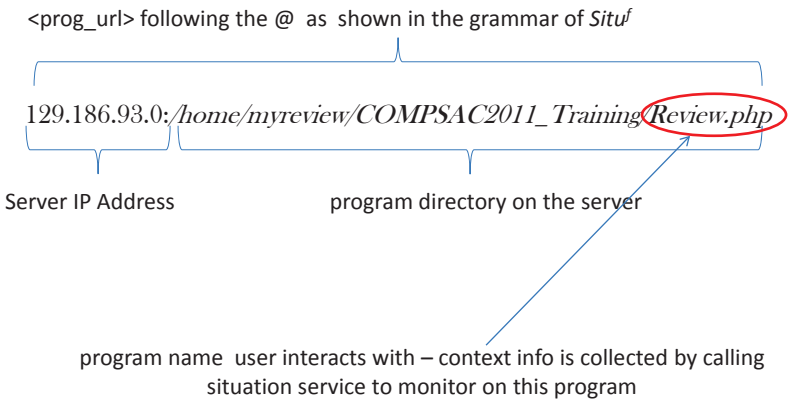


Figure 2.11 An example of the ¡prog_url¿ grammar symbol in *Situ*^{f} grammar

focus on the flow between the user actions and the reaction from the graphical user interface of a software whose evolutionary nature falls into our research spectrum. Under this purpose, the peculiar meaning of *Situ*^{f}'s monadic feature @ is:

1. perform I/O to connect to an existing software's Graphical User Interface actions.[9] This step generates the side effect, and then;

2. return the most recent context values supplied by related GUI gadegts. This step generates the main functional return value of a *Situ*^{f} function.

The details of the attribute grammar for *Situ*^{f} are given in full length in the next chapter. For now, as a gentle introduction to the formal treatment of *Situ*^{f}'s attribute grammar, more importantly, to further explain the motives behind the proposal of Program 2, Figure 2.12 is given to show the attribute propagation around the *paperReview* situation [10].

[9] specified by *¡program_url¿* as in the attribute grammar.

[10] Program 2 defines context-oriented paperReview situation, following the original *Situ* framework where all situations are based on behavioral and environmental contexts.

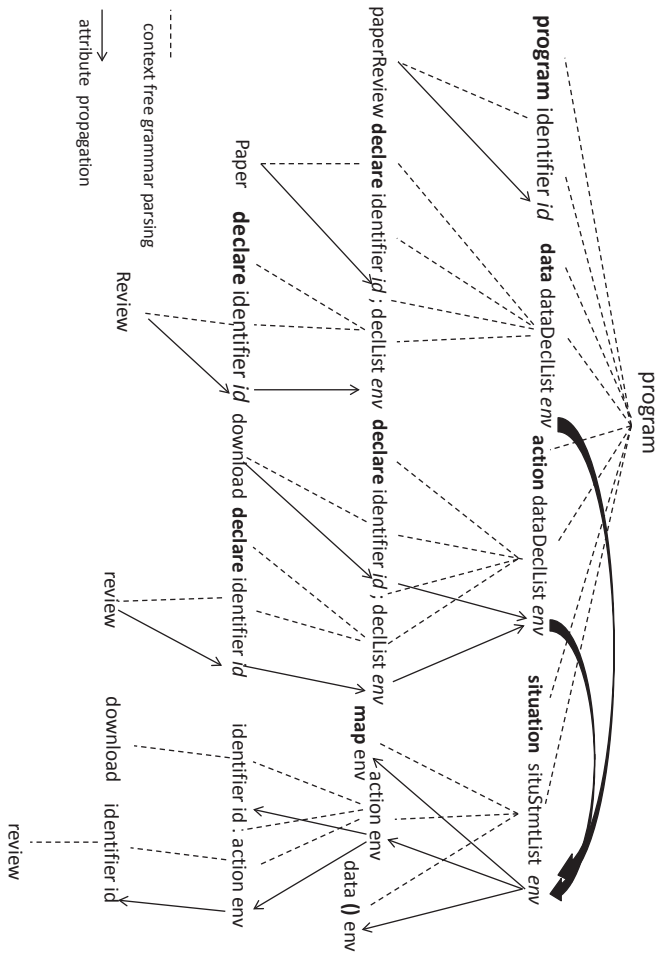


Figure 2.12 Parse tree and attribute propagation graph for Program 2

The data and action declarations in Program 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 2, the source of data for paper and review are the server-side *Review.php* page. This simply means that each time the user downloads a paper through Review.php page, the context surrounding that paper such as author list, email contact and abstract etc . . . will be collected over the Graphical User Interface and sent back to Program 2. More concretely, through paper(), context information of all assigned papers 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, the Review will be captured in terms of its context ensemble: an aggregation of review comments, review score, suggestions to the Program

Commettee, 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 2 in statement *map download.review 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 (*download.review*) to its second input - a list of papers. Readers familiar with functional programming know well that theoretically map represents 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; specifically in specification (2.4), 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 (2.4).

This section only illustrates *map* example. The precise computational meaning of *map, reduce, filter* and *apply* is given using small-step operational semantics in 3.6.2.

CHAPTER 3. FORMAL DEFINITION OF *Situ*^{f}

The motivation behind the design of *Situ*^{f} domain specific language is to provide a set of features easy to use yet powerful enough to meet the following requirements:

- Situation centric: the basic language constructs revolve around situation definition given in [1]. In every aspect, *Situ*^{f} is a continued research effort under the original *Situ* framework towards a programming model;
- Simplicity: in essence, *Situ*^{f} is a language to specify situations. A program, or script written in it emphasizes the "what" rather than "how" process, therefore it encourages smaller than average program size. As explained earlier, functional paradigm fits in nicely, hence the name *Situ*^{f}.[1];
- Situation modularity and situation re-usability: more likely than not, a domain expert usually focuses on one situation at a time. Separate concerns, when combined with situations, translate to the need of a reusable and modular situation specification mechanism;
- Conducive to the generation of an environment. The main aim of this work is to construct a *Situ*^{f}-based environment that provides an initial realization of *Situ* framework.

Under these goals, we give a formal definition for *Situ*^{f}. The syntactical part of the definition is provided through a context free grammar, and an attribute grammar based approach and operational semantics are employed to define the semantics.

3.1 Syntactical definition of *Situ*^{f}

The context free grammar, i.e., concrete syntax, of *Situ*^{f} is shown in Table 3.1.

[1]The superscript "f" refers to the term "functional"

Since attribute grammar is based on context-free grammar, which specifies concrete syntax, we will not discuss in depth the abstract syntax of *Situ*^{f} here. But to make the picture complete and to show the simplicity of the structure of the *Situ*^{f} program, we attach *Situ*^{f}'s abstract syntax in Table 3.2.

3.2 Semantic definition of *Situ*^{f} through attribute grammar

Table 3.3 through 3.5 give the definition of *Situ*^{f} in terms of attribute grammar.

(For formatting purposes we split *Situ*^{f}'s attribute grammar into Table 3.3 through Table 3.5.)

3.3 SituIO: the IO channel for *Situ*^{f} environment

To closely follow and provide built-in support for the original *Situ* framework [1], where each situation is identified to be associated with a set of behavioral as well as environmental contexts, *Situ*^{f} includes in its language proper a unique context-oriented I/O mechanism called SituIO. Through SituIO:

- Low level information processing idiosyncrasies are encapsulated, allowing domain experts to focus more on the level of situation specification;
- The reasoning model of a purely functional langauge is maintained;
- The contexts surrounding each situation is collected in *realtime* and directly provides low level support for *Situ*^{f}'s attribute grammar-based semantics;
- The human-centric nature of a situation is enhanced in *Situ*^{f}-based environment.

Before further elaborating the definition of SituIO mechanism, let us first conduct a brief historical review over sensitive IO issues affecting functional programming model [43].

A functional program contains a number of definitions, including values, functions, etc., as shown the top of page 52.

v :: Integer

```
v = 38

function :: Integer →  Integer
function n = v + n
```

The net effect of these definitions is associating a fixed value with each name:

- for v: an Integer 38;
- for function: a mapping from Integer to Integer.

Next let us take into account defining a function to get an integer value from an IO. Some approaches, for instance [44, 45] is to include the following operation:

$$inputInt :: Integer \tag{3.1}$$

The intent is to read an integer from the input stream where the value read becomes the value given to inputInt. Each time inputInt is evaluated it will be given, possibly, a new value, therefore not a fixed value under the very same function name: inputInt. To see why this causes a problem, let us examine the following example:

$$inputDiff = inputInt - inputInt \tag{3.2}$$

Suppose the first input read in through the input action is 4, and the next is 2. The result of inputDiff is 2 or -2, depending on the *order* in which the arguments to the operation '-' are evaluated. This uncertainty breaks the reasoning model over functional models: for any function that takes the same input (including no input) the output should be the same[2], therefore inputDiff should always be equal to zero, which is **not** the case should inputInt be defined as in (3.1).

The reason for this is precisely that the meaning of an expression is *also* determined by *where* it occurs in a program. This breaks the functional model. More serious is the fact that

[2]Mathematical definition of a function as you can find in any elementary math texts.

since *any* function in a functional language can use inputInt just as any other pure function, its unpure definition can be "epidemic" to the reasoning chaining to a greater range. Because of this, I/O has turned out to be a troublemaker for functional programmers for an extended period of time. A good historical overview can be found in [43].

In thinking about input/output, it makes more sense to think of actions happening in *sequence*, e.g., some input might be read first, and then on the basis of that further input might be read, or output might be produced. The current Haskell language standard follows this scheme, and provides the type *IO a*, that is, do some I/O and then return a value of type a. I/O is Haskell's primitive built-in mechanism as well as the mechanism to sequence these I/O's.

What Haskell and other functional languages have not offered, but is indeed needed by *Situ*^{f} language however, is to connect the stream of external contexts with each internally specified situation. The context stream is generated by user's raw actions, such as mouse click, filling out a textbox, etc. Those actions lead to raw context data, which after having been collected and fetched into SituIO, are organically organized as meaningful components surrounding a defined situation. *Situ*^{f}'s treatment of contexts and situations follows immediately the original conceptual definition of situation found in [1], i.e.

$$situation \sim (d, A, E)_t$$

where A and E stands for behavioral contexts and environmental contexts respectively, whereas d reflects human desire, all captured at time instant t.

As compared with mainstream IO type usually found in a functional language such as Haskell, SituIO distinguishes itself by its intrinsic, domain specific support for human centric situations, which provides indispensable support for the task of user-centric reasoning of situations.

3.4 The Monadic "@" to set up *SituIO* channel

Situ^f provides two languages features, "@" and "()", to support *SituIO*.

"@" is designed to bind context stream with a program url. A program url, whose grammar symbol is <prog_url> shown in Table 3.1, follows "@" in a legitimate *Situ*^{f} program. It can be thought of as a context generator to initiate the flow of context values captured at the interface level into the data or action variable declared on the left hand side of "@". The interface between human user and software typically consists of a GUI or of sensory type. The program url can reference both locally and remotely deployed software components. Program 2 contains the following program url:

$$129.186.93.0 : /home/myreview/COMPSAC2011\_Training/Review.php \tag{3.3}$$

(3.3) points out that context values surrounding the paper review situation as specified in Program 2 are generated from the PHP program called Review.php, which is deployed under the directory of: /home/myreview/COMPSAC2011_Training, on server whose IP address is 129.186.93.0 .

Note that in Program 2, data type variables like paper, Review and action type variables such as download and review, are all bound to the user interface (3.3) by "@", meaning that (3.3) is the user interface where a user's behavioral as well as the environmental contexts are captured. This binding enabled by "@" set up the context I/O channel so that the context values collected through interface (3.3) are available for SituIO mechanism and to be assigned as context values for paper(data), Review(data), download(action) and review(action). The context values for each declared data and action are governed by the attribute grammar given in Table 3.3 through Table 3.5. *Situ*^{f}'s context I/O pipeline is impossible without "@".

"@" is used to set up the context I/O pipleline, and therefore it is always used in the declaration section of a *Situ*^{f} program for data and action variables. The underlying workings of *Situ*^{f} runtime, especially the convertion from externally captured raw data to internally meaningful context, is intimately controlled under the "()" operator.

3.5 The Monadic "()" to convert user data to situation contexts

Following *lazy evaluation* strategy, "()" is designed by the call-by-name principle, as opposed to the call-by-value principle. It incrementally supplies contextual data captured externally one unit at a time to the specified situation under a *Situ*^{f} program. From a domain expert point of view, "()" reifies the *iterator* pattern.

In the specification of paper review situation of Program 2:

$$map \quad download.review \quad paper() \tag{3.4}$$

"()" is used to supply one paper at a time to the action of download and review, under the control of the built-in situation constructor - *map*. Each paper is declared in the data section and is bound to an external program url resource. Note that (3.4) does not give any information as to the number of papers to download and review. This is because that under Situation (3.4):

- the actual number of papers in general becomes known at run-time only;
- the actual number of papers is beyond the scope of core purposes or interests of a domain expert;
- "()" is in fact an *iterator* over contextual data, e.g. paper, for example (3.4).

The adoption of *iterator pattern* in the design of "()" targets to make situation specification more re-usable, flexible and friendly to domain experts.

3.6 A precise description of SituIO under *Situ*^{f} language

In this effort, *structural operational semantics* is used to precisely describe the machinery of the internal workings of *SituIO* at runtime, with special focus on the monadic "()" feature already introduced to *Situ*^{f}. "()" well insulates the runtime complexity of incrementally fetching contextual data from external context sources one at a time. Looking at Program 2, the point should be clear that by encapsulating those low-level details as to how many papers are expected to be reviewed, or for each paper how to get the related contexts from a user's action, etc. . . ,

the domain expert can focus entirely on the most important part of specifying a paper review situation without missing a beat.

3.6.1 Overview of semantics of programming languages: denotational, axiomatic and operational semantics

The semantic considerations are of the utmost importance both in the design of the whole or part of a programming language and during the reasoning about properties of a particular program written in that language. Historically, three types of semantics stand out during theoretical and practical development of modern programming languages.

Denotational semantics was the first mathematical account of program behavior; it arose in the late 1960s [46, 47, 48] and was poineered by Dana Scott and Christopher Strachey. In denotational semantics, the behavior of a program is described by defining a *function* that assigns meaning to every construct in the languages. The meaning of a language construct is called its **denotation**. Typically, for an imperative program, the denotation will be a state transformation, which again is a function that describes how the final values of the variables in a program are found from their initial values.

Structural operational semantics came into existence around 1980 due to Gordon Plotkin [49]. By borrowing some of the techniques developed for denotational semantics, structural operational semantics proposed a more satisfactory and simpler operational theory where greater emphasis is placed on defining the effect of running a program in terms of its structure. More specifically, behavior of a program is specified by defining a transition system whose transition relation describes the evaluation steps of a program. Structural operational semantics made it possible to give a simple account of concurrent programs, which is in general very complicated, using denotational semantcs. Structural semantics is syntax-directed; it uses *abstract syntax* to set the stage to define allowable states, which eventually leads to the desired transition system. Two or more different operational semantics can be defined for a single language, for example the **big-step** operational semantics, a.k.a. evaluation semantics, gives a high-level, rather abstract description (from programmer's point of view) of a language, while the **small-step** operational semantics, a.k.a. computation semantics, tends to provide an account of a language

from a point of view that is closer to an interpreter or compiler.

Axiomatic semantics [50, 51] takes a more direct approach than the other two kinds of semantics: rather than deriving rules by first defining the behavior of programs and then generating rules from this definition, axiomatic semantics takes rules in the form of mathematical logics *themselves* as the definition of the language. This includes assertions that must hold before and after the language construct has been executed. The meaning of a program, then, becomes just what can be proved about it based on the rules.

During the '60s and '70s, operational semantics was generally regarded as inferior to the other two styles [52]. They were considered useful for quick and dirty definitions of language features, but inelegant and mathematically weak. Examples include some of the earliest attempts at IBM's research laboratory in Vienna in the late sixties [49]. But in the 80's, the more abstract methods, i.e. denotational semantics, began to encounter increasingly difficult situations such as nondeterminism and concurrency. For axiomatic semantics it was procedures. The simplicity and flexibility of operational methods came to be more and more attractive to the research community. Among these Plotin's *Structural Operational Semantics* [49] is of special interest, for which: Kahn [53] proposed an extension called natural semantics to accommodate higher-order functions beyond first-order ones; Robin Milner used Plotkin's approach to give a labelled semantics to his Calculus of Communication Systems (CCS) [54, 55, 56]. These approaches introduced more mathematically elegant formalisms and showed potential, for powerful mathematical techniques developed in the context of denotational semantics to be transferred to a structural operational setting.

Until this day, operational semantics has remained an active research area in its own right and is often the method of choice for defining programming languages and studying their properties.

Small-step operational semantics, also known as computational semantics, is chosen for this work to precisely define the operational properties of SituIO. We propose a succinct approach when applying operational semantics.

3.6.2 Abstraction of SituIO

To articulate the operational semantics for SituIO, we first propose the following abstractions:

- The grammar symbol "<data>()" found in Table 3.1 is abstracted as a context stream expression noted by **se**;
- The grammar symbol "<data>" found in Table 3.1 is abstracted as a non-stream expression noted by **e**;
- We distinguish the evaluation for **se** from that for **e** by providing the following two forms of evaluation relations:

$$\Longrightarrow_N \quad vs. \quad \Longrightarrow_S$$

The reason for this refinement is that "()" is a non-compile time I/O operation, there is no way for *Situ*^{f} compiler to know in advance exactly how many data units in total will be streamed. The detailed internal workings of context stream within *Situ*^{f}, which are hidden from the *Situ*^{f} programmer, e.g., a domain expert, will be elaborated through operational semantics given in 3.6.3.

$\Longrightarrow_N$ is the evaluation relation for *non-stream expressions*. The type of $\Longrightarrow_N$ is:

$$\Longrightarrow_N : D \rightharpoonup ENV \rightharpoonup E \rightharpoonup E$$

Any non-stream expressions will be evaluated by $\Longrightarrow_N$, which reduce one such expression to another under declaration D, to interpret function and data name, and environment ENV, to internally check the name bindings in *Situ*^{f}.

$\Longrightarrow_S$ is the evlauation relation for context *stream expressions*. The type of $\Longrightarrow_S$ is:

$$\Longrightarrow_S : D \rightharpoonup ENV \rightharpoonup SE \rightharpoonup < E, SE >$$

This in turn leads to a derived relation $\overset{v}{\Longrightarrow}_S$, where v is a value generated from a Non-Stream expression e. The type for $\overset{v}{\Longrightarrow}_S$ is therefore:

$$\overset{v}{\Longrightarrow}_S : D \rightharpoonup ENV \rightharpoonup SE \rightharpoonup SE$$

Any context stream expressions should be evaluated by $\Longrightarrow_S$.

Note that $\overset{v}{\Longrightarrow}_S$ is a more graphic and intuitive rendering but in essence the *same* notion as $\Longrightarrow_S$. This point will be especially clear after introducing the semantic rules in 3.6.3.

The symbols used above to explain the type of $\Longrightarrow_N$, $\Longrightarrow_S$ and $\overset{v}{\Longrightarrow}_S$ are specified below:

D : *set of declarations for variables and functions*;

ENV : *Environment. It can be thought of as a function that binds a variable to storage location* [46, 47, 57]. *An environment roughly corresponds to a symbol table maintained by the compiler.*

E : *set of NonStream expressions*;

SE : *set of context stream expressions*;

v : *A value returned by Situ^f of type E.*

Note:

1. A concrete instance of value v is shown in Table 4.2, of section 4.1.1.1;
2. Details about the Environment model for *Situ^f*-based environment, i.e., ENV, is provided in 2.3

- The grammar symbol "<action>" found in Table 3.1 is abstracted as an action *function* F:

– F takes as input NonStream context variables $x_1, \ldots, x_k$ and returns a NonStream expression **e**. Formally:

$$F(x_1, \ldots, x_k) \Longleftarrow e$$

Function definition is represented by the notion of $\Longleftarrow$. The process in which a user carries out her action under a situation is abstracted by function definition, represented by the notion of $\Longleftarrow$, and e is the body of the definition. In an overly simplified but essential example of a function definition:

$$F(x) = x^2 + 1$$

F is the functional name variable. $x^2 + 1$ gives the definition of F, corresponding to the notion of e. Using the notion of $\Longleftarrow$, this function definition can be represented as:

$$F(x) \Longleftarrow x^2 + 1$$

3.6.3 The computational semantics of SituIO

Based on the abstractions of SituIO from section 3.6.2, we argue that the context streaming through SituIO can be imagined as *a stream language*. In pursuant of the semantics of SituIO, we formalize that intuition by using those aforementioned abstractions to propose such a stream language. It is an abstract language since we only care about its meaning and therefore give it an abstract syntax for semantic purposes. We do not intend to move towards any implementation level objectives. By giving computational semantics[3] to such an abstract stream language, the precise meaning of SituIO is captured. The abstract syntax is first provided for this abstract stream language under $Situ^f$.

1. Syntactic categories

[3]also known as small-step operational semantics; for more information, please refer to 3.6.1.

p ∈ *Program*

D ∈ *Declaration*

se ∈ *Stream Expression for External Context*

sx ∈ *Stream Variable*

e ∈ *NonStream Expression*

x ∈ *Unbound Context Variable*

F ∈ *Function Variable*

2. Formulation rules

p ::= ¡ se,D ¿

D ::= $F\,(\,x_1,\,\ldots,\,x_k) \Longleftarrow\; e$

where $x_1,\,\ldots,\,x_k,\; are\; free\; context\; variables$

se ::= $e : se \mid EOS \mid apply\; F\; se \mid map\; F\; se \mid$

$reduce\; F\; se \quad \mid filter\; F\; se$

e ::= $F\,(\,e_1,\,\ldots,\,e_k\,) \mid x \mid v \mid True \mid False \mid e : e$

The semantic rules are given in Table 3.6.

Interpretation of the sematnic rules:

- Rule Eval: This is the evaluation rule for the base-case stream expression in which a NonStream expression is immediately followed by EOS. This rule *links* stream and NonStream expressions of SituIO. According to the Formulation Rule, "e : EOS" by itself is a stream expression, therefore the evaluation resorts to the $\Longrightarrow_S$ relation.

  Intuitively, when a NonStream expression is followed by EOS, that is equivalent to evaluating e solely, since EOS is just a stream terminating signal. On the other hand, EOS comes only when context streaming is on. Therefore, to evalue "e : EOS", stream evaluation relation $\Longrightarrow_S$ is applied.

- Rule Map_1: EOS signals the *End Of Stream* condition for external context values in $Situ^f$. EOS echoes the well-known EOF, which signals no more data can be read from an external recourse in a computer operating system such as Unix or Linux as well as a popular language like C. EOS, as shown in the Formulation Rules, is a Stream Expression for External Context in SituIO. This rule means that under Declaration D and Environment ρ, the situation constructor map will computationally evaluate to EOS when signalled an EOS.

- Rule Map_2:

  - $e[v/x]$ denotes the result of substituting the context variable x in a NonStream expression e with context value v returned by SituIO. The precondition is that x is a free variable as pointed out, since the value associated with variables which appear bound plays no role in the evaluation of the expression e. Let us consider the following example. Without loss of generality, pseudo code is used.

The two experssions (let a = 6 in a * a + c) and (let b = 6 in b * b + c) have exactly the same value! The exact value depends on the value of c, the unbound and free variable. a and b are bound variables and they can be changed to any variables other than c without affecting the meaning of the expression. However, if we change a or b to c, we get a different value: *let c = 6 in c * c + c* will evaluate to 42. This is because c is a free variable of the original expressions of (let a = 6 in a * a + c) and (let b = 6 in b * b + c), and substituting c for a or b turns a free variable into a bound variable. In general, changing a free variable into a bound variable will change the value of an expression.

By using $e[v/x]$, we assume that x is a free variable. Should name clash occur, changing the free variable name(s) will avoid the hazard.

For additional mathematical machinery about free versus bound variables with regard to other constructs in programming languages, readers are referred to many excellent resources such as [58, 59, 60].

– $F(x) \Longleftarrow e$ means that the return value of function F is given by the evaluation of the NonStream expression e, hence $\Longleftarrow$ is noted as function definition relation in 3.6.2.

– since se is a context stream expression, such as the expression of " paper()" shown in Program 2, it must be evaluated by SituIO's stream expression evaluation relation $\Longrightarrow_S$. Using the declaration D and under programming environment ρ, the runtime stream expression se is evaluated through SituIO as the value v and produces the SituIO residual se', denoted by: $D, \rho \vdash se \overset{v}{\Longrightarrow}_S se'$. In other words, the first value in the stream associated with se is v. To find out about subsequent values we must apply the definition of $\Longrightarrow_S$ to se', i.e.

$$D \vdash se' \overset{v_1}{\Longrightarrow}_S se_1$$

Consequently, after n steps the result becomes:

$$v : v_1 : v_2 : \ldots : v_n : se_n$$

$v : v_1 : v_2 : \ldots : v_n$ is the partial result generated incrementally by map. According to Formulation Rules, $v : v_1 : v_2 : \ldots : v_n$ is a NonStream expression. While se_n is not EOS, $v : v_1 : v_2 : \ldots : v_n : se_n$ is, by Formulation Rules, a stream expression; Once se_n hits EOS, $v : v_1 : v_2 : \ldots : v_n : se_n$ remains a Stream expression since

$$se ::= se : EOS$$

In other words, the stream terminates and the incremental evaluation of map with regard to the context stream expression is then finished.

- **Note** that $D, \rho \vdash se \overset{v}{\Longrightarrow}_S se'$ mathematically expresses that SituIO is a **monad**: it returns v as the return value while causing the side effect se'.
- Computational semantics specifies the evaluation of *map F se* one step at a time, hence computational semantics is also named small-step operational semantics.

- Rule $Filter_1$:
  - Since the substituting context variable x in NonStream expression e with context value v does not involve stream expression, using declaration D and environment ρ e[v/x] is evaluated under the evaluation relation of $\Longrightarrow_N$. It does not directly involve SituIO.
  - When e[v/x] is NonStream-evaluated to True, i.e.,

$$D, \rho \vdash e[v/x] \Longrightarrow_N True$$

the value v is kept and appended to the partial result so that all value v that makes F(x) evaluate to true can be "filtered" and kept. This is exactly what the *filter* does.

- Rule $Filter_2$: when e[v/x] is NonStream-evaluated to False, i.e.,

$$D, \rho \vdash e[v/x] \Longrightarrow_N False$$

the value v is **not** kept in the partial result so that in the end all value v that makes F(x) evaluate to false will be "filtered out." This complements Rule $Filter_1$.

- Rule $Reduce_1$: to fully understand the reduce situation constructor and the reduce rule, let us first examine the example shown in Figure 3.1.

  In Figure 3.1, '+' represents an infix addition function and serves as a concrete instance of function symbol "F" in Rule $Reduce_1$. '+' takes a left and right operand, therefore the name "infix," before returning the summation value. A sequence of numbers, after being "reduced," in this case "added" as shown in Figure 3.1, the computation boils down to one number. This example, although quite simple, illustrates the power of the reduce stream expression, one of the four situation constructors in $Situ^f$. Moreover, more complex examples can be quite easily captured by the reduce expression, for instance, to generate a conference proceedings from all accepted papers.

  More formally,

  '+' is defined as x + y. To follow the Formulation Rule:

  $$'+' \Longleftarrow x + y$$

  x and y are free variables and the above form can be strictly translated to F(x,y) $\Longleftarrow$ e, where e refers to x + y for infix function '+'. Notice that '+' takes two parameters as any F in the reduce rule, this is regulated by "F(x_1, x_2)" in the premise of $Reduce_1$ Rule. If '+', however, is given *only* one argument, say 3, then that causes a *partial application* and therefore '+' turns into a **curried function** [59]. By e[v/x], which is a **curried function** for F, the follow-on " reduce F se' " expression in the conclusion part of $Reduce_1$ Rule becomes the sole argument of e[v/x].

- Rule $Reduce_2$: one way of looking at this rule is that it provides a base case scenario for the recursive $Reduce_1$ Rule. Its intuitive meaning should be straightforward.

- Rule $Apply_1$: this rule handles singleton stream where there is only one value being streamed through SituIO.

- Rule $Apply_2$: this rule points out that if the stream is an empty stream, just having EOS, then nothing happens, i.e. the result is simply EOS when the evaluation finishes.

Definition: If a stream expression *se* ending in EOS can be successfully evaluated, then *se* is said to be legally terminated.

Theorem 3.6.1. *All SituIO stream expressions are legally terminated by EOS.*

Proof. : The theorem is trivally true for EOS. Note that no semantic rule can apply to EOS implies that it immediately terminates.

If the stream expression takes the form of *e:EOS*, since it is a legal stream expression by Formulation Rules, the theorem holds.

If the stream expression takes the form of *map F se*, that is, it is a map stream expression:

Let se = e : se' . By mathematical induction, we assume se' can be legally terminated by EOS. There are two cases:

1. se' is EOS: by applying the Rule Map_2 and then Rule Map_1, *map F se* is successfully evaluated following the semantic rules of SituIO, hence legally terminated by EOS;
2. se' is not EOS: we apply Rule Map_2 on map F se' first. By induction hypothesis, $D, \rho \vdash map\ F\ se'$ will be, by $\Longrightarrow_S$, evaluated successfully since otherwise se' will not be legally terminated by EOS.

Therefore, the theorem holds for map stream expression.

If the stream expression is a filter stream expression, i.e., *filter F se*:

Let se = e : se' . By mathematical induction, we assume se' can be legally terminated by EOS. There are two cases:

1. se' is EOS: by applying the Rule $Filter_3$ and then Rule $Filter_1$ or Rule $Filter_2$ depending the truth value of F(e), *filter F se* is successfully evaluated by the semantic rules of SituIO, hence legally terminated by EOS;

2. se' is not EOS: we apply Rule $Filter_2$ or Rule $Filter_1$ on filter F se' first. By induction hypothesis, $D, \rho \vdash filter\ F\ se'$ will be, by $\Longrightarrow_S$, evaluated successfully since otherwise se' will not be legally terminated by EOS.

Therefore, the theorem holds for filter stream expression.

If the stream expression is a reduce stream expression, i.e., *reduce F se*::

Let se = e : se' . By mathematical induction, we assume se' can be legally terminated by EOS. There are two cases:

1. se' is EOS: by applying the Rule $Reduce_1$ and then Rule $Reduce_2$, *reduce F se* is successfully evaluated following the semantic rules of SituIO, hence legally terminated by EOS;
2. se' is not EOS: we apply Rule $Reduce_1$ on map F se' first. By induction hypothesis, $D, \rho \vdash reduce\ F\ se'$ will be, by $\Longrightarrow_S$, evaluated successfully since otherwise se' will not be legally terminated by EOS.

Therefore, the theorem holds for reduce stream expression.

Following exactly the same vein, we can show that the theorem holds for apply stream expression also.

□

Theorem 3.1 guarantees that a runtime environment can safely utilize EOS to delineate SituIO operations. $Situ^f$-based environment, the subject of next chapter, is precisely one of such kind.

(1) <program>	→	[**include** <service_name>][**import** <situaion_spec>] **program** <identifier> **data** <dataDeclList> **action** <actionDeclList> **situation** <SituStmtList>
(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**
(4)<dataName>	→	<identifier>
(5)<dataDeclList>	→	**declare** <dataName>**@**<prog_url>
(6)<$dataDeclList^1$>	→	**declare**<dataName>**@**<prog_url>; <$dataDeclList^2$>
(7)<action>	→	**None**
(8)<action>	→	<indentifier>
(9)<actionList>	→	<action>
(10)< $actionList$ >^{1}	→	<action>.< $actionList$ >^{2}
(11)<input>	→	**None**
(12)<input>	→	<identifier>
(13)< $input$ >^{1}	→	<identifier>,< $input$ >^{2}
(14)<output>	→	**None**
(15)<output>	→	<identifier>
(16)< $output$ >^{1}	→	<idnetifier>,< $output$ >^{2}
(17)<actionDeclList>	→	**declare**<actionList>(< $input$ >: < $output$ >) **@**<**prog_url**>
(18)<actionDeclList>	→	**declare**<actionList>(< $input$ >: < $output$ >) **@**<**prog_url**> **;**<actionDeclList>
(19)<situStmtList>	→	<situStmt>
(20)<situStmtList^1>	→	<situStmt>**;**<situStmtList^2>
(21)<situStmt>	→	**map** <actionList> <dataName>**()**
(22)<situStmt>	→	**filter** <actionList> <dataName>**()**
(23)<situStmt>	→	**reduce** <actionList> <dataName>**()**
(24)<situStmt>	→	**apply** <actionList> <dataName>

Table 3.1 A context-free grammar representing concrete syntax for $Situ^f$

Syntactic categories:

P	in	Program
IncludeStmt	in	Include Statement
ImptStmt	in	Import Statement
DataDecl	in	Data Declaration
DataDeclList	in	Data Declaration List
ActDecl	in	Action Declaration
ActDeclList	in	Action Declaration List
SituStmtList	in	Situation Statement List

Formulation rules:

P	::=	[IncludeStmt] [ImptStmt] DataDeclList ActDeclList SituStmtList
DataDeclList	::=	DataDecl | DataDeclList;DataDeclList
ActDeclList	::=	ActDecl | ActDeclList;ActDeclList
SituStmtList	::=	mapStmt | filterStmt | reduceStmt | applyStmt | SituStmtList;SituStmtList

Table 3.2 Abstract syntax for *Situ*^{f}

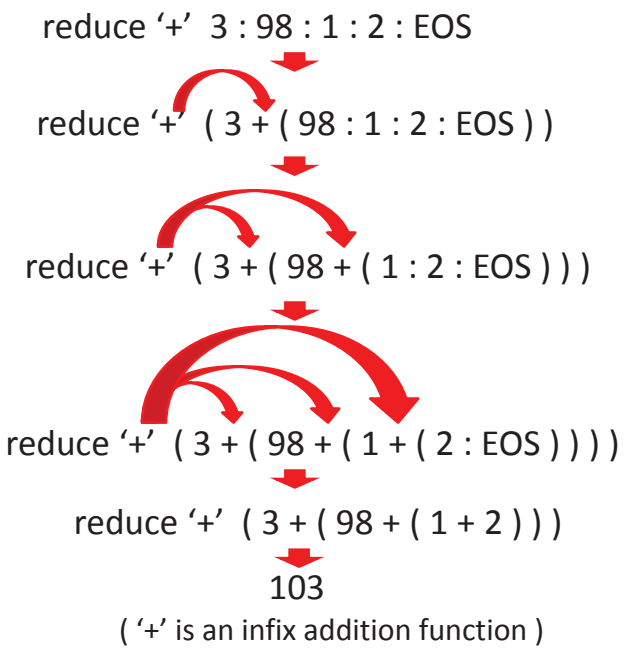


Figure 3.1 An example of reduce expression

(1) <program>	→	[**include** <service_name>][**import** <situation_spec>] **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> $\{ \quad < dataDeclList >_{env} = < dataName > .env$ $\cup \{< prog\_url > .id\} \quad \}$
(6)<$dataDeclList^1$>	→	**declare**<dataName>**@**<prog_url>; <$dataDeclList^2$> $\{ \ < dataDeclList^1 >_{env} = < dataName > .env$ $\cup \{< prog\_url > .id\} \cup < dataDeclList^2 >_{env} \quad \}$
(7)<action>	→	**None** $\{< action >_{env} = \phi\}$

Table 3.3 Attribute grammar for *Situ*^{f} (part 1 of 3)

(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>** $\{ < actionDeclList >_{env} = < actionList > .env$ $\cup < input > .env \cup < output > .env$ $\cup \{< prog\_url > .id\} \}$

Table 3.4 Attribute grammar for *Situ*^{f} (part 2 of 3)

(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>
$\{apply_{env} = < situStmt >_{env} \cup < actionList >_{env}$
$\cup < dataName >_{env}\}$

Table 3.5 Attribute grammar for *Situ*^{f} (part 3 of 3)

$$[Rule\ Eval] \quad \frac{}{D, \rho \vdash e : EOS \Longrightarrow_S e}$$

$$[Rule\ Map_1] \quad \frac{}{D, \rho \vdash map\ F\ EOS \Longrightarrow_S EOS}$$

$$[Rule\ Map_2] \quad \frac{\begin{array}{c} D, \rho \vdash se \overset{v}{\Longrightarrow}_S se' \\ D, \rho \vdash F(x) \Longleftarrow e \end{array}}{D, \rho \vdash map\ F\ se \Longrightarrow_S e[v/x] : map\ F\ se'}$$

$$[Rule\ Filter_1] \quad \frac{\begin{array}{c} D, \rho \vdash se \overset{v}{\Longrightarrow}_S se' \\ D, \rho \vdash F(x) \Longleftarrow e \\ D, \rho \vdash e[v/x] \Longrightarrow_N True \end{array}}{D, \rho \vdash filter\ F\ se \Longrightarrow_S v : filter\ F\ se'}$$

$$[Rule\ Filter_2] \quad \frac{\begin{array}{c} D, \rho \vdash se \overset{v}{\Longrightarrow}_S se' \\ D, \rho \vdash F(x) \Longleftarrow e \\ D, \rho \vdash e[v/x] \Longrightarrow_N False \end{array}}{D, \rho \vdash filter\ F\ se \Longrightarrow_S filter\ F\ se'}$$

$$[Rule\ Filter_3] \quad \frac{}{D, \rho \vdash filter\ F\ EOS \Longrightarrow_S EOS}$$

$$[Rule\ Reduce_1] \quad \frac{\begin{array}{c} D, \rho \vdash se \overset{v}{\Longrightarrow}_S se' \\ D, \rho \vdash F(x_1, x_2) \Longleftarrow e \end{array}}{D, \rho \vdash reduce\ F\ se \Longrightarrow_S e[v/x](reduce\ F\ se')}$$

$$[Rule\ Reduce_2] \quad \frac{}{D, \rho \vdash reduce\ F\ EOS \Longrightarrow_S EOS}$$

$$[Rule\ Apply_1] \quad \frac{\begin{array}{c} D, \rho \vdash se \overset{v}{\Longrightarrow}_S EOS \\ D, \rho \vdash F(x) \Longleftarrow e \end{array}}{D, \rho \vdash apply\ F\ se \Longrightarrow_S e[v/x]}$$

$$[Rule\ Apply_2] \quad \frac{}{D, \rho \vdash apply\ F\ EOS \Longrightarrow_S EOS}$$

Table 3.6 Operational semantics of *SituIO*

CHAPTER 4. *Situ*^{f}-based ENVIRONMENT

Compiling a *Situ*^{f} program involves the following major steps:

- Parse the *Situ*^{f} script;
- Link situation data structures;
- Link situation services;
- Set up SituIO channel.

The compiling process is refined and visualized by Figure 4.1.

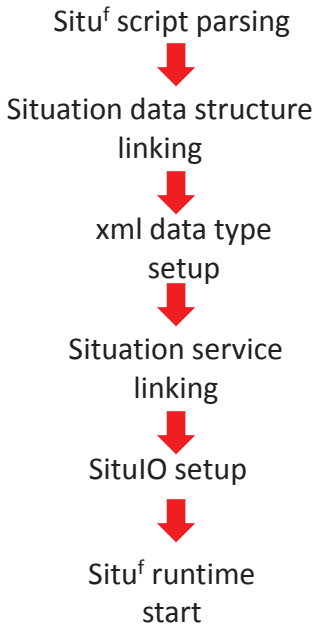


Figure 4.1 The compiling of a *Situ*^{f} script

After a *Situ*^{f} program is compiled, the corresponding runtime will start up; in the meantime, it brings up an environment shown in Figure 1.1.

A *Situ*^{f}-based environment (Figure 1.1) brings together all the important issues discussed so far. The centerpiece tying up this environment is *Situ*^{f} runtime.

While it is true that *Situ*^{f} programs are designed to precisely specify situations, the collection of context information for those situations is a direct relevant task [1], and thus it remains a central design purpose of *Situ*^{f}. The contexts that need be captured for each constituent of a specified situation, both for data and for action, are specified in XML format.

Note that *data* and *action* refer to the corresponding grammar symbols, ¡dataName¿ and ¡action¿, defined in Table 3.1. For a domain expert, the detailed aspects of context information are beyond her core concern, therefore *Situ*^{f} introduces the *import* clause to incorporate separately specified, XML-formatted context information for each situation.

In general, the context data are derived from actions exerted by a user over a software system. However, most often than not, the software system itself does not provide extra functionality to support context data collection tasks, not to say to report that collection to a third party. The design of *Situ*^{f} keeps that in mind and proposes a special *include* clause to let the situation services provide context collection capabilities. The author of this thesis and his colleagues have completed such general situation services: one targeting web-based applications which is written in Java Script, one targeting local Java JFrame based programs which is written in Java. Situation services help make the goal of collecting context information generally more reachable for different *Situ*^{f} programs.

4.1 Context specification and situation services

With concrete examples, this section elaborates on the technical details of context specification, situation serivces, 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} script, 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 come from data. In this writing, we 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. Furthermore, XML is considered the intermediate representation to exchange information, not just restricted to context information specification, but also serve the purpose of recording situation data structure as well as specifying situation services included in a *Situ*^{f}-based environment. In particular, there are two ways of defining the structure of any contents represented in XML format: DTDs, the older and more restricted way, and XML Schema, which offers extended possibilities, mainly for the definition of data types in XML [61]. Unlike DTD, which uses a different syntax separate from XML, thus needs a separate parser to interpret [61] its code, XML Schema by itself follows XML-based syntax to define new types. It is worth mentioning that as compared with DTD, which only provides character data type such as #PCDATA(Parsed Character Data) and #CDATA(Unparsed Character Data), XML Schema offers a variety of *built-in* data types:

- Numerical data types, including *integer, Short, Byte, Long, Float, Decimal*
- String data types, including *string, ID, IDREF, CDATA, Language*
- Date and time data types, including *time, Data, Month, Year*

The rich power for data structure description provided by XML Schema, as well as the ability to extend an existing data type, collectively make XML Schema a better candidate than DTD to serve the purpose of specifying context information in *Situ*^{f}-based environments.

In this work, we choose XML Schema to configure ”context” templates to synchronize the communication between a *Situ*^{f} program and the external context collection capabilities, under a *Situ*^{f}-based environment.

4.1.1 XML and context specifications

To provide concrete explanations and illustrations for key issues involved, let us revisit the paper review example given in Program 2.

The attribute grammar of *Situ*^{f} given in Table 3.3 through 3.5 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. Each paper declared in Program 2 in fact has the following attributes:

- abstract;
- author_name;
- author_affiliation;
- email_contact;
- paperID;
- submitTime;
- targetted_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** [62] bridges this gap. More concretely, *Situ*^{f} offers an *import* clause feature. As seen in Program 2, the "contextSpec_MyReview" following the "import" keyword is an instance of <situation_spec>, which is encoded as an XML Schema given in Table 4.1.

The *all* keyword represents a built-in mechanism XML Schema offers to construct new types of data. The detail is quoted as follows:

- *sequence*, a sequence of existing data type elements, the appearance of which in a predefined order is important;

```
<? 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=”targetted_trackName” type=”string” use=”required”
                        maxOccurs=”1” />
            <element name=”conference_name” type=”string” use=”required” />
        </all>
    </MyReview:complexType>
</MyReview:element>
</MyReview:schema>
```

Table 4.1 An XML Schema-based context template

- *all*, a collection of elements that must appear, but the order of which is not important;
- *choice*, a collection of elements, of which one will be chosen.

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 [61]. Complex data types can also be defined from already existing data types. The XML schema given in Table 4.1 essentially provides a template to help bind *paper*, a data variable declared in Program 2, and its closely related context. Note that Table 4.1 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 Table 4.1. That said, a paper under a different circumstance, such as the ”EasyChair” software system,

could involve compeltely 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. For more background information on namespace mechanism of XML schema, please consult [61].

Upon the import of a context specification where relevant information for a paper is provided, the *Situ*^{f} compiler automatically executes the following action:

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

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 3.3: $< dataName >_{env} = < identifier > .id$, when paper is declared, it replaces <dataName>.

In Table 4.1, "paper" is defined as a new type, where abstract, author_name, author_affiliation, email_contact, paperID, submitTime, targetted_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. In comparison, XML DTD only supports character data types, i.e., #PCDATA, for Parsed Character Data, and #CDATA, for unparsed Character Data,

4.1.1.1 Example of context values generated based on context templates

Table 4.2 is a direct instantiation of the XML schema based context template given in Table 4.1. Given that Table 4.2 strictly follows the format prescribed by Table 4.1, the latter is hence named context template.

Table 4.2 presents a concrete runtime example of a data value traveling through SituIO. This XML element is a value for the data variable "paper" delcared in Program 3.3, generated

```
<?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> Iowa State Universit </author_affiliation>
      <email_contact>jschneidre@iastate.edu</email_contact>
      <paperID>215</paperID>
      <submitTime>2012-07-24</submitTime>
      <targetted_trackName>Software Testing</targetted_trackName>
      <conference_name>IEEE COMPSAC</conference_name>
</paper>
```

Table 4.2 A sample context value collected at runtime using XML

under the governing of “contextSpec_MyReview” file which contains the XML Schema given in Table 4.1. The XML context information shown in Table 4.2 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 3.3 to Table 3.5. Table 4.2 is a concrete instance of value v, presented in the abstraction for SituIO in section 3.6.2.

4.1.2 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 details about this default service is provided in the next chapter as part of the feasiblity test of the *Situ*^{f} language. The default service offers the capability that, once deployed at the targeted url site, 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. For Program 2, it

is:

$$129.186.93.0 : /home/myreview/COMPSAC2011_Training/Review.php$$

To deploy a situation service requires security trust of the hosting system. Secutiry issues under *Situ*^{f}-based environment is part of our future research direction. An intersting question to ask from the perspective of *Situ*^{f} is: how can the design of *Situ*^{f} be evolved to be more *situation security aware*. This question can be answered as a result of our future work.

4.2 XML Situation data structures

The XML-based situation data structure is encoded and transmitted in XML format. Therefore it also involves XML Schema to define its data type similar to the discussion in 4.1.1. However, the XML situation data structure uses XML format to serve different purposes:

- To record all context data received through SituIO. This includes intemediate, as well as final functional results a running *Situ*^{f} program generates;
- the records saved in XML situation data structure are all temporally sorted;
- Save all user errors found from historical records affiliated with specific situation services, or freshly captured use errors.

4.3 EOS in *Situ*^{f}-based environment

For a *Situ*^{f} program, the compiler emits code to monitor the recorded actions of the software user, most typically through *comman_service_GUI*. Once the user moves on to a program url that is out of the scope specified in *Situ*^{f}, that information, once received, is interpreted as an EOS which wraps up the on-going stream expression evaluation. In the example of Program 2, once user's mouse clicks a url other than

$$129.186.93.0 : /home/myreview/COMPSAC2011_Training/Review.php$$

Situ^F runtime equivalently receives an EOS from the *Situ*^{f}-based environment.

CHAPTER 5. IMPLEMENTATION AND FEASIBILITY TEST

The overall objective of *Situ* framework is to improve our understanding of software evolution to involve a human-centered situation centric perspective. The *Situ*^{f}-based environment including the functional *Situ*^{f}, its underlying programming model, such as SituIO, all revolves around situations: situations are the basic building blocks that are given intrinsic support inside a *Situ*^{f}-based environment. In this section, we showcase the feasibility of *Situ*^{f}'s approach from an experimental test point of view; more specifically, we apply our mechanism on two occasions:

- a Java JFrame/Swing based Graphical User Interface software deployed locally. By writing code in *Situ*^{f} to adapt the user interface through collectively capturing user's action context, especially user's operation errors, and;
- experimental details on the paper review instance on the MyReview system, which is used as the sample case through Program 2 and else throughout this thesis.

5.1 Experiment on JFrame/Swing based User Interface Adaptation

5.1.1 Overview of adaptive user interface

Adaptive User Interfaces refers to a very broad category of interfaces. It can be precisely defined as a user interface that has the ability to adapt and change based on the user's performance. The interface is responsible for learning about user differences and preferences in order to make the decision for the user. This can be achieved by constructing a user model. This section will examine the various approaches that have been proposed for modeling adaptive user interfaces. Further, the various challenges involved in developing methogologies for adaptive user interfaces will also be examined.

Oftentimes, a software interface is designed targeting a major user group rather than any single user. But individual differences analysis offers good clues to possible exceptional usage issues, closely related to the evolution of the interface.

Several authors have looked at the possible ways for accommodating individual differences. In their paper, [63] analyzed the usability of user interface with respect to individual differences in spatial ability by using an interface that supports zoom with overview and detail. They found out that the users were able to perform much faster using an interface which had detail rather than the one with an overview.

[64] conducted an experiment to compare the performance of users with high and low spatial abilities. They were able to overcome poor performance due to low spatial ability by making some small changes to the interface like adding extra commands. One important point to be considered is improving the usability of the systems. Each and every individual has a different set of cognitive skills and preferences. [65] has differentiated the individual differences that are useful, those which are stable and have an impact on interaction. Stanney and Salvendy [65] were successful in using visual mediators to accommodate low spatial ability individuals. Their experiment found it to be useful to improve the search performance of low spatial ability individuals.

[66] suggested the development of a new inclusive design that includes people with disabilities. [67] presented a methodological design approach for implementing inclusive interface design. [68] has discussed the use of an inclusive interface for cell phones concentrating on the usability issues related to older users. [69] discussed issues about adaptive user interfaces for health-care systems, including the current knowledge, goals, and other significant characteristics of the user that are important for redesigned interfaces.

Although a high amount of work has been done on inclusive design approach, it should be noted that in an inclusive interface all the users are forced to use the same interface. This may not be the right solution. Interfaces appearing easier to use by a certain group of people may not be easier for a large number of people. [65] has pointed out that such an interface would result in a non-optimal interface for certain users.

To provide a much more accommodating solution, user interfaces must possess the ability

to adjust and adapt to individual user preferences or differences. There are wide varieties of device types, form factors and input methods that make it highly difficult for the programmers to create an interface which accommodates all these differences; thus an automated adaptive system becomes necessary under these circumstances. It must be noted that there is a key difference between the interface being adaptable and adaptive. Adaptable interfaces are those in which the users are given a choice. These interfaces are always under a user's control, but not all adaptive interfaces are controlled by the users.

In their study, [70] found out that users responded well to adaptable and adaptive user interfaces over the static interfaces. Adaptive interfaces can be differentiated from adaptable interfaces by the means of their overall performance and details in implementation. Adaptive interfaces require extra overhead in implementation.

Modeling user behavior is an interesting area as it will help us with new insights on the nature of human interaction with systems. It standardizes the way of building an adaptive user interface. It is vital to create a model for the effective implementation of adaptive user interfaces. [71] has discussed several techniques of user modeling and adaptive systems. The paper also provides a set of guidelines in building an adaptive user interface, in which user modeling is emphasized. In their design of an adaptive route advisor, Rogers et.al. in [72] used a model of driver preferences. The route advisor constantly updates the user model by interacting with the user and gathering user preferences. In developing the personalized word assistant based on episodes identification and association, Liu et al. [73] have recognized user behavior patterns and built a user profile that facilitates personalized interactions.

The most prevalent common example of adaptive user interfaces would correspond to those systems that are used to filter information and recommend users accordingly. Content based and collaborative filtering are among the basic approaches involved in adaptive systems. [74] discussed a system that retains profiles for individuals and later combines their predictions to produce both content-based and collaborative behavior. Pat Langley [75] discusses problems that involve more than just selecting from among a large set of documents or products. Horvitz et al. [76] built statistical methods and cost-benefit approaches to identify decisions on informing users in the area of context-aware interfaces and environments. Gajos et al. [77] found that

increasing the accuracy of the adaptive system significantly improved both performance and adaptive interface utilization. Further, both predictability and accuracy significantly increased participants' satisfaction. Shankar et al. [78] showcases statistically significant results indicating that an adaptive context-aware user interface can improve user-experience. An adaptive interface is rated based on the effectiveness of the algorithms used to determine the differences and preferences of the users. Langley [75] has suggested the use of machine learning algorithms for developing such interfaces. The adaptive interfaces must be able to build suggestion models that provide only recommendations to the user through the collective learning of user preferences/differences. It should be noted that the knowledge gained by the adaptive system must be capable of reflecting the preferences learnt. At the same time, an error-based model for adapting user interfaces to enhance software performance in field settings was proposed [75]. The indexing mechanism first proposed there as part of the error detection mechanism was later extended to handle complicated screen real estate indexing for indexing web pages. Both applications and software support are available as a concept-proof for that model.

Viano et al. [79] developed an adaptive interface that focused on the state of the process and state of the user. Tsandilas and Shraefel [80] examined the accuracy of algorithms for predicting user performance and satisfaction. Other works include investigating the use of user error detection as a means of adapting web pages to suit the abilities of older adults.

To show the effectiveness of $Situ^f$-based environment, I carried out an experiement that adopts the modeling of an adaptive user interface based on the errors made by the users interacting with the system. The experiments were conducted with the MyReview paper review system. Errors made by a user can be easily corresponded to adapting, thus evolving, the system, such that it minimizes errors committed by the user. The system built is capable of indexing the components on the interfaces, capturing the erros made by the user and producing multiple situation-driven user interfaces based on their preferences, reflected through errors made by the user.

5.1.2 Error, situation and the XML representation of context

A user is likely to make errors while using an interface, especially in adverse conditions. These user errors occur for many reasons. Some of them are:

- Environmental conditions (eg. Low visibility);
- User limitations (eg. Poor motor skills);
- Complexity of the interface

We focus on tapping errors, such as a user missing a button while tapping, incorrect taps where the user taps a wrong button, reversals and text entry errors. It is not easy to identify the reasons behind these types of errors. The reason for a missed tap can be anything from a user not being able to locate the button to the user not being able to hit the button properly. The types of errors supported by this experiment includes:

- Tapping Errors
  - Reversals
  - Missed Taps
    - Check Box
    - Radio Button
    - Menu
    - Text Box
    - Text Area
    - Button
    - Keyboard

The two types of tapping errors supported are reversals and missed taps. A reversal refers to the act of a user where he/she taps on a component and quickly reverses the action. We have built a situation service, named *user-action-error-detection-local*, which provides an error detection function *isError*. On the implementation level, it maintains a threshold value for the

distance around each component included in a GUI. If the missed tap is within the range of the threshold of a particular component, say a checkbox, then that tap is considered a missed tap for that component. The taps that do not fall under the radius of any of the components threshold value are considered to be errant taps. A series of errant taps suggest that the user might have difficulty in accessing the right component. Thus, by capturing repeated errant taps we can increase the threshold value for that particular component to capture the missed taps. This reflects the "adaptive nature" of this experiment.

Program 3 A *Situ*^{f} program for error-based adaptive user interface evolution

```
include GUI_error_detection_local
import contextSpec_user_interface_real_state

program _adaptiveUI
  data
    declare
      UI_component@255.255.255.255:~personalInformation_ \
        UI.class;
    declare
      tap@255.255.255.255:~personalInformation_UI.class;
    declare
      threshold@255.255.255.255:~GUI_error_detection_local;


  action
    declare
      isError<tap:Boolean>@255.255.255.255:GUI_error_ \
         detection_local;
    declare
      correctiveAction<tap:threshold>@255.255.255.255: GUI_ \
      error_detection_local;

  situation
    map correctiveAction(filter isError tap());
```

The personal information appeared in the *Situ*^{f} Program 3 is displayed in Figure 5.1.

The "contextSpec_user_interface_real_state" imported by Program 3 is, as explained in the last chapter, encoded in XML. It is shown in Figure 5.2.

Figure 5.2 shows the benefit of separating the concern of details in a graphical user interface real state. The domain expert can simply import the information, therefore being able to fully concentrate on the most import task, in this example, adaptively evolve the user interface incrementally!

5.2 Experiment on MyReview, a web-based paper review system

For the MyReview experiment, the *Situ*^{f} code that has been written to specify a paper review situation is demonstrated at Program 2. JavaCC [81], i.e., Java Compiler Compiler, is used to generate the parser for *Situ*^{f}. Our input to JavaCC is *Situf.jj*, a file ending in .jj, which contains production rules from *Situ*^{f}'s *Context Free Grammar* found in Table 3.1. Java code is injected under each production rule in *Situf.jj* to carry out the execution of attribute grammar rules given in Table 3.3 to 3.5 as syntax directed semantic actions. After conducting grammatical error checking on *Situf.jj* to prevent things like left recursion from happening, JavaCC automatically generates a parser for *Situ*^{f}. Specifically, the auto-generated *Situ*^{f} parser is a java file called *SitufParser.java*. It is automatically named by JavaCC by taking the prefix of the input grammar file name of "*Situf.jj*", and then appending "Parser.java" to it. More importantly, at the very end of *Situf.jj* file, which will only be executed after all parsing is done, lies a segment of java code that takes parsed names–both for data and for action–and XML contexts' specifications to set up XML context templates, to link in situation data structure and then finally, to start up the related situation services. Under standard java runtime environment, to parse a *Situ*^{f} Program 2 simply requires providing Program 2 as an input file to the auto-generated parser *SitufParser.java* before running java command *javac*; after that run the generated java class file using java command *java*. In the end, the *Situ*^{f}-based environment revolving around the specified paper review situation is started.

A side note is that Java socket is used to implement the context data transportation between a *Situ*^{f} program and its external runtime environment. This is because the site where a *Situ*^{f} program is run, usually locally to a domain expert, is most probably remote to where the MyReview system and the included situation services are deployed. At the current stage, we run Java sockets under a homogeneous environment where no security issues, such as firewall

policies, will arise. In future however, to meet the need of using a heterogeneous environment such as the World Wide Web, Simple Object Access Protocol(SOAP) will be considered to wrap up the sockets so that firewall policies will not deny socket access.

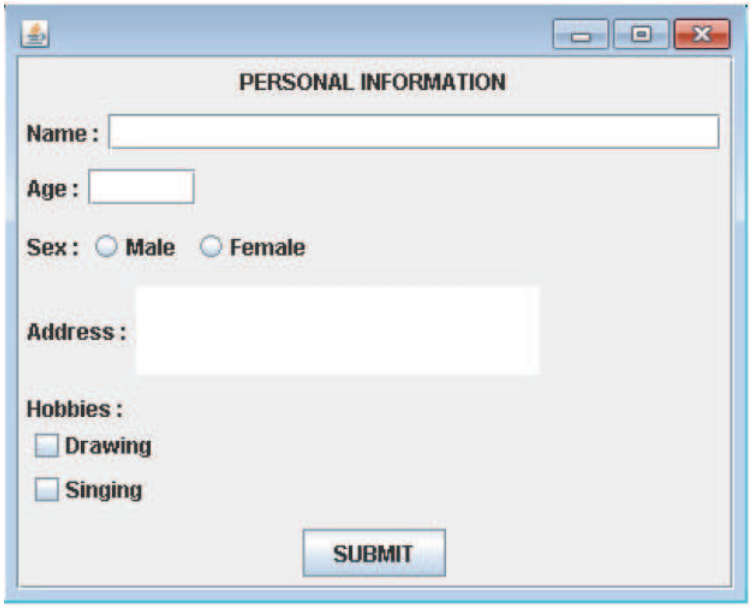


Figure 5.1 The Graphical User Interface for Personal Information

```
- <Screen screenId="1">
  - <ScreenSize>
      <Width>407</Width>
      <Height>329</Height>
    </ScreenSize>
  - <Components>
    + <Panels>
    + <ScrollBars>
      <Labels />
    - <TextBoxes>
      - <nameTxt>
        - <Start>
            <X1>49</X1>
            <Y1>31</Y1>
          </Start>
        - <End>
            <X2>383</X2>
            <Y2>51</Y2>
          </End>
        </nameTxt>
      + <ageTxt>
      </TextBoxes>
    + <TextAreas>
      <CheckBoxes />
      <RadioButtons />
    + <Buttons>
    </Components>
</Screen>
```

Figure 5.2 Context specificaation for Personal Information in XML

CHAPTER 6. CONCLUSION AND FUTURE WORK

6.1 Conclusion remark

This work marks the first step towards the realization of *Situ* framework. A domain specific functional programming language called *Situ*^{f} is proposed to bridge the concept of situation [1] to realistic computing circumstances. In this work, attribute grammar is used to aggregate dynamically captured contexts around each specified situation written in *Situ*^{f} by a domain expert. The communication between a stream of externally collected contexts and the internally specified situations is further modelled as *Monad-based SituIO*. This way, *Situ*^{f} is able to maintain its position in the purely functional category. Unlike a traditional language such as ANSI C where I/O is supplied by external libraries, SituIO is a built-in component of *Situ*^{f} language proper. Therefore, to completely define *Situ*^{f}, this thesis offers a precise mathematical description, namely computational semantics, also known as small-step operational semantics, for SituIO.

The design of *Situ*^{f} gives rise to an environment that employs XML as the intermediate representation for data transportation, context specification importing, situation services inclusion, as well as other runtime support purposes. Since such an environment, which we name as *Situ*^{f}-based environment, effectively encapsulates nontrivial underlying complexity, a domain expert is able to focus on situation level abstractions. Last, but not least, *Situ*^{f}-based environment closely supports separation of concerns for situation specification. This brings home a set of desirable results such as situation modularity and reusability.

To test the feasibility of our approach, two experiments were conducted. One was done over situations regarding user error based local graphical user interface adaptation; the other was to capture paper-review situations on top of MyReview, a web-based paper review system.

The results showed that *Situ*^{f} language, coupled with its affiliated *Situ*^{f}-based environment, provides sufficient expressive power as well as runtime support to help domain experts who write situation specifications to achieve various domain specific purposes.

6.2 Future work

The future work targets the following aspects:

- Add a strict type system on top of *Situ*^{f} to facilitate static time checking as well as static time program verification.
- Improve security mechanisms, from the perspectives of both theoretical modelling and practical implementation, for situation services inclusion;
- Engage sophisticated high performance compilation techniques, where functional languages generally have an edge over imperative languages for the purpose of ensuring program correctness.
- Integrate RDF, which shares the XML format, with *Situ*^{f}-based environment, especially into the mechanism of context specification importing. This way, *Situ*^{f}-based environment is able to enjoy the benefits derived by knowledge representation and knowledge reasoning techniques.
- Add "error propagation" machinery into *Situ*^{f} so that an erroneous situations may be captured in parallel with localizing the user and software errors to closely correspond to the situation specification.

BIBLIOGRAPHY

[1] Carl K. Chang, Hsinyi Jiang, Hua Ming, and Katsunori Oyama. Situ: A situation-theoretic approach to context-aware service evolution. *IEEE T. Services Computing*, 2(3):261–275, 2009.

[2] Michael Bratman. *Intentions, Plans, and Practical Reasoning*. Harvard University Press, 1991.

[3] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a bdi-architecture. In James F. Allen, Richard Fikes, and Erik Sandewall, editors, *KR*, pages 473–484. Morgan Kaufmann, 1991.

[4] N.G. Leveson. Intent specifications: an approach to building human-centered specifications. *Software Engineering, IEEE Transactions on*, 26(1):15 –35, jan 2000.

[5] Hua Ming, C.K. Chang, K. Oyama, and Hen i Yang. Reasoning about human intention change for individualized runtime software service evolution. In *Computer Software and Applications Conference (COMPSAC), 2010 IEEE 34th Annual*, pages 289 –296, july 2010.

[6] A. van Lamsweerde. Goal-oriented requirements engineering: a guided tour. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 249 –262, 2001.

[7] van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[8] Hua Ming, K. Oyama, and C.K. Chang. Human-intention driven self adaptive software evolvability in distributed service environments. In *Future Trends of Distributed Computing*

Systems, 2008. FTDCS '08. 12th IEEE International Workshop on, pages 51 –57, oct. 2008.

[9] Simon Thompson, Peter R. King, and Helen Cameron. Modelling reactive multimedia: Design and authoring. *Multimedia Tools and Applications*, 27(1):23–52, 2005.

[10] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ICFP '97, pages 263–273, New York, NY, USA, 1997. ACM.

[11] C. Elliot J. Peterson and G.S. Ling. Fran users manual. available from http://www.haskell.org/fran, 2000.

[12] Peter King, Patrick Schmitz, and Simon Thompson. Behavioral reactivity and real time programming in xml: functional programming meets smil animation. In *Proceedings of the 2004 ACM symposium on Document engineering*, DocEng '04, pages 57–66, New York, NY, USA, 2004. ACM.

[13] Brian W. Kernigan and Rob Pike. *The UNIX Programming Envrionment.* Prentice-Hall, 1982.

[14] Jacek Gwizdka. "what's in the context?". In *GVU Center, Georgia University of Technology, Research report*, pages 1–6, 2000.

[15] J. Call T. Behne M. Tomasello, M. Carpenter and H. Moll. Understanding and sharing intentions: The origins of cultural cognition. *Behavioral and Brain Sciences.*, 28(5):675–691, 2005.

[16] Philip R. Cohen and Hector J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(2–3):213 – 261, 1990.

[17] Qiang Yang. *Intelligent Planning. A Decomposition and Abstraction Based Approach.* Springer Verlag, 1997.

[18] Nau D. S. Ghallab, M. and P. Traverso. *Automated Planning: Theory and Practice.* Morgan Kaufmann, 2004.

[19] Ian Horrocks. Ontologies and the semantic web. *Commun. ACM*, 51(12):58–67, December 2008.

[20] Deborah L. McGuinness Franz Baader, Diego Calvanese and Daniele Nardi. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2010.

[21] Saul A. Kripke. Semantical analysis of modal logic i normal modal propositional calculi. *Mathematical Logic Quarterly*, 9(5-6):67–96, 1963.

[22] Saul A. Kripke. Semantic analysis of modal logic ii: Non-normal modal propositional calculi. *Symposium on the Theory of Models*, 1965.

[23] J.-J Ch. Meyer and W. van der Hoek. *Epistemic Logic for AI and Computer Science, Cambridge Tracts in Theoretical Computer Science, vol. 41*. Cambridge University Press, 1995.

[24] Orna Grumberg Edmund M. Clarke, Jr. and Doron A. Peled. *Model Checking*. MIT press, 1999.

[25] Yoram Moses Ronald Fagin, Joseph Y. Halpern and Moshe Y. Vardi. *Reasoning about Knowledge*. MIT Press, 1995.

[26] Michael Wooldridge. *Reasoning about Rational Agents*. MIT press, 2000.

[27] Michael Wooldridge. *An Introduction to MultiAgent Systems*. Wiley, 2009.

[28] Richard Sheer. The mental state theory of intentions. *Philosophy*, 79:121–131, 2004.

[29] Richard Kelley, Alireza Tavakkoli, Christopher King, Monica Nicolescu, Mircea Nicolescu, and George Bebis. Understanding human intentions via hidden markov models in autonomous mobile robots. In *Proceedings of the 3rd ACM/IEEE international conference on Human robot interaction*, HRI '08, pages 367–374, New York, NY, USA, 2008. ACM.

[30] A. Cichocki, Y. Washizawa, T. Rutkowski, H. Bakardjian, Anh-Huy Phan, Seungjin Choi, Hyekyoung Lee, Qibin Zhao, Liqing Zhang, and Yuanqing Li. Noninvasive bcis: Multiway signal-processing array decompositions. *Computer*, 41(10):34–42, oct. 2008.

[31] M. Adcock, Jae-Woo Chung, and C. Schmandt. Are we there yet? user-centered temporal awareness. *Computer*, 42(2):97–99, feb. 2009.

[32] M. Makatchev and S.K. Tso. Human-robot interface using agents communicating in an xml-based markup language. In *Robot and Human Interactive Communication, 2000. RO-MAN 2000. Proceedings. 9th IEEE International Workshop on*, pages 270–275, 2000.

[33] Haruo Hosoya and Benjamin C. Pierce. Xduce: A statically typed xml processing language. *ACM Trans. Internet Technol.*, 3(2):117–148, May 2003.

[34] J. Clark and M. Murata. Relax ng. http://www.relaxng.org, 2001.

[35] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[36] Donald E. Knuth. Correction: Semantics of context-free languages. *Mathematical Systems Theory*, 5(1):95–96, 1971.

[37] Mehdi Jazayeri, William F. Ogden, and William C. Rounds. The intrinsically exponential complexity of the circularity problem for attribute grammars. *Commun. ACM*, 18(12):697–706, December 1975.

[38] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 154–173. Springer Berlin / Heidelberg, 1987.

[39] D. Etienne P. Didier, R. Gilles and J. Martin. Attribute grammars: A declarative functional language. Research Report of INRIA: the French National Institute for Research in Computer Science and Control, October, 1995.

[40] Didier Parigot, Gilles Roussel, Martin Jourdan, and Étienne Duris. Dynamic attribute grammars. In Herbert Kuchen and S. Doaitse Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 1140 of *Lecture Notes in Computer Science*, pages 122–136. Springer Berlin / Heidelberg, 1996.

[41] Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Attribute grammars and functional programming deforestation. In Pascal Van Hentenryck, editor, *Static Analysis*, volume 1302 of *Lecture Notes in Computer Science*, pages 2–97. Springer Berlin / Heidelberg, 1997.

[42] Simon Thompson. *Haskell: The Craft of Functional Programming,* 3^{rd} *edition.* Addison Wesley, New York, 2011.

[43] A.J. Gordon. *Functional Programming and Input/Output. British Computer Society Distinguished Dissertations in Computer Science.* Cambridge University Press, 1994.

[44] Tofte M. Harper R. Milner, R. and D. MacQueen. *The Definition of Standard ML (revised edition).* MIT Press, 1997.

[45] Chris Smith. *Programming F♯.* O'Reilly, 2009.

[46] Christopher Strachey. Towards a formal semantics. *Formal Description Languages for Computer Programming*, pages 198–220, 1966.

[47] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13:11–49, 2000.

[48] D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. *Programming Research Group Technical Monograph PRG-6, Oxford University Computing Lab*, pages 1–43, 1971.

[49] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming. Originally Printed as Technical report FN-19. Computer Science Department, Åarhus University.*, 60-61:17–139, 2004.

[50] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[51] Krzysztof R. Apt. Ten years of hoare's logic: A survey part i. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, October 1981.

[52] Benjamin pierce. *Types and Programming Languages*. MIT Press, Cambridge, 2002.

[53] G. Kahn. Natural semantics. In Franz Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer Berlin / Heidelberg, 1987. 10.1007/BFb0039592.

[54] Robin Milner. *A Calculus of Communicating Systems, Lecture Notes in Computer Science, Vol. 92*. Springer-Verlag, 1980.

[55] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[56] Robin Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.

[57] Dan Scott. Some reflections on strachey and his work. *Higher-Order and Symbolic Computation*, 13:103–114, 2000.

[58] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. The MIT Press, 1992.

[59] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.

[60] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-Verlag, 2007.

[61] http://www.w3.org/XML/.

[62] Edsger W Dijkstra. On the role of scientific thought. In *Selected writings on Computing: A Personal Perspective*, pages 60–66, 1982.

[63] Thorsten Büring, Jens Gerken, and Harald Reiterer. Usability of overview-supported zooming on small screens with regard to individual differences in spatial ability. In *Proceedings of the working conference on Advanced visual interfaces*, AVI '06, pages 233–240, New York, NY, USA, 2006. ACM.

[64] Kim J. Vicente and Robert C. Williges. Accommodating individual differences in searching a hierarchical file system. *International Journal of Man-Machine Studies*, 29(6):647–668, 1988.

[65] D. Benyon. Accommodating individual differences through an adaptive user interface. *HUMAN FACTORS IN INFORMATION TECHNOLOGY*, 10:1–149, 1993.

[66] Alan F. Newell and Peter Gregor. "user sensitive inclusive design" — in search of a new paradigm. In *Proceedings on the 2000 conference on Universal Usability*, CUU '00, pages 39–44, New York, NY, USA, 2000. ACM.

[67] Simeon Keates, P John Clarkson, and Peter Robinson. Developing a practical inclusive interface design approach. *Interacting with Computers*, 14(4):271–299, 2002.

[68] Matthew Pattison and Alex Stedmon. Inclusive design and human factors: Designing mobile phones for older users. *PsychNology Journal*, 4(3):267–284, 2006.

[69] Krish Ramachandran. Adaptive user interfaces for health care applications. Available at http://ibm.com/developerWorks, 2009.

[70] Leah Findlater and Joanna McGrenere. A comparison of static, adaptive, and adaptable menus. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '04, pages 89–96, New York, NY, USA, 2004. ACM.

[71] B. Kules. User modeling for adaptive and adaptable software system. Available at http://otal.umd.edu/UUGuide/wmk/, 2000.

[72] S. Rogers, C.-N. Fiechter, and C. Thompson. Adaptive user interfaces for automotive environments. In *Intelligent Vehicles Symposium, 2000. IV 2000. Proceedings of the IEEE*, pages 662–667, 2000.

[73] Jiming Liu, Chi Kuen Wong, and Ka Keung Hui. An adaptive user interface based on personalized learning. *Intelligent Systems, IEEE*, 18(2):52–57, mar-apr 2003.

[74] Marko Balabanovic. Exploring versus exploiting when learning user models for text recommendation. *User Modeling and User-Adapted Interaction*, 8:71–102, 1998.

[75] Pat Langley. User modeling in adaptive interfaces. In *PROCEEDINGS OF THE SEVENTH INTERNATIONAL CONFERENCE ON USER MODELING*, pages 357–370. Springer, 1999.

[76] Eric Horvitz, Jack Breese, David Heckerman, David Hovel, and Koos Rommelse. The lumière project: Bayesian user modeling for inferring the goals and needs of software users. In *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*, UAI'98, pages 256–265, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[77] Krzysztof Z. Gajos, Mary Czerwinski, Desney S. Tan, and Daniel S. Weld. Exploring the design space for adaptive graphical user interfaces. In *Proceedings of the working conference on Advanced visual interfaces*, AVI '06, pages 201–208, New York, NY, USA, 2006. ACM.

[78] Anil Shankar, Sushil J. Louis, Sergiu Dascalu, Linda J. Hayes, and Ramona Houmanfar. User-context for adaptive user interfaces. In *Proceedings of the 12th international conference on Intelligent user interfaces*, IUI '07, pages 321–324, New York, NY, USA, 2007. ACM.

[79] Gianni Viano, Andrea Parodi, James Alty, Chris Khalil, Inaki Angulo, Daniele Biglino, Michel Crampes, Christophe Vaudry, Veronique Daurensan, and Philippe Lachaud. Adaptive user interface for process control based on multi-agent approach. In *Proceedings of the working conference on Advanced visual interfaces*, AVI '00, pages 201–204, New York, NY, USA, 2000. ACM.

[80] T. Tsandilas and M. C. Schraefel. Usable adaptive hypermedia systems. *New Review of Hypermedia and Multimedia*, 10(1):5–29, 2004.

[81] http://http://javacc.java.net/.