WEFLOW: WE FOLLOW THE FLOW

by

Nadin Kökciyan

B.S., Computer Engineering, Galatasaray University, 2009

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering
Boğaziçi University

2011

WEFLOW: WE FOLLOW THE FLOW

APPROVED BY:

Suzan Üsküdarlı, Ph.D. .....
(Thesis Supervisor)

Assoc. Prof. Haluk Bingöl .....

Assoc. Prof. Yağmur Denizhan .....

DATE OF APPROVAL: 19.12.2011

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Suzan Üsküdarlı for her supervision and guidance. Your suggestions and encouragement helped me a lot to complete this initial step in our long journey. I am lucky enough to have had a great supervisor like you. Thank you for being there when I needed you the most.

TB Dinesh.. I will never forget your endless support. We have still so much to do! It has always been a pleasure working with you. Thank you for inviting me to India and to make a big change in my life. I had a great time as an intern in Servelots where all this story began..

I am also really grateful to Prof. L.G.L.T. Meertens and members of SoSLab (Complex Systems Lab at Bogazici University) for their feedback and contributions during this work. This work is partially funded by B.U. Research Fund (BAP5709) and Turkcell Akademi.

Finally, I am very thankful to my dear sister: Karin Taşkın, my family, and my dear friends: Seda Çelebican, Swapna Gangadharan, Burcu Akar for their continual encouragement and support.

ABSTRACT

WEFLOW: WE FOLLOW THE FLOW

Web applications, such as social networking platforms and Web X.0 applications have transformed average users from consumers to producers of content. While this transition has been very successful with respect to content, the same can not be said for behavior generation. Non-savvy Web users have practically no ability to introduce any behavior on the Web. The computational powers offered by the Web are limited to those who are in the know or can afford to develop applications. Yet, there are many simple applications that average users could conceive and utilize if they were empowered to introduce behavior. This work aims to empower average web users with the ability to create simple web applications for purposeful communities. We suggest an environment with information and processes specific to a communities needs, as opposed to the ad hoc information sharing and coordination achieved via social networking platforms, in order to retain the long term value of the generated information. An interesting class of Web applications are human computation applications, where tasks are distributed among humans and computers based on their suitability for performing those tasks. Web based human computation applications seem appropriate for purposeful communities. This thesis presents a framework for the development of human computation applications for purposeful communities. In order to create such an environment, we propose a web based workflow framework. The WeFlow Framework, supports the specification, generation, and execution of community specific virtual environments. These environments are based on a workflow model, which are defined by the communities that use them. The environment is based on a workflow model, which consists of tasks, control and data flow among tasks, and people who perform those tasks. A WeFlow framework prototype is shown by examples and case studies.

ÖZET

WEFLOW: AKIŞ NEREDE BİZ ORADA

Sosyal ağ platformları ve Web X.0 gibi Web uygulamaları, ortalama Web kullanıcılarını bilgi tüketicileri sıfatından içerik yaratan kimlikler haline getirmişlerdir. İçerik bakımından düşünülürse bu geçiş başarılı gözükse de, aynı şeyi uygulama gidişatı oluşumu konusunda söylenemez. Ortalama Web kullanıcıları, Web üzerinde uygulama davranışlarını tanımlayacak bilgi birikimine sahip değillerdir. Web'in hesaplamaya yönelik kuvvetli tarafları, Web bilgi birikimine sahip bilgi teknolojilerini bilen kişilerce kullanılabilmektedir. Çok basit Web uygulamarı bile ortalama Web kullanıcıları tarafından tanımlanamamaktadır. Bir amaca yönelik bir araya gelen topluluklarda basit Web uygulamaları geliştirilmesi üzere WeFlow platformu geliştirilmiştir. Her topluluğun farklı hedefleri, farklı veri gereksinimleri ve farklı veri işleme kuralları vardır. Bu tür gereksinimler mevcut sosyal ağ platformları ile kısmi olarak gerçeklenirken, WeFlow uzun vadede oluşturulan içerikten anlam çıkarabilmeyi hedeflemektedir. Enteresan bir diğer uygulamalar insan gücünün kullanıldığı Human Computation alanıdır. Bu kapsamda, bir iş küçük parçalar haline getirilerek bir akış şeklinde ifade edilmektedir. Bu iş parçaları arasındaki sıra ve verinin akışı, bu iş parçalarının kimler tarafından yapılacağı tanımlanmaktadır. Amaç bir işi gerçeklemek ve gerekli verinin insan gücü kullanılarak bir araya getirilmesidir. Bu çalışmada, WeFlow isimli iş akışı modelimizi öneriyoruz. Sanal ortamdaki kullanıcılar, bu modeli kullanarak Web uygulamalarının nasıl davranması gerektiğini tanımlayabilecekler. İzlenen yol: (i) iş akışı tanımını oluşturmak, (ii) iş akışı tanımı doğrultusunda, Web uygulaması üretmek, (iii) iş akışı tanımı doğrultusunda, Web uygulamasını yürütmek, izlemek ve yönetmektir.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . iii
ABSTRACT . . . iv
ÖZET . . . v
LIST OF FIGURES . . . x
LIST OF TABLES . . . xvii
LIST OF ACRONYMS/ABBREVIATIONS . . . xviii
1. INTRODUCTION . . . 1
2. BACKGROUND . . . 5
2.1. Web Evolution . . . 5
2.2. Social Web Applications . . . 6
2.3. Virtual Communities . . . 8
2.4. Human Computation . . . 8
2.4.1. Genres of HCOMP . . . 9
2.4.2. Main Incentives to Participation . . . 10
2.5. A Brief Overview of Workflows . . . 11
3. RELATED WORK . . . 12
3.1. Sociological Aspect . . . 12
3.2. Pantoto Project . . . 12
3.3. Facebook API . . . 14
3.4. Workflow Management Systems . . . 14
3.4.1. Workflow Languages . . . 16
3.5. Yahoo! Pipes . . . 17
4. PROPOSED MODEL . . . 21
4.1. WeFlow Components . . . 22
4.2. Terminology . . . 22
4.3. Workflow . . . 23
4.4. Task . . . 24
4.5. WeFlow Task Types . . . 27
4.5.1. Basic Task . . . 27

4.5.2. Conditional Task 29
4.5.3. DoAll Task 30
4.5.4. Repetition Task 30
4.5.5. Collective Task 32
4.5.6. Composite Task 34
4.5.7. Data Task 35
4.6. WeFlow People 35
4.6.1. Role Based 35
4.6.2. Individual Based 35
4.7. WeFlow Flow Aspects 36
4.7.1. Control Flow 36
4.7.2. Data Flow 37
4.7.2.1. Input Mappings 37
4.7.2.2. Human Input Mappings 40
4.7.2.3. Output Mappings 42
4.8. WeFlow Application Generator 46
4.8.1. Mapping Data Types to HTML Elements 47
4.8.2. Representing a Human Task on Web 47
4.8.3. Generating a Web Application 49
4.9. WeFlow Application Execution 49
4.9.1. Data Layer 49
4.9.2. Web Server 50
4.9.3. WeFlow Execution Engine 50
4.9.3.1. Instantiation 50
4.9.3.2. State Handling 50
4.9.3.3. Control Flow Handling 50
4.9.3.4. Task Handling 50
5. IMPLEMENTATION 51
5.1. WeFlow Specification Handler 51
5.1.1. WeFlow Specification Language 52
5.1.2. General Description 53

5.1.2.1. Tasks 54
5.1.2.2. Basic Task 56
5.1.2.3. Conditional Task 56
5.1.2.4. DoAll Task 60
5.1.2.5. Repetition Task 61
5.1.2.6. Collective Task 63
5.1.2.7. Composite Task 64
5.1.3. Control Flow Description 65
5.1.4. Data Flow Description 65
5.1.5. Human / Groups Description 68
5.1.6. Specification Handler Methods 70
5.2. WeFlow Data Handler 71
5.2.1. A Python Object Database: ZODB 71
5.2.2. Specification Data Handler 71
5.2.3. Application Data Handler 72
5.3. WeFlow User Handler 72
5.4. WeFlow Application Generator 72
5.4.1. HTML Data Generator 73
5.4.2. Application Template Generator 73
5.5. WeFlow Execution Module 74
5.5.1. State Handler 75
5.5.2. Task Handler 75
5.5.3. Control Flow Handler 75
5.5.4. Instantiator 75
5.6. WeFlow Tasklist Handler 79
5.7. WeFlow Administration 79
5.8. Logger 79
6. USE CASES 80
6.1. A Human Computation Web Application: CoStory 80
6.1.1. CoStory Specification 80
6.1.2. CoStory Web Application Generation 81

6.1.3. CoStory Web Application Execution 81
6.2. A Human Computation Web Application: Hisarustu Accessibility . . . 83
6.2.1. Hisarustu Accessibility Specification 83
6.2.2. Hisarustu Accessibility Web Application Generation 85
6.2.3. Rumelihisarustu Web Application Execution 85
6.3. A Simple Web Application for Personal Use: Jack Summing Integers . 86
7. DISCUSSION AND FUTURE WORK 92
8. CONCLUSION 94
APPENDIX A: WEFLOW SPECIFICATION FOR COSTORY 95
REFERENCES 99

LIST OF FIGURES

Figure 2.1. Evolution of Web. 5

Figure 2.2. reCaptcha Example. 11

Figure 2.3. Future Sophisticated Captchas. 11

Figure 3.1. A Facebook Application: PhotoGrab. 15

Figure 3.2. Top 10 Movies Trailer. 19

Figure 3.3. Top 10 Movies Trailer Output. 20

Figure 4.1. WeFlow Framework. 22

Figure 4.2. An Example Workflow. 24

Figure 4.3. A Generic Task. 27

Figure 4.4. Basic Task. 28

Figure 4.5. IfElse Task. 29

Figure 4.6. Case Task. 30

Figure 4.7. DoAll Task. 31

Figure 4.8. A Generic Repetition Task. 32

Figure 4.9. Semantics of Repetition Task t_1. 33

Figure 4.10. Control Flow in a Repetition Task. 33

Figure 4.11. A Generic Collective Task. 34

Figure 4.12. Composite Task. 34

Figure 4.13. Control Flow Example 1. 37

Figure 4.14. Task Input Mappings. 38

Figure 4.15. Specification of Two Tasks. 39

Figure 4.16. After execution of *Generate Numbers* Task. 39

Figure 4.17. Data Mappings between Two Tasks. 39

Figure 4.18. Mapping Data to *Sum Integers* Task. 39

Figure 4.19. Supertask Input Mappings. 40

Figure 4.20. Human Input Mappings. 41

Figure 4.21. *put* Channel Operation. 42

Figure 4.22. *get* Channel Operation. 42

Figure 4.23. *Sum Integers* Task. 42

Figure 4.24. Task Performer Providing Data. 43

Figure 4.25. Outmappings within a Task. 43

Figure 4.26. Outmappings within a Task. 44

Figure 4.27. Outmappings in a Repetition Task. 44

Figure 4.28. Outmappings in a Collective Task. 45

Figure 4.29. Outmappings in a DoAll Task. 45

Figure 4.30. Sum Integers Task Mapping. 46

Figure 4.31. Sum Integers Task after mapping. 46

Figure 4.32. Tag a Bird Picture Human Task. 48

Figure 4.33. WeFlow Web Application Stack. 49

Figure 5.1. WeFlow Implementation Architecture. 52

Figure 5.2. XML Representation Example. 53

Figure 5.3. WeFlow Specification Structure. 53

Figure 5.4. Defining Workflow Information. 54

Figure 5.5. Defining Workflow Information Example. 54

Figure 5.6. WeFlow Specification Structure with Tasks. 55

Figure 5.7. WeFlow Specification: Task. 56

Figure 5.8. WeFlow Specification: Basic Task. 56

Figure 5.9. WeFlow Specification: IfElse Task Skeleton. 57

Figure 5.10. WeFlow Specification: DoAll Task. 58

Figure 5.11. WeFlow Specification: Choice Task Skeleton. 59

Figure 5.12. WeFlow Specification: Choice Task. 60

Figure 5.13. WeFlow Specification: DoAll Task Skeleton. 61

Figure 5.14. WeFlow Specification: DoAll Task. 62

Figure 5.15. WeFlow Specification: Repetition Task Skeleton. 62

Figure 5.16. WeFlow Specification: Repetition Task. 63

Figure 5.17. WeFlow Specification: Collective Task Skeleton. 64

Figure 5.18. WeFlow Specification: Collective Task. 64

Figure 5.19. WeFlow Specification: Control Flow Skeleton. 65

Figure 5.20. WeFlow Specification: Control Flow. 65

Figure 5.21. WeFlow Specification: Data Flow Skeleton. 66

Figure 5.22. WeFlow Specification:Data Flow Input Mappings Skeleton. 67

Figure 5.23. WeFlow Specification:Data Flow Input Mappings. 67

Figure 5.24. WeFlow Specification:Data Flow Output Mappings Skeleton. . . . 68

Figure 5.25. WeFlow Specification:Data Flow Output Mappings. 69

Figure 5.26. WeFlow Specification:Resourcing Skeleton. 69

Figure 5.27. WeFlow Specification:Resourcing. 70

Figure 5.28. Algorithm for *app_state_handler* method. 76

Figure 5.29. Algorithm for *call_task* method. 77

Figure 5.30. Algorithm for *get_next_construct* method. 78

Figure 6.1. CoStory – A Human Computation Web Application. 81

Figure 6.2. CoStory – Starting the Web Application. 82

Figure 6.3. CoStory – Starting a Story. 82

Figure 6.4. CoStory – Updating a Story. 83

Figure 6.5. CoStory – Finishing a Story. 84

Figure 6.6. CoStory – Displaying Finished Story. 84

Figure 6.7. Rumelihisarustu Specification. 85

Figure 6.8. Rumelihisarustu – Workflow Ready to Start. 87

Figure 6.9. Rumelihisarustu – Choose Task. 87

Figure 6.10. Rumelihisarustu – Share Item. 87

Figure 6.11. Rumelihisarustu – Show Items. 88

Figure 6.12. Rumelihisarustu – Update Item. 88

Figure 6.13. Rumelihisarustu – Control Item. 88

Figure 6.14. Rumelihisarustu – Show Item. 89

Figure 6.15. Rumelihisarustu – List Items. 89

Figure 6.16. Rumelihisarustu – Show Items II. 89

Figure 6.17. Jack – summing integers. 89

Figure 6.18. Code for Generate Numbers Task. 90

Figure 6.19. Code for Compute Sum Task. 90

Figure 6.20. Code for Compare Task. 90

Figure 6.21. Jack – Do Sum Operation. 90

Figure 6.22. Jack provides an answer. 91

Figure 6.23. Jack – Notify User Task. 91

Figure A.1. Workflow Information for for CoStory. 95

Figure A.2. Task Information for CoStory. 96

Figure A.3. Resourcing Information for CoStory. 97

Figure A.4. Mappings Information for CoStory. 98

LIST OF TABLES

Table 4.1. WeFlow Data Types. 25

Table 4.2. Task Types with Corresponding Shapes. 28

Table 4.3. Mapping Data Types to HTML Elements. 47

Table 5.1. Mapping Data Types to web.py Form Elements. 73

LIST OF ACRONYMS/ABBREVIATIONS

API	Application Programming Interface
CAPTCHA	Completely Automated Public Turing test to tell Computers and Humans Apart
CMS	Content Management System
FOSCMS	Free and Open Source Content Management System
GWAP	Game with a Purpose
HCOMP	Human Computation
HCR	Human Character Recognition
HTML	HyperText Markup Language
IT	Information Technology
KCVC	Knowledge Collection from Volunteer Contributors
NGO	Non-Governmental Organization
OCR	Optical Character Recognition
WWW	World Wide Web

1. INTRODUCTION

The use of Web is so integral to daily life in many regions of the world that not having access to it would seriously limit the quality of life. In other regions more and more people are getting connected. The Web has changed how people access services such shopping, banking, entertaining themselves, communicating, learning and much more. Web users have been transformed from mere consumers to also being producers of content thanks to Web 2.0 applications. Over time, the Web experience became interactive. In another words, Web 2.0 applications were interacting with other Web 2.0 applications in order to interchange data and also with users willing to share content.

Virtual communities are social groups where people forms webs of personal relationships in cyberspace [1]. Members of virtual communities share common interests, ideas and feelings through communication platforms. Communication platforms such as online groups, blogs [2], wikis [3], Facebook[1] and Twitter[2] provide means for contributing (sharing, rating, tagging) multimedia content, creating discussions and organizing activities. The nature of how people use these applications greatly vary. Example uses are chit-chat style communication, serious discussions, accessing information, disseminating information and organizing activities.

Initial Web users primary served as consumers of content and services. They were just looking at static webpages, and browsing from one page to another without making any contribution to webpages. Web 2.0 applications changed the nature of Web by making it participatory and inclusive such that there was an explosive growth in the number of users, i.e. content producers. Moreover, end users have began producing content at an increasing rate.

User participation and collective activity are increasingly handled via Web applications. As the Web is a distributed environment, Web applications enable commu-

[1] http://www.facebook.com
[2] http://www.twitter.com

nication among distributed users. In another words, users share information through Web applications which distribute this information to other users. By using Web applications, a user could keep track of the most recent activities (notifications, invites, subscriptions, friend requests) in the community, discuss on a specific topic, share multimedia content (pictures, videos, music) and arrange meetings. Beyond these activities, it is important to accomplish more complex activities that users can help computers to create solutions for coordinating, communicating and managing collective efforts [4, 5] because there is a need for tools that help creating applications appealing purposeful communities. Such tools may be used to utter the seemingly most trivial minutiae to coordinating highly useful disaster response activities.

While the transition from a consumer to a producer of Web content has been quite successful, there remain some serious shortcomings. For one, users have little control over the structure and access to their contributions. In another words, their contributions are not presented in a structured manner but in a free form. And thus, it makes difficult to find relevant information, manage information and make any sense of it. While users can make content contributions, they can rarely define syntactic or semantic aspects of this information. Furthermore, they have virtually no impact on how their contributions get used. For random conversations these limitations may be of no consequence, however, for communities of purpose these limitations greatly diminish the utility of the Web, since its computational powers are not accessible to the end user. There is a strong need for powerful and flexible environments, customizable to the culture, skills and needs of very diverse non IT savvy people [6].

The Web is a great platform for distributed computation, i.e. solving a problem in a collaborative way. A problem may be decomposed in subproblems and distributed to various Web participants to make it solved. Web participants may be human or automated agents. Moreover, the inclusion of humans in the computation cycle have been introduced by many collaborative applications like Wikipedia as well as human computation applications [7, 8]. These applications demonstrate the ability to achieve impressive results by including humans in the computation cycle (since they are so good at performing impressive tasks). These applications, however, were conceived

and implemented by pretty Information Technology (IT) savvy people. End users who are not IT savvy have virtually no influence on the behavior of Web, in that they can not define applications. Obviously, programming is not a trivial task even for IT savvy people. In [9], Nardi defines the semantics of end-user programming as empowering end-users with the ability to customize and adapt a software system according to their needs in order to increase efficiency of the work being performed. Allowing to realize end-user programming systems with minimal effort and in a natural way is subject to active research [6, 10, 11].

Most recently communication through Facebook and Twitter have demonstrated the appeal of such platforms to create solutions for coordinating, communicating and managing collective efforts. However, these efforts are usually ad hoc, address present time and excessively rely on human computation. An interesting class of Web applications are human computation applications, where tasks are distributed among humans and computers based on their suitability for performing those tasks. Since users already make use of social networking systems to communicate and coordinate activities such as responding to disasters and coordinating activities, support for user generated human computation applications seems very appropriate. This study presents a framework for user generated human computation applications. Unlike the coordination of activities via social networking platforms, such applications are dedicated to a specific purpose, support the acquisition of application-specific data, and coordinate participation.

In this work, a community defined workflow model is developed as it is not always possible to find IT specialists to work with, especially for small communities. Additionally, it is not always desired to get support from IT specialists who may perform incorrectly due to their lack of domain knowledge. Non IT savvy people need to create collaborative environments, accomplish required tasks on their own, create and manage knowledge for their needs. And thus, it is important to empower non IT savvy people with the ability to define Web application behavior.

In this study, mainly focus is on the area of Human Computation, i.e. users are solving a problem by collaborating with computers. While human computation

is helpful in some tasks, it is not the most effective in many others. Some computations, which take time for humans to solve, can be easily carried out by computers and/or vice versa. However, in addition to a communication platform, there is little support for enabling a technology to define computational processes which are required to reach targeted goals. Hence, a workflow model of specifying human computation applications for the Web, WeFlow, is proposed that can be used by virtual community members to specify Web application behavior desired in their community. Both human and computer processes are involved in this model. Through the use of such specifications, community members define ad hoc workflows in a web-based environment to create, manage and accomplish tasks according to their community needs. Given such specifications, Web applications can be generated preserving the semantics of those specifications. WeFlow model helps online communities to use the Web for collective activity and it empowers Web users to carry out envisioned applications that would serve their needs.

In proposed approach, a workflow is defined in terms of the tasks, the people, and the control/data flow among tasks. The framework, WeFlow, proposes a human computation specification language, an application generator, and an execution engine for running applications. A prototype is implemented and case studies using that prototype are developed for purposes of demonstration.

The remainder of the thesis proceeds as follows. In Chapter 2, background information related to this study is explained. In Chapter 3, related work from various fields is provided. In Chapter 4, proposed model is detailed with various examples. In Chapter 5, WeFlow framework implementation is explained in terms of its components. In Chapter 6, two human computation web application use cases are presented. In Chapter 7, a brief discussion of future work is introduced. And finally, conclusion of this study is provided in Chapter 8.

2. BACKGROUND

This chapter provides an overview of the important technologies that are relevant to this work. Section 2.1 gives a brief description of the Web and explains its current use. Section 2.3 describes virtual communities and the Social Web. Section 2.4 discusses human computation and its benefits with some examples. Section 2.5 briefly describes the workflow concept.

2.1. Web Evolution

Web is a commonly used as an abbrevation for World Wide Web (WWW). Web consists of a huge set of hyperlinked documents. Figure 2.1 shows the evolution of Web and its associated technologies.

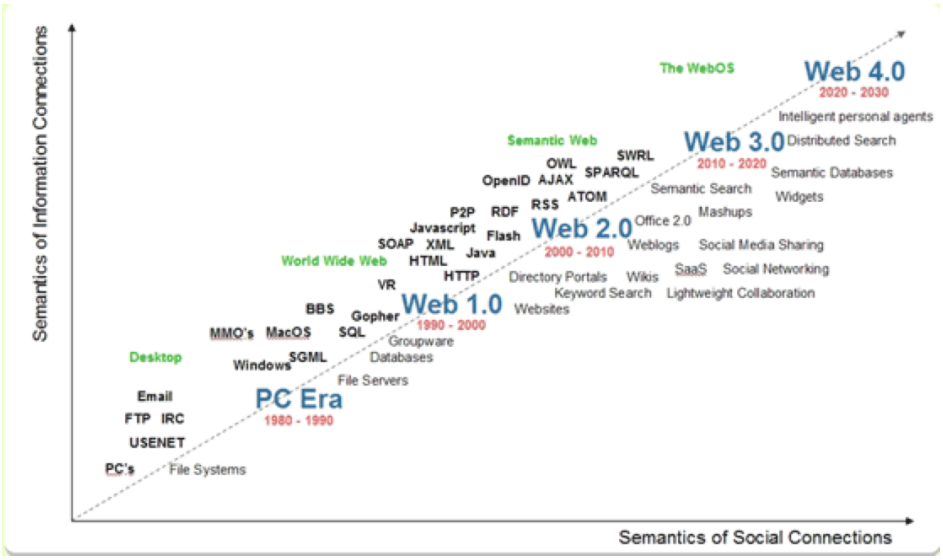


Figure 2.1. Evolution of Web [12].

The Web started by hosting hyperlinked documents. The documents were written with HTML (Hypertext Markup Language), which enabled the description of document structures and links to other HTML documents. A collection of such documents formed web sites, which often belonged to institutions and companies. These documents were

static, stored on Web and was presented to interested audience through Web browsers. Users passively consumed content in these pages. The information offered by these webpages was a read-only information for web users, and could only be modified by owner of these webpages. And this idea became obsolete once Web 2.0 emerged.

With Web 2.0 (Social Web), the Web became a participatory platform where users can join, create and share information in a collaborative and collective way in virtual communities. So interactivity is a key point of this stage of WWW. Rather than being consumers of information, users became creators of user-generated content and also consumers of their information, i.e. users are prosumers now. Most commonly known examples are social networking sites, blogs, wikis, video and/or picture sharing sites and so on.

With Web 3.0 (Semantic Web), the goal is to provide information not only for people but also for machines. One of the big problems that Web users encounter with is the overload of information on Web, and it is difficult to find only what a Web user is really looking for. Web 3.0 offers a machine readable format to query Web in a more efficient way. Current stage is Web 3.0, and geeks already started to envision about Web 4.0 which is predicted to be about avatar-based virtualization, i.e. creating representatives of people on virtual worlds [13].

2.2. Social Web Applications

With Web 2.0 (Social Web), the Web became a participatory platform where users can join, create and share information in a collaborative and collective way. Social web applications are widely used by end users, e.g., Facebook is one of the biggest virtual communities and has more than 750 million active users, as of August 2011 [14, 15]. Furthermore, community members use these applications to meet their community needs. For example, they coordinate activities to deliver goods for earthquake victims, organize meetings, broadcast useful information to the world and so on. Social web applications such Facebook, Twitter are for general use and offer limited capabilities to search and process information. Thus, they are not sufficient to retain the long term

value of the generated information useful for purposeful communities.

There are other online social platforms that promote collaboration and collective work. Social network applications such as groups Yahoo groups[3] , Google Groups[4] , blogs [2], wikis [3] are part of daily life to collaborate and publish content. As the Web enables a distributed platform, all these applications enjoy the benefit of this platform. The high usability of these applications shows that such systems are successful and promote sharing and collaboration in communities.

Groups are online spaces where people with a shared interest meet and communicate over email and on the Web. Google Groups and Yahoo Groups are free and popular in this field and they provide a repository to upload documents, photos etc., archive all shared messages and enable people to organize events. Blogs are online personal spaces where people update it from time to time in order to publish content without doing any programming. A research conducted by A. Nardi [2] shows that blogs are mostly written by ordinary people. Wikis are websites that allow people to create and update any number of web pages. Only the current version of a web page is displayed, and all pages are owned by group of people rather than individuals. There are various wiki systems and one of them is Semantic Wiki [16] which enables to structure data within a page and the relationships between pages resulting in semantic processing.

Social networking applications like Facebook [15] and Twitter have further fueled user participation [17]. The nature of how people use these applications greatly vary. For example, some use them for chit-chat style communication, some for accessing or disseminating information, some for organizing activities. The use of Facebook and Twitter has increased the rate and quantity to an incredible level. Such tools are used to utter the seemingly most trivial minutiae to coordinating highly useful disaster response activities [4, 5].

[3]http://groups.yahoo.com/
[4]http://groups.google.com/

On the other hand, Virtual Communities are dedicated to specific topics. Ad hoc topics are specified by purposeful communities [1], and community members discuss about specific contexts. As other Social Web Applications, Virtual Communities are useful to get short term value of the generated content. Virtual Communities are discussed in next section.

2.3. Virtual Communities

In Section 2.1, concept of Social Web is discussed briefly. People want to pursue mutual goals or interests and this is why they tend to be part of virtual communities. Virtual communities are social groups where people start public discussions which form webs of personal relationships in cyberspace [1]. Virtual communities are just like real life communities because they both provide support, information, friendship and acceptance between people who mostly do not know each other previously [18]. Virtual Communities are usually dispersed geographically, and depend on social interaction between community members.

Virtual Communities have their own online spaces to communicate with other members. Such online spaces are developed by IT skilled people according to community needs. Members are mostly talking about specific contexts such birdwatching, cuisine, snakes and so on. In all these contexts, there is an ad hoc vocabulary that is used or emerged over time.

2.4. Human Computation

HCOMP is an abbreviation used for Human Computation in Human Computation workshops. HCOMP is a computational process in which some steps are performed by humans. By harnessing human time and energy, large-scale computational problems that are challenging for computer programs but trivial for humans are solved [8].

In traditional computation, humans are not part of the computational process, i.e. they are the ones providing formalized problem descriptions to computers in order

to receive solution(s) for that specific problem [19].

HCOMP is a type of [20] :

- *Collective Intelligence:* In the context of HCOMP, a computational process is performed by a group of people, i.e. a solution for a problem is obtained by collaborative efforts of many people doing things that seem intelligent [21].
- *Crowdsourcing:* HCOMP technique is outsourcing tasks to a group of people.

2.4.1. Genres of HCOMP

HCOMP is widely applied to many fields, to get a better understanding of HCOMP, some genres of HCOMP are listed below [22]:

- *GWAP* Players perform some computation in order to get a score. So GWAP [5] players are motivated by the fun the game provides, but they also provide useful computation as a side effect. Luis von Ahn published the first work on GWAP in his PH.D. thesis.
- *Crowdsourcing:* Unpaid volunteers do explicitly defined tasks, and there is no obligation for volunteers to continue a task.
- *Mechanized Labor:* Contrary to crowdsourcing, volunteers are paid to do some predefined tasks.
- *Wisdom of Crowds:* Crowd intelligence is the key concept. Different individual judgments are aggregated in order to make a correct judgment.
- *Grand Search:* In this genre, people are supposed to find a correct answer solving a given problem.
- *Human-based Genetic Algorithms:* Humans can contribute solutions to problems and all functions of a typical genetic algorithm(initialization, mutation and recombination) are performed by humans.
- *KCVC:* The aim is to improve machine learning algorithms by using volunteers' knowledge to build large databases of common sense facts.

[5] http://www.gwap.com/gwap/

2.4.2. Main Incentives to Participation

Computer programs do not need incentives to compute a given problem. Unlike computer programs, humans have to be motivated in order to accomplish a task which is part of a large-scale problem. Some incentives are as the following [19, 20]:

- *Volunteerism:* People want to support a cause/project, and they work on behalf of others. As an example, Wikipedia [6] is an open source software that everyone can contribute, i.e. Wikipedia get strong through collective intelligence.
- *Monetary Compensation:* Unlike volunteers, some people work in computational process for money. As an example, Amazon Mechanical Turk[7] is an online marketplace for work. Workers can choose a suitable task to complete for a monetary payment.
- *Fun:* While having fun, people produce useful data as a side effect. As an example, ESP game[8] is a GWAP which aims labeling images. Two random people are trying to label a same image with some tags and get points for matched tags.
- *Others:* Sometimes there is a need to identify a user from an automated agent to prevent spamming activites on websites. As an example, CAPTCHA asks a question which is easy for a human but difficult to solve for a computer. CAPTCHA method is widely used to gather useful data from users, a comics can be seen in Figure 2.3.

  reCAPTCHA[9] is another system which is trying to digitize books where a human is supposed to recognize words from old scanned books. These words are difficult to be identified by OCR [23] technique, so HCR is a complementary method to OCR. Figure 2.2 displays a reCAPTCHA example. In this example, the word "morning" is not recognized by OCR. reCAPTCHA picked the word, transformed it to another object adding a line through it, and then presented it as a challenge to a human. Another word for which the answer was known ("overlooks") was also shown to decide whether the user entered the correct answer.

[6] http://en.wikipedia.org
[7] https://www.mturk.com/mturk/welcome
[8] http://www.gwap.com/gwap/gamesPreview/espgame/
[9] http://www.google.com/recaptcha/learnmore

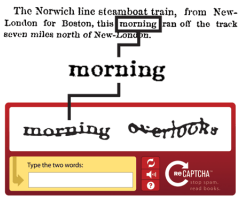


Figure 2.2. reCaptcha Example.

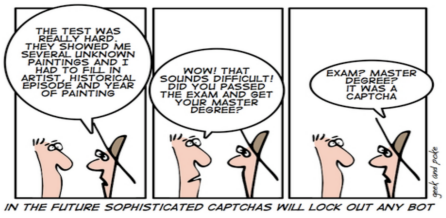


Figure 2.3. Future Sophisticated Captchas (See http://geekandpoke.typepad.com/).

2.5. A Brief Overview of Workflows

A workflow comprises a sequence of connected tasks which are carried out by humans/machines, according to a workflow definition. Workflow engines enable the automation of the workflow, i.e. it interprets the workflow definition, distribute tasks to workflow participants in order to achieve business goals [24, 25].

3. RELATED WORK

3.1. Sociological Aspect

Some research is done about communities in sociology field. In a work of Coleman [26], human capital and social capital are emphasized. Human capital is created by changes in persons as they get new skills and capabilities to act in new ways. In another words, human capital includes the skills and knowledge gathered in formal and informal learning. However, social capital is based on changes in the relations among persons and facilitates the learning and use of these skills and knowledge [27]. Human capital and social capital faciliate productive activity. In [28], social capital is defined as an accumulation of the knowledge in purposeful communities. And the paper concludes that social capital can only 'exist' if it is somehow able to be produced. Hence, it gives us the idea that people have to be somehow able to handle collective work in order to produce human capital and social capital.

3.2. Pantoto Project

The Pantoto project is an open source content management system following a community based approach, and enabling an easy to use technology for communities to build their identity on web and manage community information. Pantoto stores information in a structured manner, hence it offers efficient searching capabilities. Over the last several years, Pantoto has been deployed in various small-scale organisations like CHIREC[10] , SAATHI[11] and artisan groups, mostly in Southern India [29–31]. Applications developed by these organisations clearly show the appeal of such platforms to create solutions for coordinating, communicating and managing collective efforts.

In Pantoto, community members are encouraged to structure their own data. Thus, Pantoto allows communities to search better an information in order to process

[10] A K12 school in Hyderabad, India
[11] A community concerning the HIV infection in India

it, and Pantoto also enables getting long term value from generated information by the community itself. Pantoto is used by purposeful communities and enables a collaborative environment to work with people. Web pages, *Pagelets*, are specified by community members, authorization is defined through the use of *Views*, i.e. it helps describing roles, *Personas*, that can see and modify *Pagelets*.

Some important features of Pantoto are as the following:

- Community knowledge is managed without software developer support.
- Pantoto enables content and document management via forms which is simple to create and maintain.
- As the information is structured, it is possible to query data collected from users and have better results than querying unstructured data.
- Pantoto provides a role based access control for sharing information via forms. It is important to define how, when and by whom information can be created or altered.

Pantoto is similar to WeFlow in many aspects: (i) the audience is purposeful communities, (ii) web applications are specified by community members, (iii) units of work are described, *Pagelets* in Pantoto and *Human Tasks* in WeFlow, (iv) units of work are done by specified roles, *Personas* in Pantoto and *Roles* in WeFlow, (v) human computation is used to achieve a community goal. Pantoto is different from WeFlow because (i) there is no explicit way to describe workflows, execute them and distribute units of work to community members, (ii) all computation is done by humans thus computational powers of computers cannot be used, (iii) it doesn't allow to interact with external systems. The most powerful part of Pantoto is its data model. It enables to create user defined data which is very important to specify in purposeful communities. Such a robust data model is missing in WeFlow and its integration is part of future work.

3.3. Facebook API

Facebook has introduced an Application Programming Interface (API) for developing Facebook applications, whereby users are able to develop and deploy applications[12] . Such applications are disseminated in the usual Facebook viral manner. The support for user defined applications has been very successful, every day 20 million applications are installed on Facebook by developers [15].

Although Facebook API is designed to be used by its users, it is mostly used by developers as technical programming skills are needed. Facebook applications are web applications created by IT savvy people and these applications are used by Facebook users. There is no way to specify a workflow explicitly, but Facebook API compares to WeFlow in terms of generated output. Both aims to generate a web application to be used by an unknown group of people on Web in a collaborative environment.

An example application can be seen in Figure 3.1. This game, PhotoGrab, is built by using Facebook API. It is a social game where users are asked to upload some pictures from their Facebook albums to create their own PhotoGrab application[13] . Then, friends of this user are asked to recognize given patterns in pictures defined by the application owner.

3.4. Workflow Management Systems

Sometimes a collective work is broken down into small pieces and distributed to various people within a community, which may be managed through the use of workflows. In order to define workflows and use them as applications, Workflow Management System (WMS) is needed. A WMS defines, creates and manages the execution of workflows using a software as defined by Workflow Management Coalition (WfMC)[14] . WMSs are widely used by enterprises to achieve business goals. These systems provide workflow descriptions specified according to some business needs which are mostly de-

[12]http://developers.facebook.com
[13]http://apps.facebook.com/photograb
[14]http://www.wfmc.org/

Figure 3.1. A Facebook Application: PhotoGrab.

scribed with Business Process Execution Language (BPEL) [32]. IBM Business Process Management (BPM) Suite and Oracle BPM Suite are the leading industrial solutions available in the market [33, 34].

These softwares are private, very costly, difficult to customize and thus not appropriate to be used by communities. YAWL, jBPM, Ruote, Enhydra Shark are open source products available on Web. YAWL [35, 36] is the most widely used BPM environment and enables users to specify complex workflows. In a research conducted by Wil.M.P [37], patterns-based evaluation of jBPM [38], OpenWFE (now Ruote) [39], and Enhydra Shark [40] is provided. This report shows that the range of constructs supported by the three systems is somewhat limited in terms of control-flow, data and resourcing perspective. In addition, OpenWFE do not offer documentation for its graphical notation, jBPM requires Java programming, and Enhydra Shark offers desirable functionality such administration and user functionality in a closed-sourced version. And authors conclude that these open source systems are more towards developers than towards business/software analysts. Thus, these open source workflow

systems are mostly complex, difficult to customize, and offer centralized workflow modeling capabilities for IT savvy people.

WMSs are similar to WeFlow in many aspects: (i) a work is described in terms of a workflow, (ii) web tasks are generated according to human task specification, i.e. a task has typed input and output variables and it is mapped to Web to be executed by humans, (iii) tasks may be performed by automated agents or humans, (iv) human tasks are executed based on roles or individuals: in WMSs it is possible to assign a task to a specific person whereas in WeFlow, it is possible to refer to a person executing some specific task and this is because in WeFlow, tasks are distributed among a group of unknown people. WMSs are different from WeFlow because (i) they are not appropriate for purposeful communities as they are complex enough, (ii) workflows are specified by business analysts, (iii) security is a main issue in WMSs as opposed to WeFlow where non secure critically purposes are specified, (iv) in companies, employees are known as well as their roles. Thus, tasks are distributed among a known group of people.

3.4.1. Workflow Languages

There are various languages designed based on different formalisms which inspired us to build *WeFlow Specification Language*. YAWL extends Petri nets capabilities in order to describe workflows and is used by YAWL workflow system [36, 41]. This language is complete in terms of workflow patterns defined by Workflow Patterns Initiative [42] but is complex enough to be used by an average user. Occam [43, 44] is a simple parallel programming language to express concurrency. In Occam, concurrency is expressed explicitly at the statement level which is a natural and easy way to express [45]. Occam language constructs are similar to *WeFlow Specification Language* components. SMAWL is a small workflow language based on Calculus on Communication Systems (CCS) which remains as a conceptual language [46]. In this language, data and human task definitions are out of the scope and thus is not suitable to express community needs. There are also some efforts made to introduce a workflow model based on π-Calculus into Pantoto project [47].

Most of these languages are covering many workflow patterns. WeFlow aims to generate human computation web applications based on workflow idea, thus it doesn't aim to be a complete workflow language but a simple language to be used by community members. So, basic control-flow(sequence, parallel split, synchronization, exclusive choice, simple merge), data(task data, multiple instance data) and resource(role-based distribution) patterns are supported [42].

3.5. Yahoo! Pipes

Yahoo Pipes[15] enables users to create mashups using content on the Web. A mashup is an application that uses and combines data from other sources to create new services. Yahoo Pipes offers a visual editor letting end users to create mashups without having to write code. Pipes are useful for aggregating and manipulating existing data such as feeds. In addition, pipes can be published for the use of others and can also be created for personal use, *Private Pipes*.

Creating mashups is done by piecing together pre-configured modules. Modules are grouped into categories based on their functionality. *Sources* category consists of modules fetching data from other resources by parsing CSV, XML/JSON, RSS feeds. *User Inputs* category consists of modules helping in defining parameters for pipes, and these parameters are set by the user running the pipe before execution of a pipe. Input modules are used to get structured data such date, location, number, text, URL from the user running the pipe. *Operators* category consists of modules helping in transforming and filtering data coming from other resources. *Operators* modules only work with a list of items in CSV, XML/JSON or RSS feed format.

Yahoo Pipes, like WeFlow, are meant to be accessible to non savvy IT users. The WeFlow editor has not yet been implemented. It is, of course, a critical part of the framework, which is currently being designed. The most significant difference is that WeFlow aims to support collaborative applications for purposeful communities. It would be possible to utilize Yahoo Pipes in terms of creating useful web services for

[15] http://pipes.yahoo.com/pipes/

WeFlow applications.

Yahoo Pipes is also similar to WeFlow in many aspects: (i) units of work are defined, *module* in Yahoo Pipes and *tasks* in WeFlow, (ii) work is defined based on workflow, i.e, control flow and data flow are defined between units of work, (iii) data related to units of work is defined as typed variables, (iv) control flow is defined by wiring modules together, and data flow is defined between any modules according to appropriate data types, i.e, a text field may be mapped to another text field. Yahoo Pipes is different from WeFlow because: (i) pipes are for personal use only, once published they become accessible on Web. Pipes may be used by other users, in this case the pipe is duplicated in order to be used by other users, (ii) there is no interaction between Yahoo Pipes users, i.e. it is not a collaborative environment, (iii) only predefined modules can be used on the other hand, WeFlow supports user defined tasks, (iv) once a pipe is running, it is not interrupted by a user. A user can only provide parameters before running a pipe. In WeFlow, humans are interrupting the workflow at execution time, (v) modules are executed by automated agents but in WeFlow, tasks are mostly performed by humans, (vi) output of a pipe is always a list of items but in WeFlow output of a workflow is defined by the specifier of the workflow, (vii) pipes are not running applications. Pipes are executed on demand thus there is no data collection related to running pipe.

In Figure 3.2, a Yahoo Pipe application, *Top 10 Movies Trailer*, is depicted. This pipe gets top ten movies from Rotten Tomatoes[16] and then use Youtube[17] to get the movie trailer for each movie. *Fetch Feed* construct extract information from a URL, *Loop* construct gets title of a movie and then extract trailer information from Youtube, and Pipe Output construct shows the result as a list of movies, see Figure 3.3. In a recent work [48], authors try to observe end-user programming behaviors in Yahoo Pipes and they conclude that users employ only a small number of constructs that Pipes offers to them. In another words, users compose simple pipes consisting of only three or four constructs. This work shows the necessity to develop an environment

[16]Rotten Tomatoes is a website devoted to reviews, information, and news of films
[17]http://www.youtube.com/

to make creation of workflows simple and let users use it effectively to address their community needs.

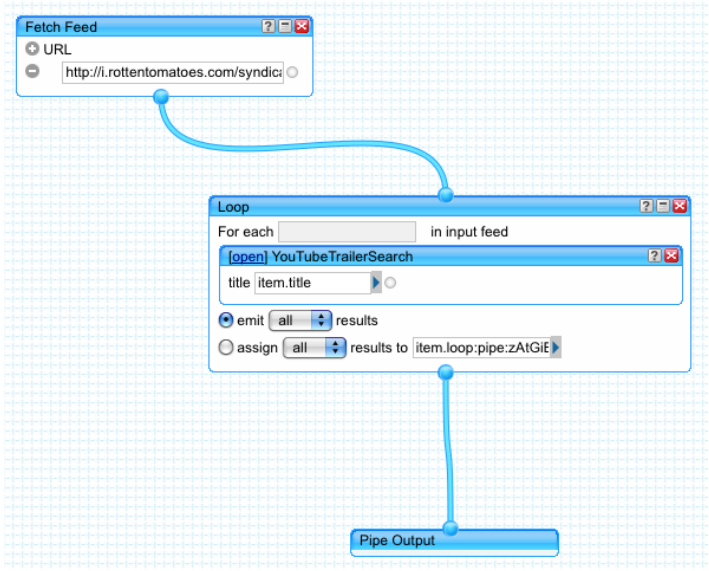


Figure 3.2. Top 10 Movies Trailer.

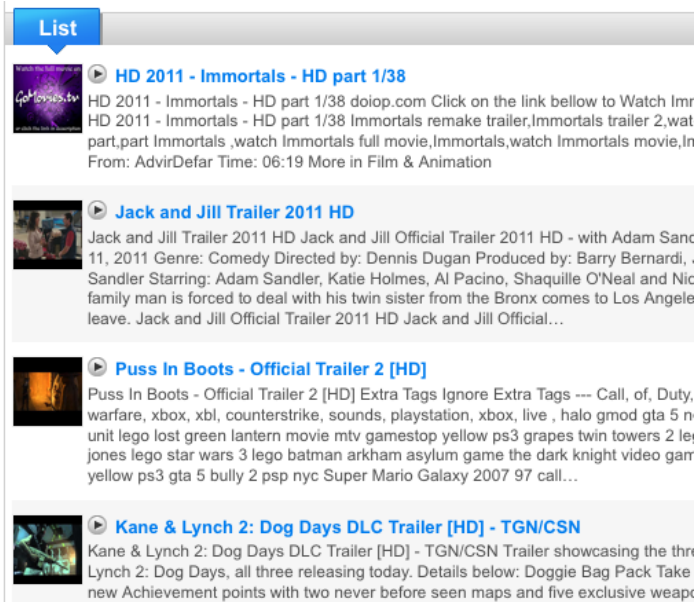


Figure 3.3. Top 10 Movies Trailer Output.

4. PROPOSED MODEL

In purposeful communities, community members would like to create places where they can come together according to their community needs. Web is a great platform to collaborate with people from all around the world, thus community members need web applications to use. Developing web applications is a trivial task for community members who mostly lack technical skills, and such web applications are developed by IT savvy people. And this is even true for simple web applications that any person could think of. WeFlow aims to empower community members to develop their own web applications. As humans are involved mostly in these applications and core computation is done by them, main focus is on specifying human computation web applications.

Communities have a purpose to achieve and this purpose is the actual work to be done by community members. This work is broken down into small units of work, *tasks*, to be performed by people. Relationships among these tasks have to be defined in order to accomplish this work, the community purpose. In WeFlow, workflow model is considered in order to specify such a work.

In this study, mainly focus is on the area of Human Computation, i.e. users are solving a problem by collaborating with computers. A workflow model of specifying human computation applications for the Web, WeFlow, is proposed and virtual community members can use it to specify Web application behavior desired in their community.

Through the use of workflow specifications, community members define ad hoc workflows in a web-based environment to create, manage and accomplish tasks according to their community needs. Given such specifications, Web applications are generated preserving the semantics of these specifications. Our model helps online communities to use the Web for collective activity and it empowers Web users to carry out envisioned applications that would serve their needs. Proposed approach is as the

following: (i) define a workflow specification, (ii) generate a Web application given such specification and (iii) monitor and manage this application by preserving specification semantics.

WeFlow Framework is depicted in Figure 4.1. Workflow specification, *WeFlow Specification*, consists of the description of collective work to be done, and it is provided by the *Specifier*. The *Specifier* is an ordinary person who wants to define a human computation web application. Starting from a *WeFlow Specification*, a human computation web application is generated by *WeFlow Application Generator*. The workflow engine, *WeFlow Execution Engine*, executes this web application and distributes tasks to workflow participants preserving the semantics of given specification. Note that *WeFlow Execution Engine* runs many human computation web applications. For the purposes of clarity, workflow participant interactions are only depicted for one executing web application.

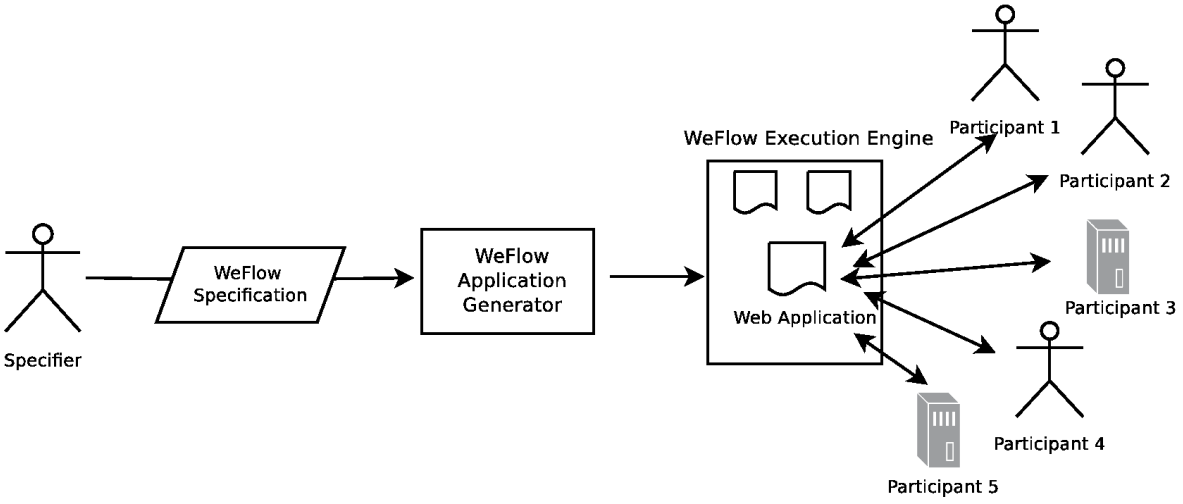


Figure 4.1. WeFlow Framework.

4.1. WeFlow Components

4.2. Terminology

In this section, we briefly describe some basic terms.

- *Task:* It describes a unit of work that may need to be performed as part of a workflow.
- *Workflow:* It is composed of tasks that need to be performed to achieve a workflow goal.
- *Workflow Specification:* It describes workflow behavior by defining which tasks should be performed by whom using which data, under which conditions and in which order.
- *Workflow Instance:* A specific instantiation of a workflow that needs to be performed.
- *Task Instance:* A specific instantiation of a task that needs to be performed as part of a given workflow instance.

In the next section, we detail these fundamental terms.

4.3. Workflow

A workflow represents a collective work which is broken down into tasks. Tasks are connected to each other by *control flows*. The transfer of work between two tasks is done through a control flow. In a workflow, set of control flows specifies how tasks are to be ordered. The data transfer between tasks is done through *data flows*. In a workflow, set of data flows specifies how data is moved from one task to another.

In Figure 4.2, an example workflow is shown. Note that tasks are depicted as rectangles, control flows as unidirectional arrows and data flows as unidirectional dashed arrows. This example shows that once execution of *task 1* is completed, control is passed from *task 1* to *task 2*, and from *task 2* to *task 3*. In Section 4.7, flow aspects will be detailed.

There is a unique first task, *Start Task*, and may be multiple end tasks in a workflow, *End Tasks*. Start Task is the first task to run, and End Task(s) is/are last task(s) to run in a given workflow instance. In Figure 4.2, *task 1* is the *Start Task* and there is only one *End Task* depicted, *task 3*. In another words, after the execution of

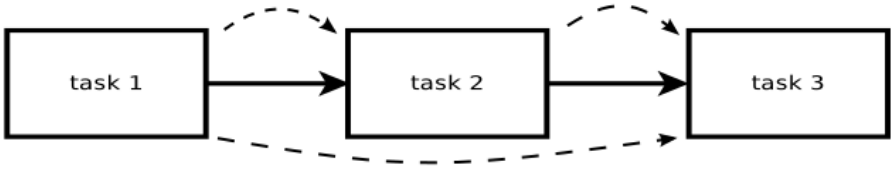


Figure 4.2. An Example Workflow.

task 3, this workflow terminates.

4.4. Task

A task specifies the smallest unit of work in a workflow. There are two types of tasks in terms of who performs the task: (i) human, (ii) automated. Each task which requires human intervention (getting data from, displaying information to) is a *Human Task*, *Automated Task* otherwise.

Hence, a task may be carried out:

- manually by a human, *Human Task*:
  - (i) If a human task has inputs but no outputs, it means that input data will be showed to the human, Human is a *consumer*
  - (ii) If a human task has both inputs and outputs, it means that human will provide inputs according to the human task description, Human is a *prosumer*
- automatically by a software application, *Automated Task*
  - (i) The body of code to be executed can be coded by a savvy computer user.
  - (ii) Predefined automated tasks from the library can be used, i.e. the body of code is ready to execute.
  - (iii) External services such as Web services can be used.

Each task specification consists of the specification of the following:

- *Inputs:* A task may have several inputs which are represented as WeFlow typed

Table 4.1. WeFlow Data Types.

WeFlow Data Type	Used for
Text	textual data
URL	url data
Date	date data
Email	email data
Integer	integer number data
File	file data
Password	password data
Hidden	hidden data
Image	image data
Paragraph	few lines of text data
Choice	selecting only one of a limited number of choices
Multiple Choice	selecting one or more options of a limited number of choices
Checkbox	selecting one or more options of a limited number of choices
Radio	selecting only one of a limited number of choices

variables, see Table 4.1.

There are two types of inputs in terms of from where values are received:

(i) *task input variables* receive values from previously performed tasks.

(ii) *human input variables* receive values from a human performing the task.

- *Behavior:* For *Human Tasks*, the behavior is specified in terms of a description that includes:

  (i) textual directions that inform the user what they are expected to perform, e.g., "Tag the given picture"

  (ii) zero or more task input variables, which are values provided by previously performed tasks, e.g., "Upload *$number $animal* pictures". *number* and *animal* are task input variables.

  (iii) one or more human input variables, which are values to be provided by the person performing the task, e.g., "Provide a tag *$tag* for this picture *$picture*". *tag* is a human input variable hence expecting some value from a human, and *picture* is a task input variable having its value coming from a previously performed task.

  Note that the sign *$* is a special character pointing to the value of a variable and further details about how to display a human task's description will be discussed in Section 4.8.

  For *Automated Tasks*, the behavior is specified in terms of a description that includes textual directions describing the task to be performed.

- *Outputs:* A task may have zero or more output variables which are also represented as typed variables, see Table 4.1. The value of an output variable is formulated in terms of the task input and human input variables. This formulation will be detailed in Section 4.7.2.

In Figure 4.3, an example task is depicted. This task has m task input variables, n task output variables and p human input variables. Task input variables are getting their values from previously performed tasks, human input variables are set by the

human performing this task, and output variables are sending their values to following tasks.

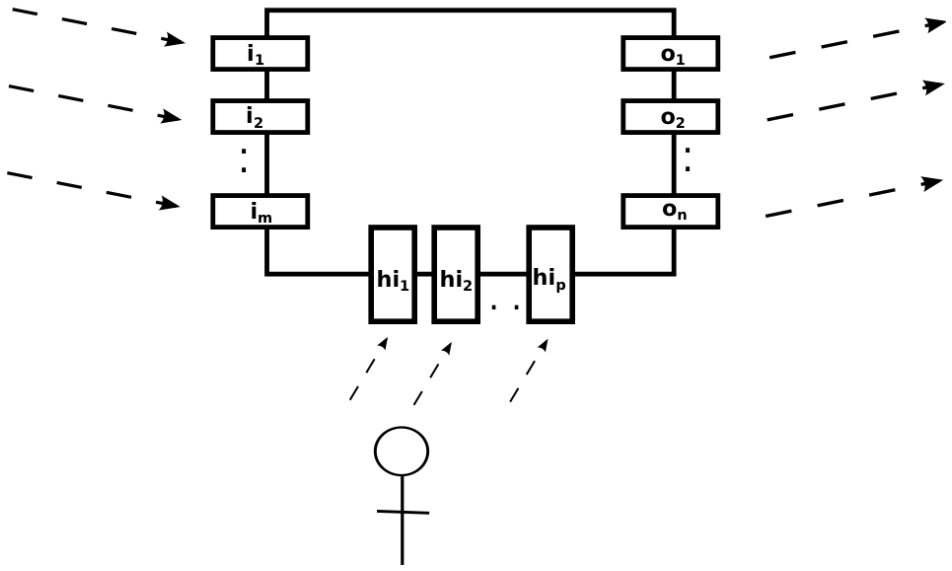


Figure 4.3. A Generic Task.

In Table 4.2, task types and corresponding shapes are represented. These shapes will be used in further figures.

There are various preconfigured task types which are defined in Section 4.5.

4.5. WeFlow Task Types

4.5.1. Basic Task

It represents a single task to be performed and can not be divided into subtasks. A Basic Task is depicted as Figure 4.4. If the task performer is human, a human figure is added to the task figure.

Table 4.2. Task Types with Corresponding Shapes.

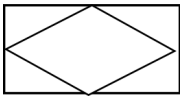
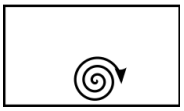
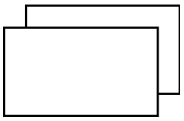
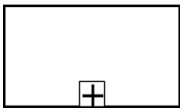
Task Types
Basic
Conditional 
Repetition 
Collective 
Composite 

Figure 4.4. Basic Task.

4.5.2. Conditional Task

There are various types of *Conditional Task*. These types require the use of a *Conditional Expression* to satisfy and choose an appropriate task to execute.

A *Conditional Expression* is an expression in the form of a triple: <literal, operator, literal>. Operator should be one of: <, >, =, <=, >=, !=. A literal may be a value received from previously performed tasks or a hand coded value by default. For example, "$a >5", "$a != $b", "1 == 2" are all valid conditional expressions.

- *IfElse Task* provides a choice between two tasks (t_1, t_2) based on satisfiability of the *Conditional Expression*. If it is satisfied t_1 is executed, t_2 otherwise. This behavior is depicted in Figure 4.5. Depending on the *Conditional Expression*, one of subtasks (t_1, t_2) will be executed.

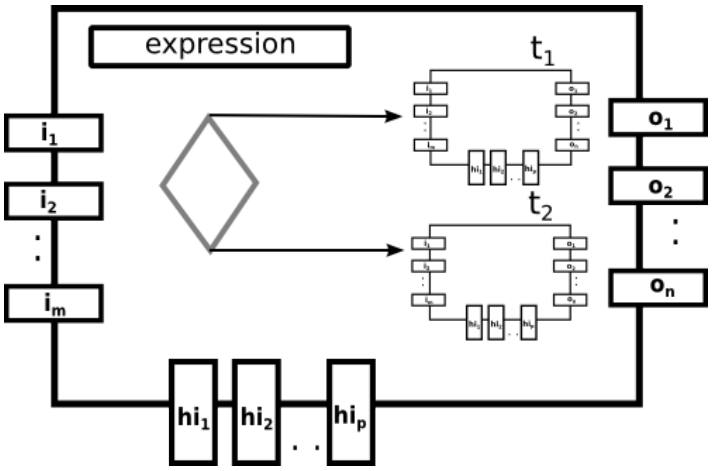


Figure 4.5. IfElse Task.

- *Choice Task* provides a choice between multiple tasks (t_1, t_2, .. ,t_n) based on value that gets the *Conditional Expression* after its evaluation. According to this value, one of these tasks is executed. Note that this is a deterministic choice. This behavior is depicted in Figure 4.6. Depending on the *Conditional Expression*, one of subtasks (t_1, t_2, ..., t_n) will be executed.

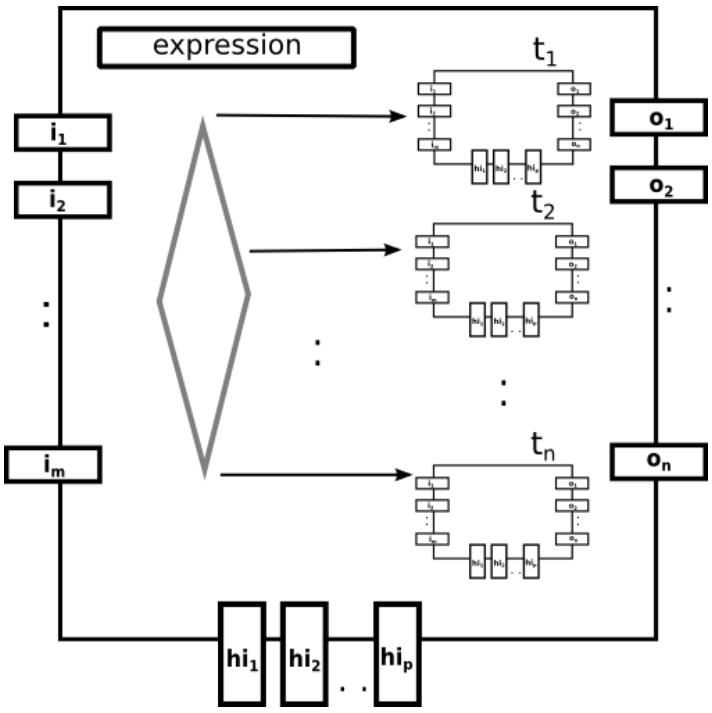


Figure 4.6. Case Task.

4.5.3. DoAll Task

It requires a set of tasks, and all tasks have to be executed in parallel which is the implicit condition. In Figure 4.7, a generic *DoAll Task* is depicted. In this figure, *DoAll Task* is a supertask and has n subtasks denoted as t_1, t_2, ..., t_n. Note that this supertask will be completed once all its subtasks are executed.

4.5.4. Repetition Task

Repetition Task has a *Conditional Expression* to satisfy, and a task to repeat while this condition is satisfied.

At each repetition step, a new instance of the task to be repeated is created i.e. Repetition Task allows to run multiple instances of a task sequentially until its *Conditional Expression* is satisfied.

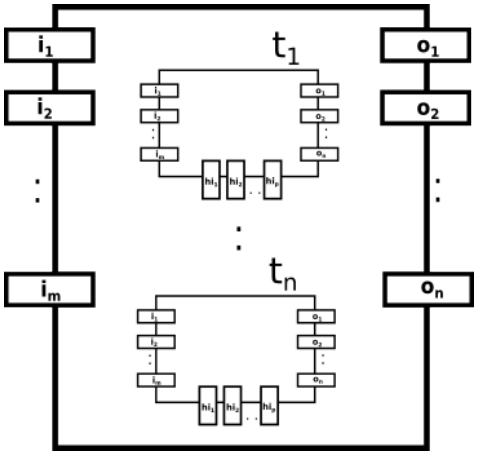


Figure 4.7. DoAll Task.

Conditional Expression has at least one variable, and may have multiple variables. These variables are initialized by Repetition Task's inputs(task inputs and/or task human inputs). After each repetition step, these variables are updated by Repetition Task's inputs which in its turn are updated by outputs of the task to be repeated. Thus, data mappings between the task to be repeated and Repetiton Task's inputs should be defined. Note that there should be at least one input mapping(mapped variable should also be a conditional expression variable), from the repeated task to the Repetition Task, i.e. from subtask to supertask, in order to prevent infinite loop cases.

In Figure 4.8, a generic Repetition Task is depicted. t_1 is the Repetition Task, and t_2 is the task to be repeated while the condition is fulfilled.

The semantics of t_1 is detailed in Figure 4.9. Data flow aspects are also shown in the algorithm. All input mappings for t_2 are coming from its supertask, t_1. All variables for the conditional expression are also t_1's variables, i.e. t_1's inputs $\supseteq$ *exp*'s variables. Note that data mappings are not shown on the figure for the purposes of clarity. In Section 4.7.2, data flow aspect will be detailed.

For a Repetition Task, control flow semantics for Figure 4.8 are depicted in Fig-

ure 4.10. Note that control passes from t_0 to t_1, which is the Repetition Task. While its condition is fulfilled, control is passed to t_2, once t_2's execution is done, t_1 takes the control and so on. If the condition is not fulfilled, t_1's execution is done and next task, t_3, takes the control of the workflow. In Section 4.7.1, control flow aspect will be detailed.

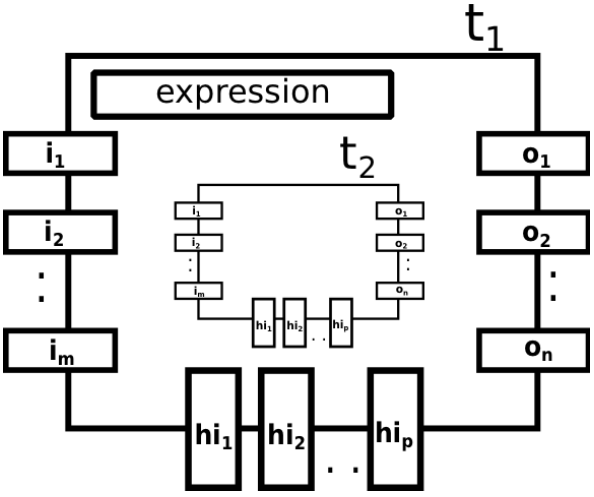


Figure 4.8. A Generic Repetition Task.

4.5.5. Collective Task

It is a supertask, containing a subtask, and allows to run multiple instances of the subtask concurrently. The output of a *Collective Task* is an array including output values coming from each subtask instance, i.e. it collects data from all executing subtask instances. After each execution of a subtask, this subtask's output values are appended to supertask's output array (See Section 4.7 for data flow definition). The number of instances may be specified as infinite or a finite number. In infinite case, a new instance of the subtask is created at every request to execute the subtask and the workflow never terminates unless the *Specifier* terminates it. A *Collective Task* is depicted as Figure 4.11. t_1 is a *Collective Task*, and t_2 is its subtask. t_2 is executed several times, and t_2's outputs are appended to t_1's output arrays.

Require: **exp** of type Conditional Expression, $\mathbf{t}_2$ of type Task

call *DoInputMappings* **with** t_1

for each variable v **in** exp's variables **do**

v $\Leftarrow$ v of t_1

end for

while exp **do**

call *DoInputMappings* **with** t_2

call t_2

call *DoOutputMappings* **with** t_2

call *DoInputMappings* **with** t_1

for each variable v **in** exp's variables **do**

v $\Leftarrow$ v of t_1

end for

end while

call *DoOutputMappings* **with** t_1

Figure 4.9. Semantics of Repetition Task t_1.

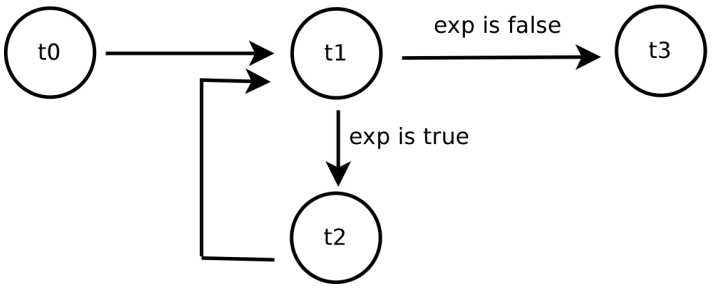


Figure 4.10. Control Flow in a Repetition Task.

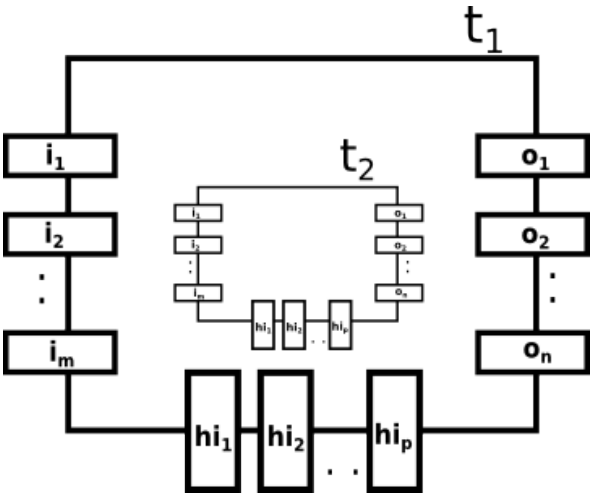


Figure 4.11. A Generic Collective Task.

4.5.6. Composite Task

It's a set of connected tasks, thus a container of a subflow. When a *Composite Task* is started, it passes control to the first task of the subflow, and after execution of this subflow, *Composite Task* takes the control back. After that, the execution of the main flow continues as specified. A *Composite Task* is depicted as Figure 4.12. *Composite Task* includes a subflow starting with t_1 and terminated by t_n. Once the execution of this subflow is done, control flow is owned by *Composite Task*, and data mappings from subflow to *Composite Task* are done.

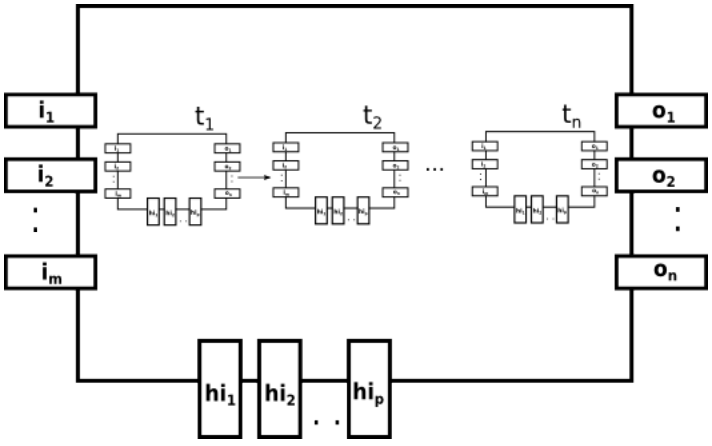


Figure 4.12. Composite Task.

4.5.7. Data Task

Data task is used to define relations between a task and data model. It helps to store some data, or retrieve some data given a query. Data task allows its users to explicitly specify which data to store or retrieve and enables data persistency within *WeFlow Framework*.

4.6. WeFlow People

In Section 4.4, task properties have been described. A workflow may consist of a number of human/automated tasks. For human tasks, the *Specifier* has to describe which participants should be doing the work, i.e. people should have sufficient permission to perform human tasks.

4.6.1. Role Based

In *WeFlow Framework*, workflow participants have roles and each human task is associated with one or more roles. Thus, each human task may be performed by one of specified roles.

For example, assume that Jack have roles such *manager* and *member* in a WeFlow web application. In this case, Jack may perform all tasks which are specified to be executed by *manager* and *member* roles.

4.6.2. Individual Based

It is also possible to refer to a task performer in order to let another task to be executed by the same performer. In another words, it is possible to say something like: "*Task B* has to be executed by the same person who executed *Task A*". Note that as WeFlow aims to generate human computation web applications which will be executed by unknown users, tasks cannot be assigned to specific users.

4.7. WeFlow Flow Aspects

So far, it was explained how to define tasks and people to perform these tasks. In this section, flow perspective of workflows will be discussed. There are two flow aspects: (i) control flow, (ii) data flow.

As mentioned in section 4.2, tasks are attached one to another via control flows in a workflow. In a workflow instance, task instances have to be executed in a specific order and this order defines *control flow* aspect of a workflow. In a workflow, data is mobile and moving from one task to another. A task uses some input data and may create some output data. Hence, data may be incoming to a task or outgoing from a task. And this data moving from one task to another defines *data flow* aspect of a workflow.

WeFlow takes care of control flow and data flow aspects of a workflow. In following sections, these flow aspects will be detailed.

4.7.1. Control Flow

In a workflow, tasks have to be executed in a specific order and this sequencing in which work needs to be done is the *control flow* aspect of a workflow.

In Figure 4.13, there are three tasks: A, B and C. Two control flows are shown on the figure, one from task A to task B, and another one from task B to task C. Once task A completes, control is passed to task B. Note that control cannot pass from task A to task C because there is no control flow between these tasks. In this example, control flow is a simple sequence: first A, then B and then C.

Control flow semantics for different types of tasks will be explained in Section 4.5

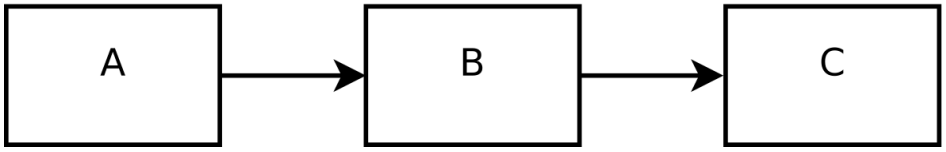


Figure 4.13. Control Flow Example 1.

4.7.2. Data Flow

In a workflow specification, data flow is a declarative definition of data movement within and between tasks. In another words, data is mapped to task variables via data mappings. According to data mapping information, a task is instantiated and executed.

At execution time, a task has to be initialized in terms of its inputs, i.e. data required to do the work has to be mapped to task input variables, and a task has to be finalized in terms of its outputs, i.e. produced data has to be mapped to task output variables. This data perspective defines *data flow* aspect of a workflow. In addition, task initialization and finalization operations are done by the *WeFlow Execution Engine* which will be detailed in Section 4.9.

As detailed in Section 4.2, a task has input and output variables. Task input variable values are mapped from previously performed tasks, human task input variable values are set by human performers and task output variables are formulated in terms of task inputs. There are three types of data mappings: (i) task input mappings, (ii) human input mappings and (iii) output mappings.

Note that $ sign is pointing to the value of a variable and will be used in data flow examples.

4.7.2.1. Input Mappings. These are data mappings between different tasks. Before execution of a task t, data required to do the work is mapped to t's task input variables. This data should come from previously performed tasks or if t is a subtask, it should

come from its supertask (*Repetition Task*, *Collective Task*, *DoAll Task*), human input variable values are provided by the task's performer. Note that task input variable values of a subtask are always provided by its supertask.

In Figure 4.14, input mapping behavior is depicted. Data is coming from outside of the task and mapped to task input variables. Note that for each task input variable, an input data mapping is needed. Hence, m mappings are shown in this figure.

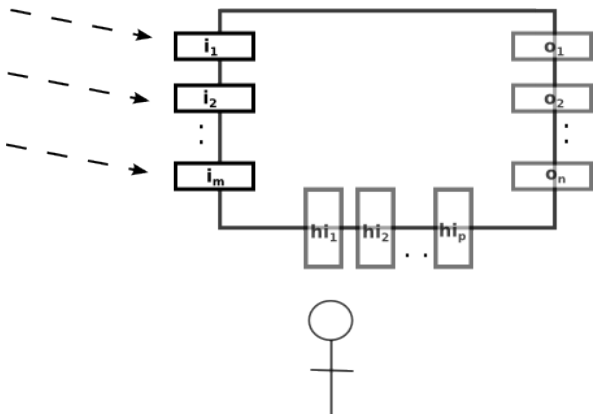


Figure 4.14. Task Input Mappings.

In Figure 4.15, two tasks are depicted: *Generate Numbers* and *Sum Integers*. *Generate Numbers* task has no input variables and two output variables (*no1*, *no2*) and is an *Automated Task* which computes two integers. *Sum Integers* task is a *Human Task*, has two task input variables (*no1*, *no2*), one human input (*ans*) and one output variable, *res*. Note that there is one control flow between these two tasks.

At execution time of *Sum Integers* task, *Sum Integers* task input variables are initialized by received values from *Generate Numbers* task which is previously executed. *2* and *3* are computed values by *Generate Numbers* task, see Figure 4.16. This mapping from one task to another one is called an *input mapping* and is depicted as dashed arrows in Figure 4.17. *Sum Integers* task input variables are then initialized by *2* and *3*, see Figure 4.18.

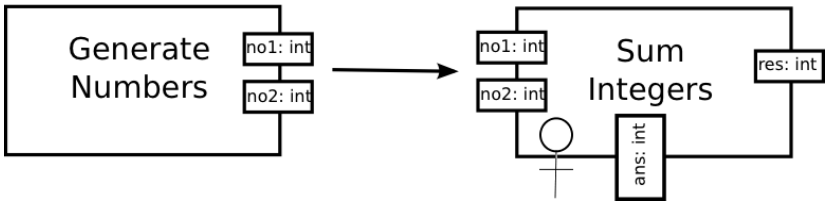


Figure 4.15. Specification of two tasks: *Generate Numbers* and *Sum Integers*.

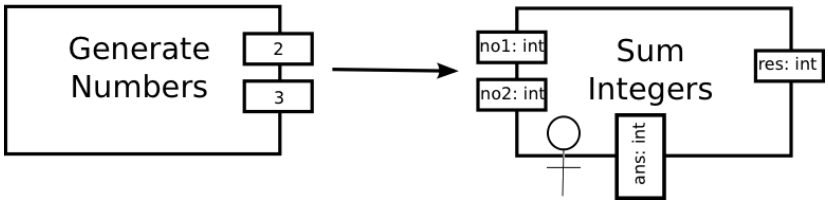


Figure 4.16. After execution of *Generate Numbers* Task.

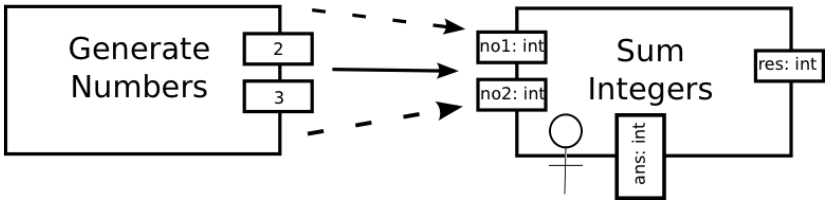


Figure 4.17. Data Mappings between two tasks.

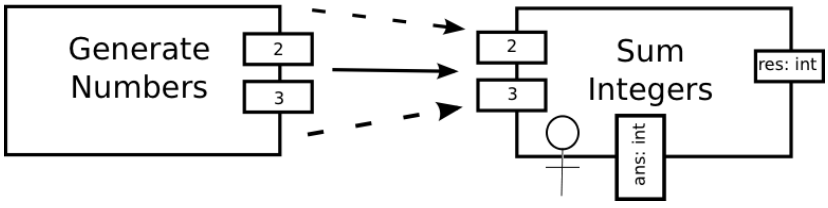


Figure 4.18. Mapping data to *Sum Integers* task.

In Figure 4.19, a supertask-subtask data relation is depicted. t_1 is a supertask and t_2 is its subtask, t_2's input mappings are shown with dashed arrows. t_1's input variable values are mapped to t_2's input variables. Note that, in this figure t_1 is a *Repetition Task*. For *Collective Task* and *DoAll Task* input mappings are done in a same way.

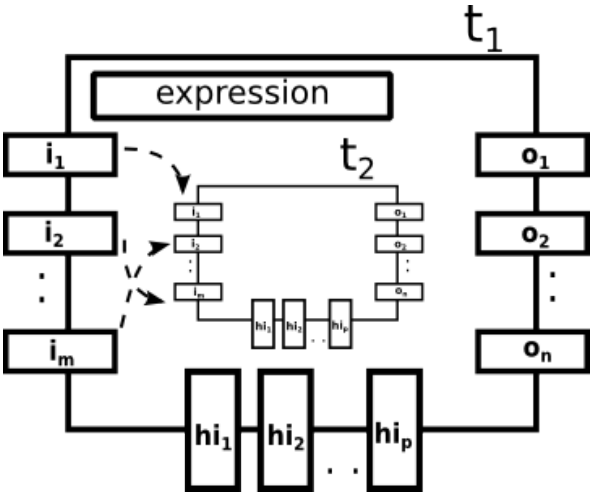


Figure 4.19. Supertask-Subtask Input Mappings: Repetition Task Case.

4.7.2.2. Human Input Mappings. These are data mappings between a human task and a human. At execution time of a task t, human data is mapped to t's human input variables. In Figure 4.20, this behavior is depicted. For each human input variable, a data mapping is defined. In the figure, p data mappings are shown and human provides data for these human input variables.

Communication between humans and human tasks are done through the use of *channels*. A channel provides an infrastructure to pass and collect data from humans. There are two channel operations:

- *put(ch, desc)* is used to put a task description *desc* in channel *ch*
- *get(ch, d)* is used to get data *d* from channel *ch*

A channel is created at execution time of a human task to communicate with the task

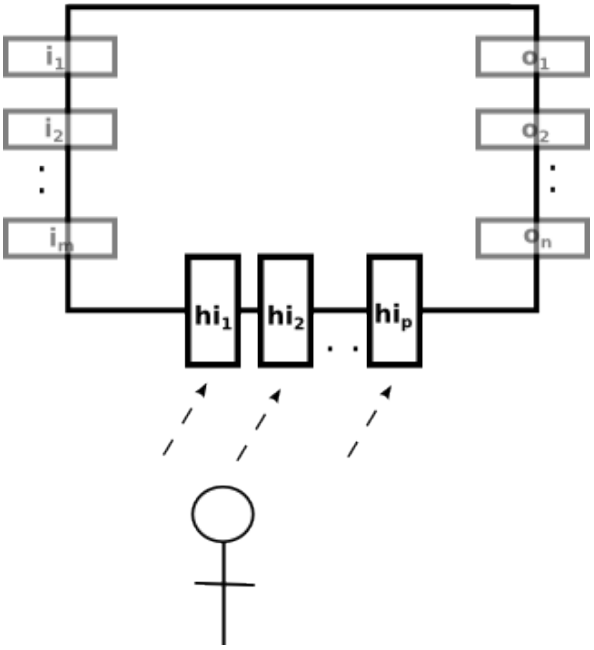


Figure 4.20. Human Input Mappings.

performer. First, *put* operation is invoked and task description is sent to the task performer via this channel. Then, another channel is created to invoke *get* operation in order to get data provided by the task performer.

For example, let's take a human task with a description *desc* like "Provide a tag *$tag* for this picture *$picture*". In this description, *tag* is a human input variable and *picture* is a task input variable which takes its value from a previously performed task. A channel $channel_1$ is created, and a put operation is invoked by *put(*$channel_1$*,desc)*, see Figure 4.21. *desc* is shown to the human who in his turn provides a value for *tag* human input variable. Then, another channel $channel_2$ is created in order to get *tag* value by invoking *get(*$channel_2$*, tag)* operation, see Figure 4.22. This is how human data is mapped to task's human input variable *tag*.

Let's consider *Sum Integers* task mentioned in previous section, see Figure 4.23. In previous section, remember that *no1* was computed as 2 and *no2* as 3. Now, task performer has to provide a value for human input variable *ans*. Through channel

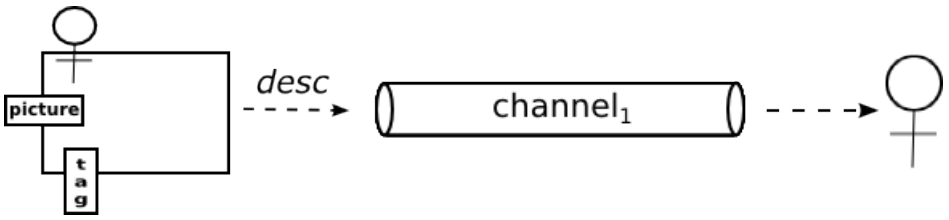


Figure 4.21. *put* Channel Operation.

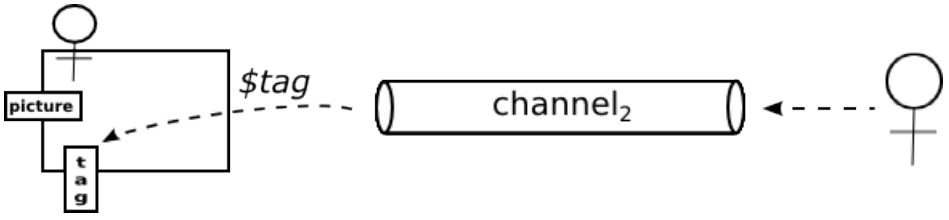


Figure 4.22. *get* Channel Operation.

communication, data is received from task performer and mapped to *ans* which is depicted in Figure 4.24.

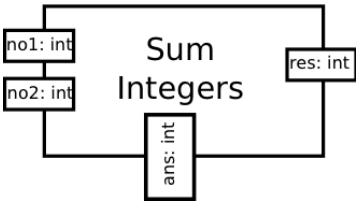


Figure 4.23. *Sum Integers* Task.

<u>4.7.2.3. Output Mappings.</u> These are data mappings occuring within a same task or between different tasks. The idea is mapping data to output variables of tasks.

These data mappings may be:

- *within same task* Input variable values may directly be mapped to output variable values, e.g., "$\$o_1 \leftarrow \$i_1$" where input i_1's value is assigned to o_1's value, or evaluated expressions which uses input variable values may be mapped to output

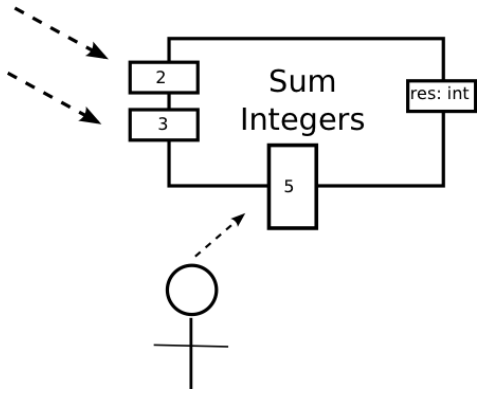


Figure 4.24. Task performer providing data.

variable values, e.g.,"$\$o_1 \leftarrow \$i_1 + \$i_2$" where input i_1 and input i_2 values are concatenated and the result value is assigned to output o_1's value.

In Figure 4.25, direct mappings are are as the following: $\$o_n \leftarrow \$i_2$, $\$o_1 \leftarrow \$hi_1$. Whereas in Figure 4.26, a more complex expression is assigned to output variable: $\$o_2 \leftarrow \$i_1 + \$hi_1$.

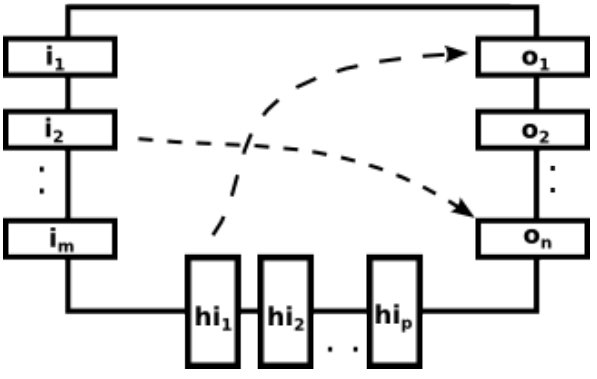


Figure 4.25. Direct output mapping.

- *between different tasks* If output data mappings occur from t_2 to t_1, then t_1 is a supertask (*Repetition Task*, *Collective Task*, *DoAll Task*) and t_2 is its subtask.

  If t_1 is a *Repetition Task*, then last executed t_2's output values are mapped to t_1's

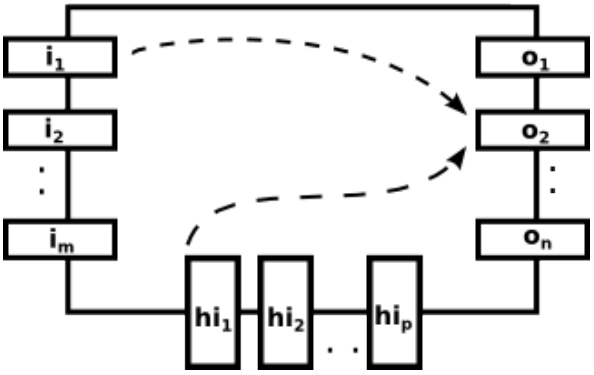


Figure 4.26. More complex expression mapping to output.

output values. In Figure 4.27, last executed t_2's output values are mapped to t_1's output values: $t_1.\$o_2 \leftarrow t_2.\$o_1$, $t_1.\$o_1 \leftarrow t_2.\$o_2$ and $t_1.\$o_n \leftarrow t_2.\$o_n$.

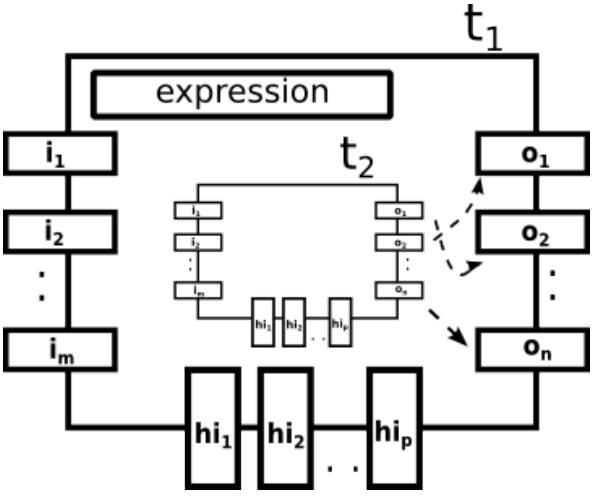


Figure 4.27. Outmappings in a Repetition Task.

If t_1 is a *Collective Task*, then for each executed t_2, t_2's output values are appended to t_2's output array. In Figure 4.28, t_2's output values are appended to t_1's output values: $t_1.\$o_2 \xleftarrow{append} t_2.\$o_1$, $t_1.\$o_1 \xleftarrow{append} t_2.\$o_2$ and $t_1.\$o_n \xleftarrow{append} t_2.\$o_n$.
If t_1 is a *DoAll Task*, then t_2's output values are mapped to t_1's output variables. In Figure 4.29, t_2's and t_3's output values are mapped to t_1's output values: $t_1.\$o_2 \leftarrow t_2.\$o_1$, $t_1.\$o_1 \leftarrow t_2.\$o_2$, $t_1.\$o_{n-2} \leftarrow t_2.\$o_n$, $t_1.\$o_{n-1} \leftarrow t_3.\$o_n$ and $t_1.\$o_n \leftarrow t_3.\$o_2$.

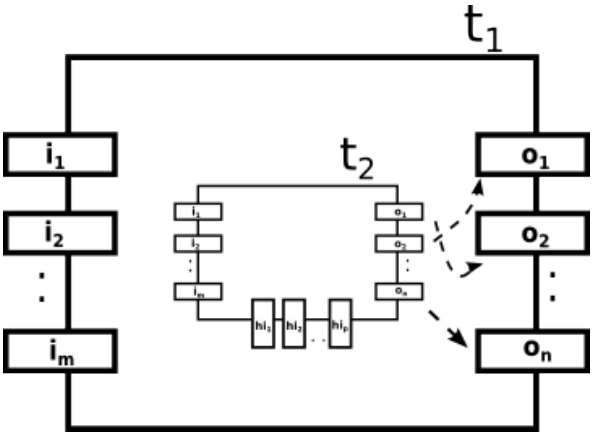


Figure 4.28. Outmappings in a Collective Task.

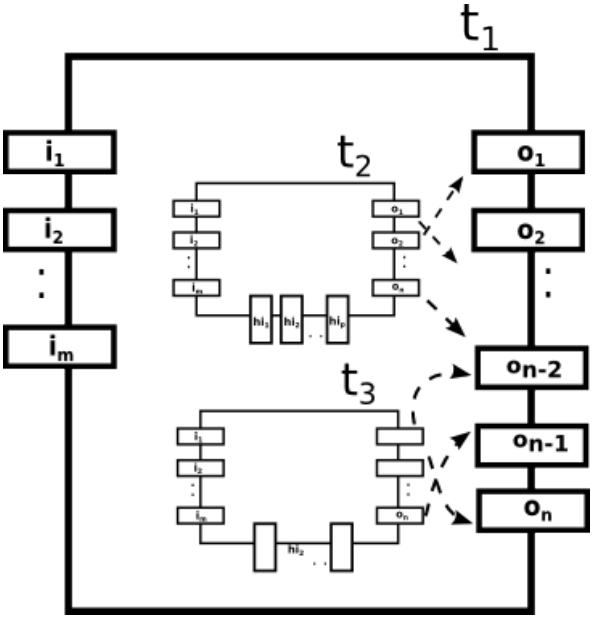


Figure 4.29. Outmappings in a DoAll Task.

Let's take again our *Sum Integers Task* example. In Figure 4.23, task input variables were initialized by *2* and *3*. While task instance was executing, human input variable value got the value *5* from the human, see Figure 4.30. And this human input value was mapped to human input variable. And now, this human input variable will be mapped to output variable, *res*, see Figure 4.30. After data mappings, value of output variable becomes *5*, see Figure 4.31. This example combines all three data mappings (Input Mappings, Human Input Mappings, Output Mappings) that were discussed in this section.

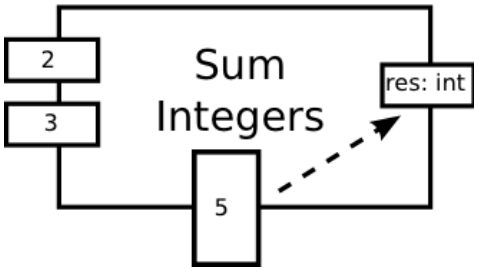


Figure 4.30. Sum Integers Task doing mappings.

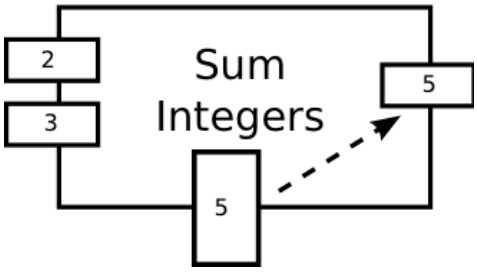


Figure 4.31. Sum Integers Task after Execution.

4.8. WeFlow Application Generator

This section focuses on how to generate a human computation web application given such *WeFlow Specification* which includes tasks, people and data flow information.

Table 4.3. Mapping Data Types to HTML Elements.

WeFlow Data Type	HTML Element
Text	<input type="text">
URL	<input type="url">
Date	<input type="date">
Email	<input type="email">
Integer	<input type="number">
File	<input type="file">
Password	<input type="password">
Hidden	<input type="hidden">
Image	<img>
Paragraph	<textarea>
Choice	<select>
Multiple Choice	<select multiple="multiple">
Checkbox	<input type="checkbox">
Radio	<input type="radio">

4.8.1. Mapping Data Types to HTML Elements

In Section 4.4, it was mentioned that a task may have zero or more task/human input variables and output variables which are both represented as typed variables. In order to represent a task on Web, these variables should be mapped to HyperText Markup Language (HTML) elements. In Table 4.3, corresponding HTML5 elements are listed. Note that new input types offered by HTML5 are also listed [49].

4.8.2. Representing a Human Task on Web

In order to perform a human task, a human performer should know information about the human task which is provided by the task's description. As previously discussed, a task's description may include textual information and/or references to task input/human variables. In a task's description, referenced variables are marked with a $ sign which is a special character pointing to the value of variables. Task

description is the interface between a human task and its task performer.

A task description is represented on Web as the following:

- textual information is displayed as text, e.g., "Tag the given picture"
- each referenced task input variable is replaced by its value.
  For example, if task description is "Upload *$number* *$animal* pictures". In this description, *number* and *animal* are task input variables. Thus, *$number* will be replaced by its value as received from another task, and same for *$animal*. Finally, this description will look like as "Upload 4 bird pictures" if *$number* is equal to 4, and *$animal* is equal to "bird".

- each referenced human input variable is replaced by a corresponding Web input method as described in the previous section.
  For example, if task description is "Provide a tag *$tag* for this picture *$picture*". In this description, *tag* is a human input variable, and *picture* is a task input variable. Thus, value of *picture* variable is shown to the user who provides a value for *tag* variable. If *$picture* is a bird picture coming from a previously performed task, this bird picture is shown to the user who should provide a tag for it. In Figure 4.32, this behavior is depicted. Note that *picture* variable is of WeFlow data type "image", and thus it corresponds to a HTML *img* element; *tag* human input variable is of WeFlow data type "text", thus it corresponds to a HTML *input* element as described in Table 4.3.

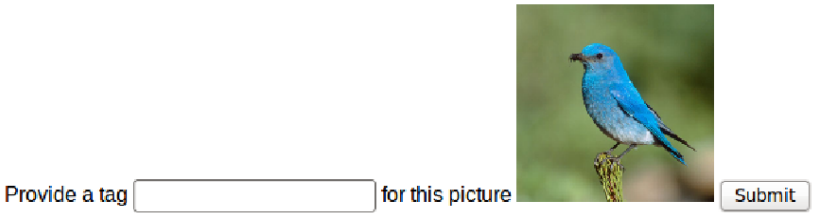


Figure 4.32. Tag a Bird Picture Human Task.

4.8.3. Generating a Web Application

In the previous section, it was discussed how to represent a human task on Web. A workflow consists of many tasks, and for each human task a Web page will be generated in terms of task description. The collection of these generated web pages will handle human interaction part and collect data from humans. Note that the execution semantics are not bind in this web application and are controlled by *WeFlow Execution Engine* which will be detailed in next section.

4.9. WeFlow Application Execution

In Section 4.8, it was detailed how to generate human computation web applications. In this section, execution semantics of these web applications will be discussed. In Figure 4.33, WeFlow web application stack is depicted. Each of these layers will be described in following sections.

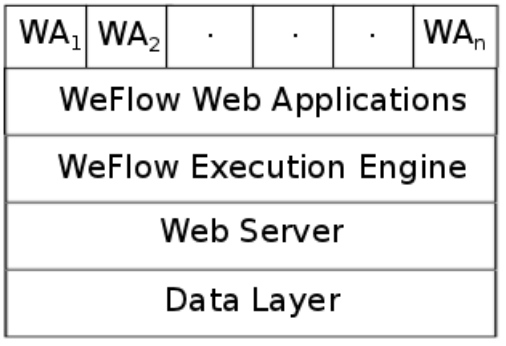


Figure 4.33. WeFlow Web Application Stack.

4.9.1. Data Layer

Data layer is the very bottom layer of *WeFlow Web Application Stack*. Data may come from (i) WeFlow Specifications, (ii) running human computation web applications, or external services such web services. WeFlow specifications have to be stored and/or fetched. WeFlow's running web applications provide valuable information that have to be kept and managed by workflow participants. This layer handles all data related operations, and provides data to upper layers.

4.9.2. Web Server

Web server is the bridge between *Data layer* and *WeFlow Execution Engine*. The main role of web server in this stack is to run the *WeFlow Execution Engine*.

4.9.3. WeFlow Execution Engine

WeFlow Execution Engine is the core of WeFlow framework. As discussed earlier, *WeFlow Application Generator* generates human computation web applications given WeFlow specifications. The execution of these Web applications is handled by this engine. Main roles of this engine are as the following:

4.9.3.1. Instantiation. After that a *WeFlow Specification* is loaded to the WeFlow framework, a human computation web application gets ready to start and may be instantiated by this engine. *WeFlow Execution Engine* also instantiates tasks by doing required data mappings, i.e. (human) input mappings and distributes tasks to workflow participants.

4.9.3.2. State Handling. By this way, at any time, it enables to track workflow participants' activities and also web applications' current state.

4.9.3.3. Control Flow Handling. Hence it enables to move the control from one task to another. In a workflow, next task to be executed is computed by this engine.

4.9.3.4. Task Handling. In another words, task executions are invoked by this engine. According to the performer of a task, it calls automated or human tasks for execution. After completion of a task execution, it does required data mappings, i.e. output mappings and updates current state information of the workflow.

5. IMPLEMENTATION

Our prototype is implemented in Python[18] language. Python runs on any operating system, is free to use and has an OSI-approved open source license. It has a very good documentation powered by Python community, and widely used in implementing web applications.

WeFlow Framework supports generation of human computation web applications running on *WeFlow Execution Engine*. In Section 4.9, execution semantics were detailed and web application stack was depicted. Data layer is based on ZODB [50] object database which will be explained is Section 5.2. *web.py* [51] is the python web server environment that *WeFlow Execution Engine* is based on. *web.py* is a Python web framework which is simple and powerful. It has clear semantics to understand and use.

Our prototype includes several important modules that will be detailed in following sections. They are depicted in Figure 5.1. Note that arrows show "uses" relationship between modules. For example, *WeFlow Application Generator* uses *WeFlow Specification Handler* and so on. Each module has a specific section to be discussed and it will be referred to this architecture in order to clarify relationships between modules.

5.1. WeFlow Specification Handler

In Section 5.1.1, *WeFlow Specification Language* implementation details will be discussed. A human computation web application is described in terms of this specification and uploaded to *WeFlow Framework*. It works together with *WeFlow Data Handler* and *WeFlow User Handler*. In Section 5.1.6, *Specification Handler* module methods will be explained.

[18] http://www.python.org/

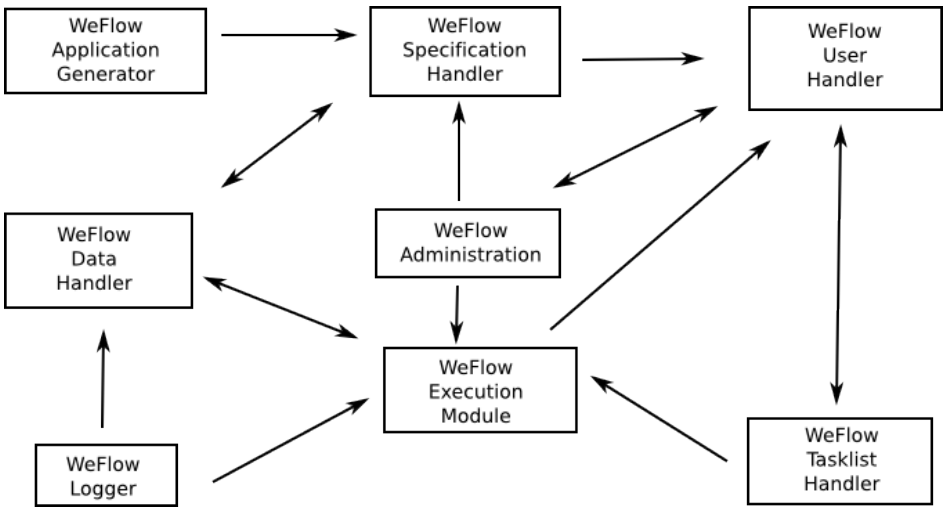


Figure 5.1. WeFlow Implementation Architecture.

5.1.1. WeFlow Specification Language

In Section 4.2, workflow components of our model were introduced. In order to specify a human computation web application, *WeFlow Specification Language* is developed using eXtensible Markup Language (XML), i.e. workflow components are described in XML structure.

XML is chosen because it is the most common tool for data transmissions between various applications. XML is designed to carry data and enables interoperability for all sort of applications. And most importantly, XML is a W3C recommendation [52].

There are no predefined tags to store data as seen in HTML tags, one must define own tags in order to keep data. XML is self-descriptive and mostly human readable. In Figure 5.2, an example of XML document is represented. XML tags are *book*, *name* and *author*, and *isbn* is an XML attribute giving information about *book* tag. In this example, it is clear that the name and the author of a book is represented.

Each WeFlow specification starts with a <*WeFlow*> tag and terminates with a </*WeFlow*> tag. In the following of this section, for each workflow component related

```
<book isbn='978-0452284234'>
        <name>Nineteen Eighty Four</name>
        <author>George Orwell</author>
</book>
```

Figure 5.2. XML Representation Example.

XML structure will be explained. Note that all these components are defined between *<WeFlow>* and *</WeFlow>* tags, see Figure 5.3.

```
<?xml version="1.0" encoding="UTF-8"?>
<WeFlow>
        .
        all other workflow related information goes here
        .
</WeFlow>
```

Figure 5.3. WeFlow Specification Structure.

5.1.2. General Description

A workflow is created for a specific purpose, and it is possible to introduce textual information about the workflow itself. Note that for a better understanding, XML structure of workflow components will be followed by concrete examples.

In Figure 5.4, XML structure of a workflow information is depicted. A WeFlow specification has some XML tags:

- a *description* tag to define description of a workflow,
- a *name* tag to provide a workflow name,
- a *start_task* tag to define starting task of a workflow by providing task id attribute,
- a *final_tasks* tag which is a bag for various final tasks. Each final task is defined with a *final_task* tag by providing task id attribute.

```
<name>...</name>
<description>...</description>
<start_task id="..."/>
<final_tasks>
    <final_task id="..."/>
    .
    .
</final_tasks>
```

Figure 5.4. Defining Workflow Information.

Workflow information of a human computation web application, CoStory, is depicted in Figure 5.5. Note that provided information is self-descriptive. In another words, this application's name is *CoStory: Writing Collaborative Stories*, its description is *Ready to create collaborative stories? Start using CoStory now!*, its start task is *create_story*, and there is one final task, *show_story*.

```
<name>CoStory: Writing Collaborative Stories</name>
<description>
        Ready to create collaborative stories? Start using CoStory now!
</description>
<start_task id="create_story"/>
<final_tasks>
    <final_task id="show_story"/>
</final_tasks>
```

Figure 5.5. Defining Workflow Information Example.

5.1.2.1. Tasks. Workflow tasks are defined between <*tasks*> and </*tasks*> tags. Each task is described between <*task*> and </*task*> tags. Thus main skeleton of *WeFlow Specification Language* becomes as Figure 5.6.

```
<?xml version="1.0" encoding="UTF-8"?>
<WeFlow>
        .
        <tasks>
                <task>...</task>
                .
                .
        </tasks>
        .
</WeFlow>
```

Figure 5.6. WeFlow Specification Structure with Tasks.

In Figure 5.7, XML structure of a general task is depicted. Each task has some XML attributes:

- an *id* attribute which is a unique name identifying a task,
- *performer* attribute which is either *human* or *auto*,
- *type* attribute which is one of *basic*, *conditional*, *repetition*, *collective* and *composite*.

Each task has some XML tags:

- a *name* tag representing the name of a task,
- a *description* tag representing the task description,
- a *params* tag which is a bag for input and output variables of a task. Each variable is described with a *param* tag which has some attributes such *type* which is either *input* or *output*, *name* which is the variable name, and *datatype* which is the data type of the variable.

Depending on the task type, there are additional XML tags and attributes. For various types of tasks, see Section 4.2, additional XML tags and attributes for these tasks will be explained.

```
<task id="..." performer="..." type="...">
        <name>...</name>
        <description>...</description>
        <params>
                <param type="..." name="..." datatype="..."/>
                .
                .
        </params>
</task>
```

Figure 5.7. WeFlow Specification: Task.

5.1.2.2. Basic Task. In Figure 5.7, all XML tags and attributes for a *Basic Task* are depicted. Note that task *type* is set as *basic*. In Figure 5.8, a basic human task example, *create_story*, is shown. Its name and description is provided, it has four variables: two input variables and two output variables.

```
<task id="create_story" performer="human" type="basic">
    <name>Create Story</name>
    <description>
        Your story title is: $title  And start your story: $story
    </description>
    <params>
        <param type="input" name="title" datatype="text"/>
        <param type="input" name="story" datatype="textarea"/>
        <param type="output" name="new_story" datatype="textarea"/>
        <param type="output" name="s_title" datatype="text"/>
    </params>
</task>
```

Figure 5.8. WeFlow Specification: Basic Task.

5.1.2.3. Conditional Task. *Conditional Task* model was detailed in Section 4.5.2, and there are three conditional task types: (i) *IfElse Task* and (ii) *Choice Task*. Each conditional task has *type* attribute equal to *condition*.

(i) In Figure 5.9, XML structure of a *IfElse Task* is depicted. Each *IfElse Task* has some XML attributes:

- *subtype* attribute which is equal to *ifelse*.
- *performer* attribute which is equal to *auto* or *human*.

Each *IfElse Task* has some XML tags:

- a *exprs* tag which is a bag for conditional expressions. Each expression is described via three XML tags: *condition*, *then* and *else*. *condition* tag has three attributes *operand1*, *operator* and *operand2*, which represent a conditional expression. *then* and *else* tags have a *taskid* attribute used to refer to a task.

```
<task id="..." performer="..." type="condition" sub="ifelse">
    <name>...</name>
    <description>...</description>
    <params>
        <param type="..." name="..." datatype="..."/>
        .
        .
    </params>
    <exprs>
        <expr>
            <condition operand1="..." operator="..." operand2="..."/>
            <then taskid="..."/>
            <else taskid="..."/>
        </expr>
    </exprs>
</task>
```

Figure 5.9. WeFlow Specification: IfElse Task Skeleton.

A *IfElse Task* example, *check_condition*, is shown in Figure 5.10. As mentioned in its description, this task is used to check given conditional expression, "$param1 > $param2". It has two variables: two input variables. One of the two tasks, *show_human_result* and *show_auto_result* , is executed depending on the satisfia-

bility of the conditional expression.

```
<task id="check_condition" performer="auto" type="condition" sub="ifelse">
    <name>Check sum operation results..</name>
    <description>check the condition</description>
    <params>
        <param type="input" name="param1" datatype="integer"/>
        <param type="input" name="param2" datatype="integer"/>
    </params>
    <exprs>
        <expr>
            <condition operand1="$param1" operator=">" operand2="$param2"/>
            <then taskid="show_human_result"/>
            <else taskid="show_auto_result"/>
        </expr>
    </exprs>
</task>
```

Figure 5.10. WeFlow Specification: DoAll Task.

(ii) In Figure 5.11, XML structure of a *Choice Task* is depicted. Each *Choice Task* has some XML attributes:

- *subtype* attribute which is equal to *case*.
- *performer* attribute which is equal to *auto* or *human*.

Each *Choice Task* has some XML tags:

- *expr* tag is a container for the conditional expression. Conditional expression is described via *condition* tag. *condition* tag has three attributes *operand1*, *operator* and *operand2*, which represent a conditional expression.
- a *tasklist* tag which is a bag for various tasks. Each task is referred using a *toTask* tag with task *id* attribute specified.

A human *Choice Task* example, *check_condition*, is shown in Figure 5.12. As mentioned in its description, this task is used to choose between tasks. It has one variable: one input variable. One of the two tasks, *show_human_result* and *show_auto_result* , is executed depending on the human choice which is the con-

```
<task id="..." performer="..." type="condition" subtype="case">
    <name>...</name>
    <description>...</description>
    <params>
        <param type="..." name="..." datatype="..." />
        .
        .
    </params>
    <expr>
        <condition operand1="..." operator="..." operand2="..."/>
        <tasklist>
            <toTask id="..."/>
            .
            .
        </tasklist>
    </expr>
</task>
```

Figure 5.11. WeFlow Specification: Choice Task Skeleton.

ditional expression, *$param1*.

```
<task id="check_condition" performer="human" type="condition" sub="case">
    <name>Check sum operation results..</name>
    <description>Choose one the following tasks:</description>
    <params>
        <param type="input" name="param1" datatype="integer"/>
    </params>
    <expr>
        <condition operand1="$param1" operator="" operand2=""/>
        <tasklist>
            <toTask id="show_human_result"/>
            <toTask id="show_auto_result"/>
        </tasklist>
    </expr>
</task>
```

Figure 5.12. WeFlow Specification: Choice Task.

5.1.2.4. DoAll Task. In Figure 5.13, XML structure of a *DoAll Task* is depicted. Each *DoAll Task* has some XML attributes:

- *performer* attribute equal to *auto*,
- *subtype* attribute which is equal to *doall*.

Each *DoAll Task* has some XML tags:

- a *tasklist* tag which is a bag for various tasks. Each task is referred using a *toTask* tag with task *id* attribute specified.

A *DoAll Task* example, *do_sum*, is shown in Figure 5.14. As mentioned in its description, this task is used to do the sum operation of two integers. It has four variables: two input variables and two output variables. There are two tasks to execute

```
<task id="..." performer="auto" type="condition" subtype="doall">
    <name>...</name>
    <description>...</description>
    <params>
        <param type="..." name="..." datatype="..."/>
        .
        .
    </params>
    <tasklist>
        <toTask id="..."/>
        .
        .
    </tasklist>
</task>
```

Figure 5.13. WeFlow Specification: DoAll Task Skeleton.

as specified in its tasklist, *do_sum_operation* and *do_sum_auto*.

5.1.2.5. Repetition Task. *Repetition Task* model was detailed in Section 4.5.4. In Figure 5.15, XML structure of a *Repetition Task* is depicted. Each *Repetition Task* has some XML attributes:

- *performer* attribute equal to *auto*,
- *type* attribute equal to *repetition*.

Each *Repetition Task* has some XML tags:

- *expr* tag is a container for the conditional expression. Conditional expression is described via *condition* tag. *condition* tag has three attributes *operand1*, *operator* and *operand2*, which represent a conditional expression. The task to be repeated is described via *repeat* tag where *taskid* is specified.

```
<task id="..." performer="auto" type="condition" subtype="doall">
    <name>Do Sum</name>
    <description>this function sum two integers..</description>
    <params>
        <param type="input" name="no1" datatype="integer"/>
        <param type="input" name="no2" datatype="integer"/>
        <param type="output" name="result" datatype="integer"/>
        <param type="output" name="result2" datatype="integer"/>
    </params>
    <tasklist>
        <toTask id="do_sum_operation"/>
        <toTask id="do_sum_auto"/>
    </tasklist>
</task>
```

Figure 5.14. WeFlow Specification: DoAll Task.

```
<task id="..." performer="auto" type="repetition">
    <name>...</name>
    <description>...</description>
    <params>
        <param type="..." name="..." datatype="..."/>
        .
        .
    </params>
    <expr>
            <condition operand1="..." operator="..." operand2="..."/>
            <repeat taskid="..."/>
    </expr>
</task>
```

Figure 5.15. WeFlow Specification: Repetition Task Skeleton.

A *Repetition Task* example, *rep_update_story*, is shown in Figure 5.16. As mentioned in its description, this task is used to repeat update story task. It has three variables: two input variables and one output variable. The conditional statement is *$is_finished != 'yes'*, and while this condition is satisfied, *update_story* task is repeated.

```
<task id="rep_update_story" performer="auto" type="repetition">
    <name>repeat</name>
    <description>repeat update story task</description>
    <params>
        <param type="input" name="story" datatype="textarea"/>
        <param type="input" name="is_finished" datatype="yes/no"/>
        <param type="output" name="result_story" datatype="textarea"/>
    </params>
    <expr>
        <condition operand1="$is_finished" operator="!=" operand2="'yes'"/>
        <repeat taskid="update_story"/>
    </expr>
</task>
```

Figure 5.16. WeFlow Specification: Repetition Task.

5.1.2.6. Collective Task. *Collective Task* model was detailed in Section 4.5.5. In Figure 5.17, XML structure of a *Collective Task* is depicted. Each *Collective Task* has some XML attributes:

- *performer* attribute equal to *auto*,
- *type* attribute equal to *collective*,
- *cardinality* attribute equal to an integer value or may be unbound.

Each *Collective Task* has some XML tags:

- *collectedTask* tag is used to define a task to execute. *id* attribute is used to refer to a task.

```
<task id="..." performer="auto" type="collective" cardinality="...">
    <name>...</name>
    <description>...</description>
    <params>
        <param type="..." name="..." datatype="..."/>
        .
        .
    </params>
    <collectedTask id="..."/>
</task>
```

Figure 5.17. WeFlow Specification: Collective Task Skeleton.

A *Collective Task* example, *do_sum_collective*, is shown in Figure 5.18. As mentioned in its description, this task is used to collect data from *do_sum_operation* task. It has three variables: two input variables and one output variable. *do_sum_operation* task is done five times as defined in cardinality attribute.

```
<taskid="do_sum_collective"performer="auto"type="collective"cardinality="5">
    <name>Do Sum Operation</name>
    <description>collect data from sum operation task</description>
    <params>
        <param type="input" name="no1" datatype="integer"/>
        <param type="input" name="no2" datatype="integer"/>
        <param type="output" name="results" datatype="text[]"/>
    </params>
    <collectedTask id="do_sum_operation"/>
</task>
```

Figure 5.18. WeFlow Specification: Collective Task.

5.1.2.7. Composite Task. *Composite Task* model was detailed in Section 4.5.6. *Composite Task* has not been implemented yet, and remains as part of our future work.

5.1.3. Control Flow Description

In Section 4.7.1, control flow model was detailed. In Figure 5.19, XML structure of a *Control Flow* is depicted. It has no XML attributes but has a *controlFlow* tag containing a list of tasks and each task is defined by a *task* tag using *id* attribute.

```
<controlFlow>
    <task id="..."/>
    .
    .
</controlFlow>
```

Figure 5.19. WeFlow Specification: Control Flow Skeleton.

A *Control Flow* example is depicted in Figure 5.20. This example shows the sequence of three tasks: *create_story*, *rep_update_story* and *show_story*.

```
<controlFlow>
    <task id="create_story"/>
    <task id="rep_update_story"/>
    <task id="show_story"/>
</controlFlow>
```

Figure 5.20. WeFlow Specification: Control Flow.

5.1.4. Data Flow Description

In Section 4.7.2, data flow model was detailed. There are three types of data mappings: (1) task input mappings, (2) human input mappings and (3) output mappings. There is no difference in implementation of task input mappings and human input mappings thus it will be simply mentioned as *Input Mappings*. In Figure 5.21, XML structure of a *Data Flow* is depicted. There are input mappings and output mappings which should be defined in a *dataFlow* tag.

In Figure 5.22, XML structure of a *Input Mappings* is depicted. A *mapping* tag

```
<dataFlow>
    <outputMappings>
        .
        .
        .
    </outputMappings>

    <inputMappings>
        .
        .
        .
    </inputMappings>
</dataFlow>
```

Figure 5.21. WeFlow Specification: Data Flow Skeleton.

is used to describe an incoming mapping to a task which is set by *taskID* attribute. An *inputParam* tag is used, and it has two attributes: *name* and *expression*. *name* attribute is used to refer to an input variable, and *expression* attribute is used to specify the expression, which evaluates to a value in its turn, to be mapped. Note that *:=* operator is used to show a value assignment.

An *Input Mapping* example is depicted in Figure 5.23. This example shows input mappings for two tasks: *update_story* and *show_story*. For *update_story* task, there is one input mapping incoming to *story* variable of *update_story* task. And it gets its value from *story* variable of *rep_update_story* task. For *show_story* task, there are two input mappings: (1) incoming to *story_title* variable of *show_story* task which is getting its value from *s_title* variable of *create_story* task, (2) incoming to *story* variable of *show_story* task which is getting its value from *result_story* variable of *update_story* task.

In Figure 5.24, XML structure of a *Output Mappings* is depicted. A *mapping* tag is used to describe an outgoing mapping to a task which is set by *taskID* attribute.

```
<inputMappings>
    <mapping taskID="...">
        <inputParam name="..." expression=":= ..."/>
        .
        .
    </mapping>
    .
    .
</inputMappings>
```

Figure 5.22. WeFlow Specification: Data Flow Input Mappings Skeleton.

```
<inputMappings>
  <mapping taskID="update_story">
    <inputParam name="story" expression=":= $rep_update_story.story"/>
  </mapping>
  <mapping taskID="show_story">
    <inputParam name="story_title" expression=":= $create_story.s_title"/>
    <inputParam name="story" expression=":= $update_story.result_story"/>
  </mapping>
</inputMappings>
```

Figure 5.23. WeFlow Specification: Data Flow Input Mappings.

An *outputParam* tag is used, and it has two attributes: *name* and *expression*. *name* attribute is used to refer to an output variable, and *expression* attribute is used to specify the expression, which evaluates to a value in its turn, to be mapped. Note that *expression* tag should include either *:=* operator (assignment) or *:+* operator (append).

```
<outputMappings>
    <mapping taskID="...">
        <outputParam name="..." expression="..."/>
        .
        .
    </mapping>
    .
    .
</outputMappings>
```

Figure 5.24. WeFlow Specification: Data Flow Output Mappings Skeleton.

An *Output Mapping* example is depicted in Figure 5.25. This example shows output mappings for two tasks: *do_sum_operation* and *do_sum_collective*. For *do_sum_operation* task, there is one output mapping outgoing to *result* variable of *do_sum_operation* task. And it gets its value from *humanResult* variable of *do_sum_operation* task. For *do_sum_collective* task, there is one output mapping outgoing to *results* variable of *do_sum_collective* task, *result* variable of *do_sum_operation* task is appended to this output variable, *results*.

5.1.5. Human / Groups Description

In Section 4.6, resourcing model was detailed. In Figure 5.26, XML structure of*resourcing* is depicted. A *resourcing* tag is used to allow some groups to execute a task. Tasks are defined by the use of *task* tag with *id* attribute, and for each task a list of groups is defined with *groups tag*. Each group is described with *group* tag for which a *name* tag is specified.

A *resourcing* example is depicted in Figure 5.27. There is one task, *create_story*

```
<outputMappings>
 <mapping taskID="do_sum_operation">
   <outputParam name="result" expression=":=$do_sum_operation.humanResult"/>
 </mapping>
 <mapping taskID="do_sum_collective">
   <outputParam name="results" expression="+= $do_sum_operation.result"/>
 </mapping>
</outputMappings>
```

Figure 5.25. WeFlow Specification: Data Flow Output Mappings.

```
<resourcing>
    <task id="...">
        <groups>
            <group>
                <name>...</name>
            </group>
        </groups>
    </task>
    .
    .
</resourcing>
```

Figure 5.26. WeFlow Specification: Resourcing Skeleton.

for which one group definition is defined as *anyone*.

```
<resourcing>
    <task id="create_story">
        <groups>
            <group>
                <name>anyone</name>
            </group>
        </groups>
    </task>
</resourcing>
```

Figure 5.27. WeFlow Specification: Resourcing.

5.1.6. Specification Handler Methods

Specification Handler module has some important methods:

- *create_new_spec* is used to create a new WeFlow specification,
- *parse_spec* is used to parse a WeFlow specification described with XML. This method uses other methods fro parsing operation:
  - (i) *parse_wf_info* parses basic workflow information,
  - (ii) *parse_tasks* parses information about tasks,
  - (iii) *parse_controlflow* parses control flow information,
  - (iv) *parse_dataflow* parses data flow information,
  - (v) *parse_resourcing* parses resourcing information.
- *verify_spec* is used to verify if given WeFlow specification has a valid syntax.
- *delete_spec* is used to delete an existing WeFlow specification.
- *modify_spec* is used to modify an existing WeFlow specification.

verify_spec, *delete_spec*, *modify_spec* methods have not been implemented yet.

Specification Handler module is programmed in Python, and in order to parse XML documents a built-in Python library, *xml.dom.minidom*, is used. This library

is a light-weight implementation of the Document Object Model (DOM) interface. It provides simple functions to parse XML documents.

5.2. WeFlow Data Handler

In this section, data perspective of *WeFlow Framework* will be explained. This module works together with *WeFlow Specification Handler* and *WeFlow Execution Module*.

5.2.1. A Python Object Database: ZODB

One of the difficulties of object oriented programming is that there is a need to store objects into tables in databases. For this purpose, data is transformed to raw data in order to be stored in a table, and raw data is transformed to objects in order to be used by object programming languages. Computation cost for these transformations is high and effects badly the performance of a system. As a solution to this common problem, there are some object databases that allow to store objects directly in a database. So it gives the freedom to work in an environment where there is no need to map object oriented code to relational model stucked in schemas.

ZODB is a native object database for Python and is widely used by Python programmers [50]. For data storing purposes, ZODB is used as part of data layer of *WeFlow Framework*.

5.2.2. Specification Data Handler

This module is used to fetch and/or store data related to *WeFlow Specification*. It is a layer between ZODB and *WeFlow Framework*. Two main methods are as the following:

- *add_workflow_desc* is used to store a WeFlow specification.
- *get_workflow_x* methods are used to fetch x related data where x is one of *speci-*

fication, *tasks*, *controlflow*, *dataflow* and *resourcing*.

5.2.3. Application Data Handler

This module is used to fetch and/or store data related to running human computation web applications. It is a layer between ZODB and *WeFlow Framework*. Main methods are as the following:

- *add_wf_instance* adds a workflow instance.
- *update_wf_instance* updates a workflow instance.
- *get_wf_x* where x is one of:
  - (i) *instances* gets all workflow instances.
  - (ii) *instance* gets a specific workflow instance.
  - (iii) *task_instances* gets workflow task instances.
  - (iv) *task_instance* gets a specific workflow task instance.
- *get_ongoing_workflows* returns all ongoing workflows.
- *prepare_task_instance* gets ready a task instance by doing its data mappings via *do_start_mappings* method.
- *finish_task_instance* terminates a task instance by doing its data mappings via *do_end_mappings* method.

5.3. WeFlow User Handler

All user related methods are part of this module. This module is used for creating a new user, registering, authenticating a user, checking access rights, deleting and/or modifying a user, getting user workflows ongoing or terminated and setting access rights.

5.4. WeFlow Application Generator

This module generates a human computation web application using WeFlow specification. There are two main classes: (i) HTML Data Generator and (ii) Application

Table 5.1. Mapping Data Types to web.py Form Elements.

WeFlow Data Type	web.py Form Element
Text	web.form.Textbox
Integer	web.form.Textbox
Password	web.form.Password
Image	web.form.Input
Paragraph	web.form.Dropdown
Multiple Choice	web.form.Dropdown
Checkbox	web.form.Checkbox

Template Generator.

5.4.1. HTML Data Generator

Mapping from WeFlow specification data to HTML components is done by methods this class:

- *datatype_to_html* is used to map task variables to HTML form elements. Only some WeFlow datatypes are implemented such as *integer*, *text*, *checkbox*, *integer[]*, *text[]*, *textarea* and *image*. As *web.py* [51] is chosen as the webserver for *WeFlow Framework*, this method maps WeFlow datatypes to web.py form elements as described in Table 5.1. web.py allows the usage of regular expressions, thus it is possible to define *integer* datatype using a textbox element.
- *desc_to_html* is used to map task description to HTML text. As discussed in Section 4.8, a task description may contain a special character $ to refer to task variables. In this case, this method finds out the referred task value and replaces it in the description.

5.4.2. Application Template Generator

This module creates human computation web application folder structure, templates and files for execution. Main methods are as the following:

- *generate_form* is used to generate an HTML form for a given task.
- *generate_webapp_structure* is used to create a web application space for newly uploaded WeFlow specifications. Default naming mechanism is "./webfiles/web_apps/*wfname_wfid*" where *wfname* and *wfid* are workflow name and workflow id respectively.
- *generate_template* is used to generate HTML templates in order to display task data. Note that HTML templates are some generic structured files that can be used dynamically. *web.py* uses its own templating system thus this method generates *web.py* friendly HTML templates.
- *generate_root_file* is used to generate controller semantics in order to respond to HTML requests. Note that each human computation web application has its own generated controller. Default naming mechanism is "./webfiles/web_apps/*wfname_wfid*/app_/*wfname_wfid*.py" where *wfname* and *wfid* are workflow name and workflow id respectively.

Hence in order to generate a web application, a web application space is created, controller file is generated and HTML templates are set. This module works with some other libraries. *htmlDoc* library is used in order to create HTML templates given HTML tags. *PyGen* library is used in order to generate python files including python code. As indentation is a main issue in Python programming, the use of such a library is needed.

5.5. WeFlow Execution Module

This module is the core module of *WeFlow Framework*, it instantiates workflows, executes all web applications, distibutes tasks to various workflow participants and aggregate responses, keeps track of the state of every web application and user participant. These are done by some submodules of *Execution Module* which will be detailed now.

5.5.1. State Handler

This module keeps track of state information of (i) users, (ii) web applications. Note that each human computation web application has various instances executed by various user participants. Given a workflow instance, this module shows current state of a web application and a user participant. State information is important as it will tell *Task Handler* which task to execute next.

There is one method implemented for this purpose, *app_state_handler*. This function computes current state of web application and calls *Task Handler* module for execution of next task. Algorithm for this function can be found in Figure 5.28.

5.5.2. Task Handler

This module call tasks for execution. It gets next task information from *Control Flow Handler* module, checks type of next task and then call appropriate method. Note that for each task type there is an implemented call function. After execution, it calls *State Handler* module for updating state info of application instance and workflow participant. Algorithm for *call_task* function can be found in Figure 5.29.

5.5.3. Control Flow Handler

This module calls *State Handler* to check current state of an application instance. According to control flow semantics of the application instance, it decides next task to execute via *get_next_construct* method. Algorithm for this function can be found in Figure 5.30. It calls *User Handler* module to check access rights of current user participant. And finally next task is executed by *Task Handler* module.

5.5.4. Instantiator

This module basically instantiates a web application once a new WeFlow specification is loaded to the *WeFlow Framework* and started by a user participant. This

```
Require: wfid, wfinstanceid, taskname, taskinstanceid, userid, next_taskname;
  wfinstance ⇐ get_wf_instance(wfid, wfinstanceid);
  if wfinstance current state is set then
    if taskname is set then
      taskinstance ⇐ get_wf_task_instance(wfinstanceid, taskname, taskinstan-
      ceid);
      if userid is set then
        update user state;
        set user as owner of taskinstance;
      end if
      mark task instance as completed;
      if taskname is an end state AND next_taskname is not set then
        mark wfinstance as terminated
      end if
    end if
    if wfinstance is not terminated then
      call TaskHandler with wfid, wfinstanceid, next_taskname
    end if
  end if
```

Figure 5.28. Algorithm for *app_state_handler* method.

```
Require: wfid, wfinstanceid, taskname, next_taskname, next_tasktype;
  cfh ⇐ ControlFlowHandler()
  next_taskname, tasktype ⇐ get_next_construct(wfid, wfinstanceid,
  next_taskname)
  if next_tasktype is set then
    wfinstance ⇐ get_wf_instance_byparam1(wfinstanceid)
    if next_tasktype is "human" then
      wfdesc ⇐ get_workflow_desc(wfid)
      call_human_task(wfdesc.name, wfid, wfinstanceid, next_taskname)
    else if next_tasktype is "auto" then
      call_automated_task(wfinstanceid, next_taskname)
    else if next_tasktype is "condition" then
      call_condition_task(wfinstanceid, next_taskname)
    else if next_tasktype is "collective" then
      call_collective_task(wfinstanceid, next_taskname)
    else if next_tasktype is "repetition" then
      call_repetition_task(wfinstanceid, next_taskname)
    end if
  end if
```

Figure 5.29. Algorithm for *call_task* method.

```
Require: wfid, wfinstanceid, next_taskname;
  wfinstance ⇐ get_wf_instance(wfid, wfinstanceid);
  controlflow ⇐ get_workflow_controlflow(wfid);
  wfdesc ⇐ get_workflow_desc(wfid);
  if current workflow instance state is not set then
    wfinstance.current_state ⇐ wfinstance.init_state
  else
    taskname ⇐ wfinstance.current_state
  end if
  if next_taskname is not set then
    taskindex ⇐ taskname index in controlflow;
    model_task ⇐ get_workflow_task(wfdesc, taskname);
    if model_task is instance of CollectiveTask OR DoAllTask OR RepetitionTask
    then
      temp_task ⇐ get_wf_task_last_instance(wfinstanceid, taskname);
      if temp_task is not completed then
        next_taskname ⇐ taskname;
      else
        next_taskindex ⇐ taskindex + 1;
      end if
    else
      next_taskindex ⇐ taskindex + 1;
    end if
  end if
  if next_taskname in controlflow then
    wfinstance.current_state ⇐ next_taskname
  end if
  tasktype ⇐ get_next_ïtasktype(wfid, next_taskname)
  return next_taskname, tasktype
```

Figure 5.30. Algorithm for *get_next_construct* method.

module also takes care of the instantiation of tasks in web applications.

5.6. WeFlow Tasklist Handler

This module shows tasks to users via notifications. Users are involved in some workflows, and get notifications related to these workflows. They find a list of tasks to execute and also a list of executed tasks to display history information such who performed which task and when. Tasklist Handler gets this information from *WeFlow Execution Module* and *WeFlow User Handler*.

5.7. WeFlow Administration

This module is the administration part of WeFlow Framework. One can manage, monitor, modify workflow instances by using this nodule. It works together with *WeFlow Execution Module*, *WeFlow Specification Handler* and *WeFlow User Handler*.

5.8. Logger

This module is used to log all activities within WeFlow Framework. As this framework handles a huge number of users, their activities are stored for further processing. This module works together with *WeFlow Data Handler* and *WeFlow Execution Module*.

6. USE CASES

6.1. A Human Computation Web Application: CoStory

In this section, a human computation example is illustrated. It is a simple application, the kind that an ordinary person could think of and want to define. Note that this application is part of a larger application, and only small part of it is presented for purposes of clarity.

Carine is an elementary school teacher who wants to encourage her students to collaborate. By the use of WeFlow framework, she builds a simple application, CoStory, to let her students write stories together. Carine, the *Specifier* of CoStory, decides to design her application with three main tasks: *Create Story*, *Update Story* and *Show Story*. A story should be created, then updated many times by her students, and displayed at the end. And all these tasks should only be executed by her students.

CoStory application was deployed in Karagözyan Elementary School and used by 5th grade students who created a story together using this application.

6.1.1. CoStory Specification

In Figure 6.1, CoStory is depicted in terms of WeFlow tasks, and control flows. For clarity, data flow information is not depicted but for a better understanding of data mappings, similar or exact task variable names are used in the example. Human tasks are denoted with a human figure on top of the task.

In *Create Story* task, a *story* and a story *title* is provided by a student. In *Update Story* task, current stage of the *story* is shown to a student who provides a new *entry* for the story and declares whether the story is finished or not, *fin*. While the story is not finished, *Update Story* task is repeated by students. And once story is finished, *title* and *story* is displayed by *Show Story* task. In this example, data mappings are

defined as explained in Section 4.7.2.

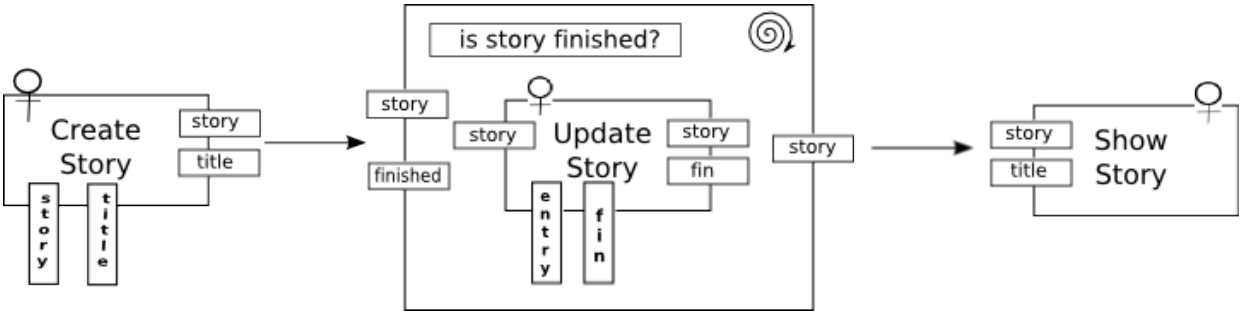


Figure 6.1. CoStory – A Human Computation Web Application.

6.1.2. CoStory Web Application Generation

In Section 4.8, it was discussed how to generate a human computation web application given a WeFlow specification. So, each human task of CoStory will be represented on Web in order to be performed. Task descriptions provided by Carine are as the following:

(i) for *Create Story* task: "Your story title is: $title And start your story: $story"

(ii) for *Update Story* task: "This is how the story is $story Add your words now! $entry Do you think that the story is finished? $fin"

(iii) for *Show Story* task: "Story title: $title Your Story: $story".

6.1.3. CoStory Web Application Execution

In this paragraph, *WeFlow Execution Engine* properties will be discussed as introduced in Section 4.9. In Figure 6.2, CoStory specification is loaded to WeFlow framework and ready to run. This specification can be found in Appendix A.

First, a student starts CoStory application by pressing *Start Workflow* button. At this point, engine instantiates CoStory web application, and loads CoStory specification in order to get application semantics. It checks the control flow, finds which task to execute, *Create Story*, and prepare it for execution by instantiating it and doing its input mappings. Current state of the application is marked as *Create Story* task.

A student executes *Create Story* task as depicted in Figure 6.3. *Create Story* task description is shown to this student via a *put* channel, then s/he provides *title* and *story* data, which are received by a *get* channel once s/he presses *Submit* button. WeFlow engine does output mappings for *Create Story* task and terminates its execution.

WeFlow engine checks again the control flow, and detects that the next task is a *Repetition Task*. Instantiation, state handling, and data mappings are computed in the same way as described previously. For *Repetition Task* case, engine checks the condition, if it is fulfilled then it executes the subtask, *Update Story*. *Update Story* task gets ready to execute and performed by a student, see Figure 6.4. As the story is not finished, *Update Story* task continues to execute, see Figure 6.5. Since now the story is marked as finished by a student, control flow is moved to task *Show Story* which in its turn displays the story and its title, Figure 6.6. Once a student presses *Finish* button, WeFlow engine terminates execution of current CoStory web application instance.

Carine and her students are now able to create as many as stories they want, and enjoy collaborating by writing stories together.

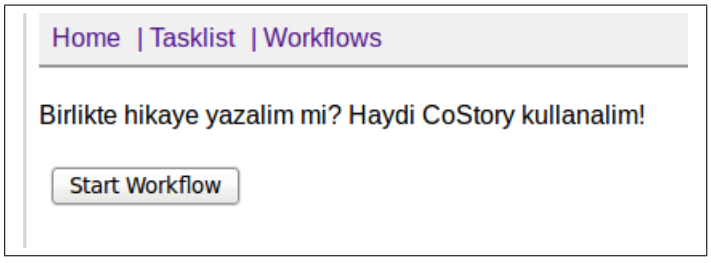


Figure 6.2. CoStory – Starting the Web Application.

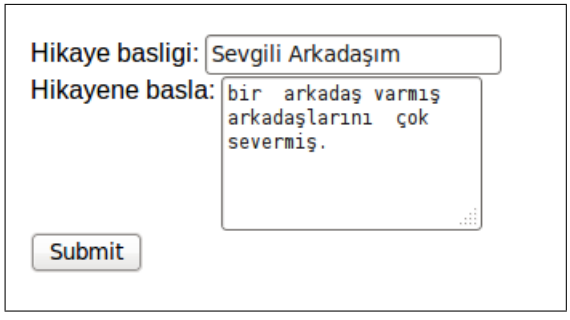


Figure 6.3. CoStory – Starting a Story.

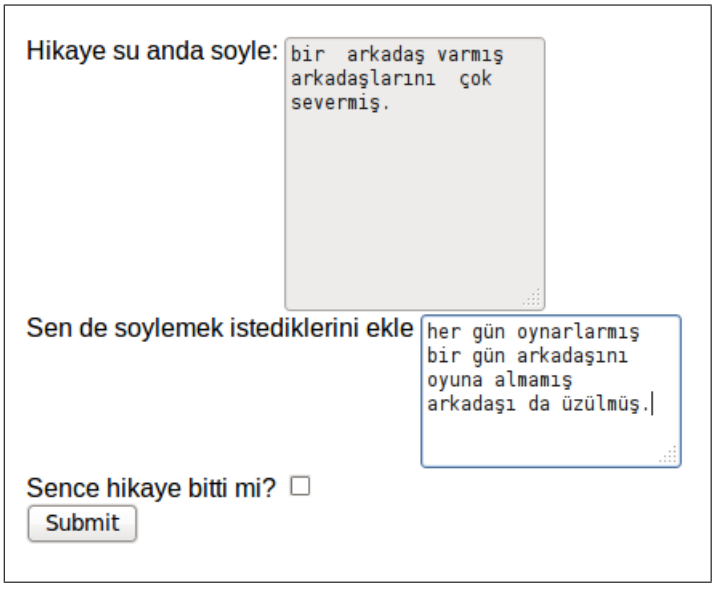


Figure 6.4. CoStory – Updating a Story.

6.2. A Human Computation Web Application: Hisarustu Accessibility

Ali is a student living in Rumelihisarustu and wants to create a place where people in the neighborhood can share information about Rumelihisarustu. In Rumelihisarustu project, the idea is to increase the life quality in Rumelihisarustu by reporting things in a bad condition in order to be fixed by anyone. In other words, this project will make Rumelihisarustu accessible to anyone living there. People simply may share information about anything to be fixed, or may see and update reported information by other users.

6.2.1. Hisarustu Accessibility Specification

In Figure 6.7, Rumelihisarustu Accessibility is depicted in terms of WeFlow tasks, and control flows. For clarity, data flow information is not depicted. Human tasks are denoted with a circle figure on top of the task, and different colors show that users belong to different groups.

In *Choose Task* task, a user can do two things: (i) she can share an information about an item. She provides a picture of the item, gives some information about that item and reports current status of the item which is 'resolved' or 'pending', (ii) she

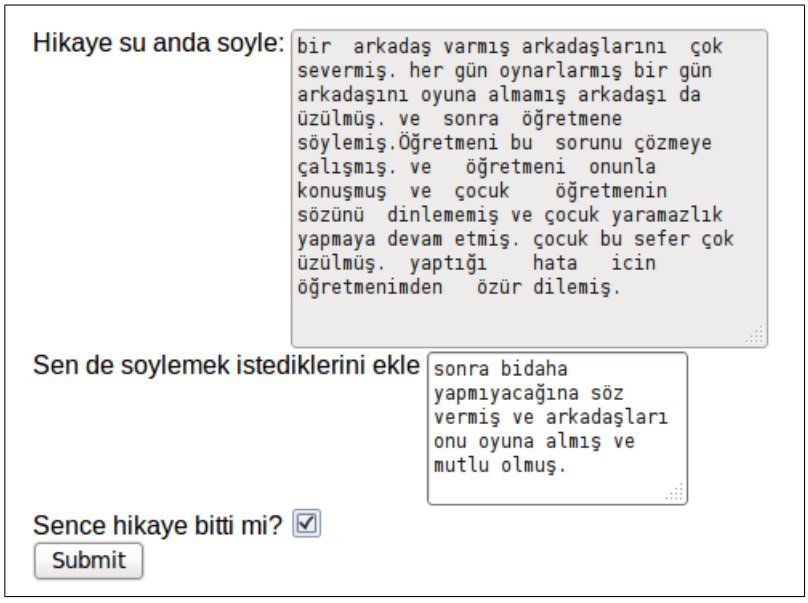


Figure 6.5. CoStory – Finishing a Story.

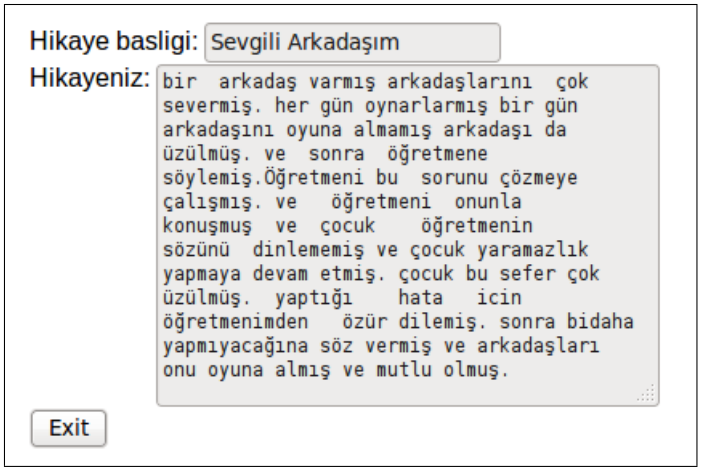


Figure 6.6. CoStory – Displaying Finished Story.

can list current items reported by other users. *List Items* task is an automated task gathering this information from the data storage. In case (ii), *Show Items* task shows current items to the user and user picks one item from the list which is provided by the previous task, *List Items*. WeFlow gets selected item from data storage, *Pick Item*, and related information to this item is shown to the user who, in her turn, may update status information of current item, *Update Item*. A volunteer from control group checks updated information and decides whether provided information is correct or not, *Control Item*. Then, the item is updated according to *Control Item* task's output information and saved by *Save Item* task. Updated information is shown to the user by *Show Item* task.

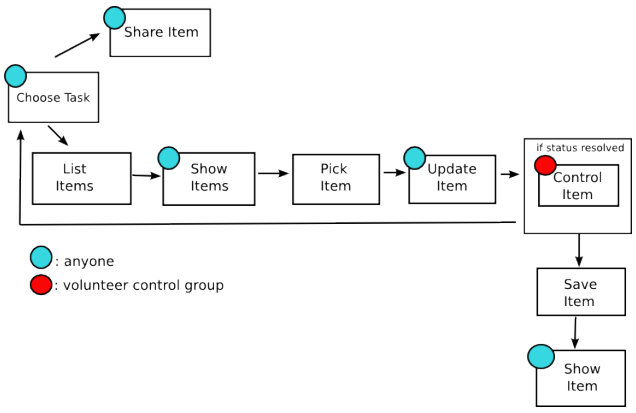


Figure 6.7. Rumelihisarustu Specification.

6.2.2. Hisarustu Accessibility Web Application Generation

Task descriptions provided by Ali are as the following:

(i) for *Choose Task* task: "Choose one task from the menu on the left"

(ii) for *Share Item* task: "Give a picture url: $pic_url . Enter some information: $information related to provided picture. Current status of provided item is $status"

(iii) for *Show Items* task: "Pick one accessibility item by providing its item id number: $itemno"

(iv) for *Update Item* task: "For $pic some information $information is provided. Current status is $status . Is the status is updated? Please type 'resolved' or 'pending' as a new status $hstatus"

(v) for *Control Item* task: "For $pic some information $information is provided. Someone said that current status for that item is $status . If so type 'resolved' or 'pending' as a new status $hstatus"

(vi) for *Show Item* task: "For $pic some information $information is provided. Current status is $status"

6.2.3. Rumelihisarustu Web Application Execution

Rumelihisarustu specification is loaded to WeFlow framework and ready to run, see Figure 6.8. Once the application is started by pressing *Start Workflow* button, the

execution engine instantiates the application, loads Rumelihisarustu specification in order to get application semantics. It finds which task to execute next, *Choose Task*, and prepare it for execution by instantiating it and doing its input data mappings. Current state of the application is marked as *Choose Task* task. A user executes this task as depicted in Figure 6.9, task description informs the user who, in her turn, provides some data in order to proceed in the workflow. She picks 'share_info', which is *Share Item* task in the specification, to execute next as depicted. Figure 6.10 shows *Share Item* web task, task description is shown and user provides some data: a picture url, some textual information and status of the shared item. Other users are also using this application for sharing new items.

Another user joins the application and picks *List Items* task in order to execute, see Figure 6.15, this time the execution engine checks the control flow, and detects that the next task is a *Show Items*. Instantiation, state handling, and data mappings are computed in the same way as described previously. Figure 6.11 shows *Show Items* task, note that there are currently two listed items. The user picks one item by providing an accessibility item id. *Update Item* task is instantiated by the engine which is depicted in Figure 6.12. The user is able to update status field of that item. This updated information is controlled by someone from *volunteer control group* who checks whether provided information is correct, see *Control Item* task in Figure 6.13. This updated item is shown, *Show Item* task, to any user from *anyone* group as depicted in Figure 6.14.

Another user joins the application and wants to see shared information contributed by other users. Figure 6.16 shows current list of accessibility items. Note that the status information of the first item is modified to 'resolved' as reported by a user of the system.

6.3. A Simple Web Application for Personal Use: Jack Summing Integers

Jack needs to practice basic sum operation of two given numbers. He wants to know if his answer is correct or not. And this is such a simple application that he

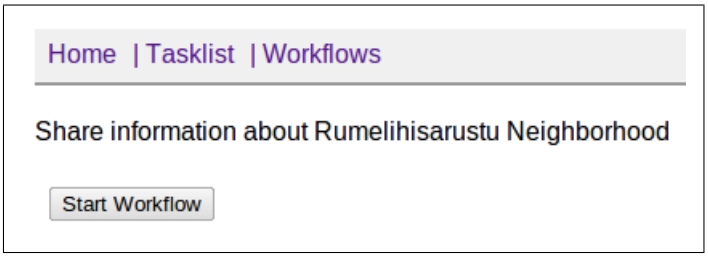


Figure 6.8. Rumelihisarustu – Workflow Ready to Start.

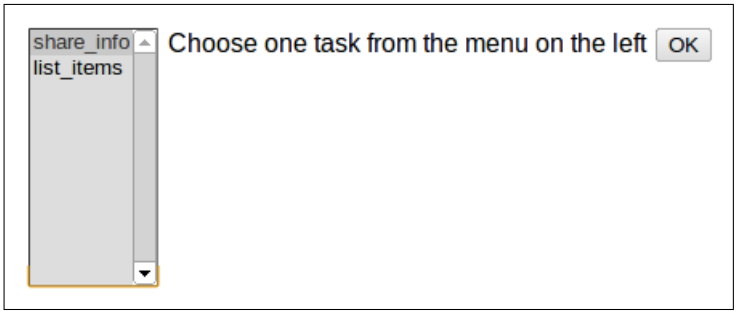


Figure 6.9. Rumelihisarustu – Choose Task.

wants to create.

First, he thinks of basic steps of this application:

(i) pick two random numbers: Generate Numbers
(ii) ask Jack to sum these numbers: Do Sum Operation
(iii) ask computer to sum these numbers: Compute Sum
(iv) compare Jack's answer with computer's answer: Compare
(v) notify Jack about the correctness of his answer: Notify User

Figure 6.17 depicts Jack's application in terms of tasks and its control flow. Note that there are three automated tasks: *Generate Numbers* Figure 6.18, *Compute Sum* Figure 6.19 and *Compare* Figure 6.20 and two human tasks: *Do Sum Operation* and

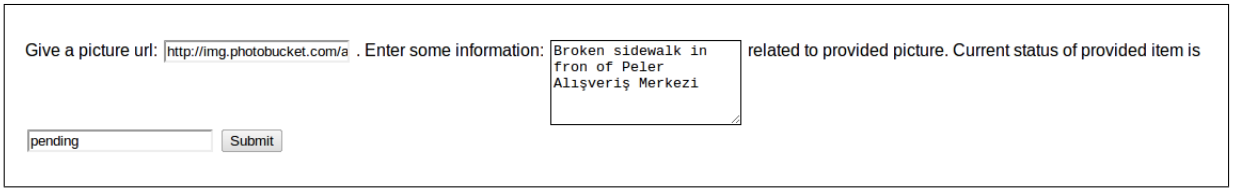


Figure 6.10. Rumelihisarustu – Share Item.

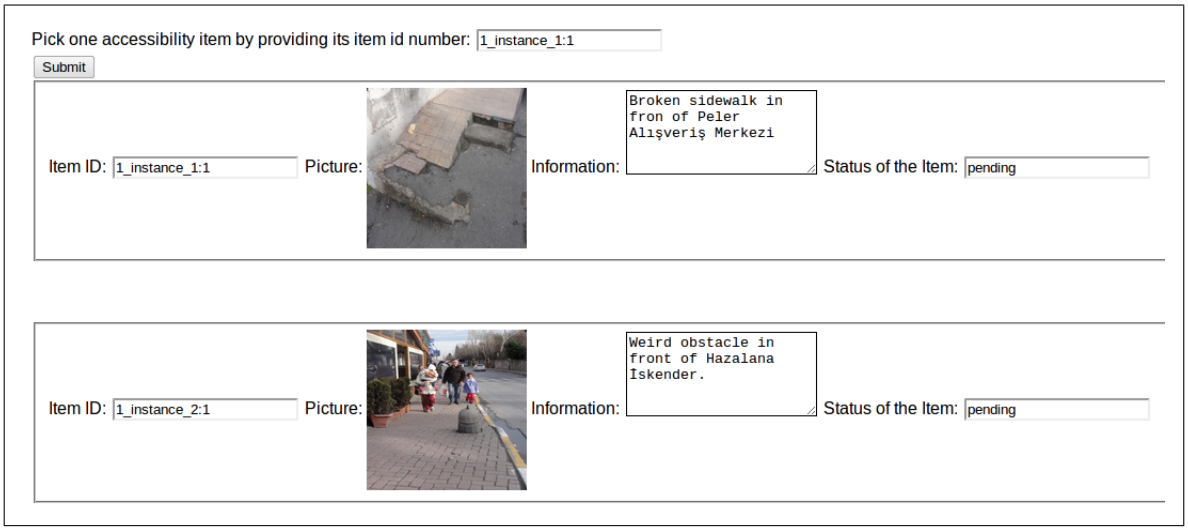


Figure 6.11. Rumelihisarustu – Show Items.

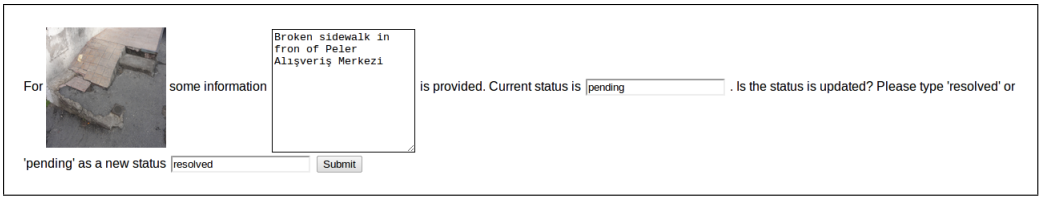


Figure 6.12. Rumelihisarustu – Update Item.

Notify User. Automated tasks are predefined tasks.

Figure 6.21 depicts initilization of *Do Sum Operation* task. Two generated numbers are 88 and 63 which are computed by *Generate Numbers* task. Jack provides an answer, 45, in Figure 6.22, same sum operation is also done automatically by the computer using *Compute Sum* task. And finally, Jack is notified about the correctness of his answer in Figure 6.23 and sees that his answer is not correct.

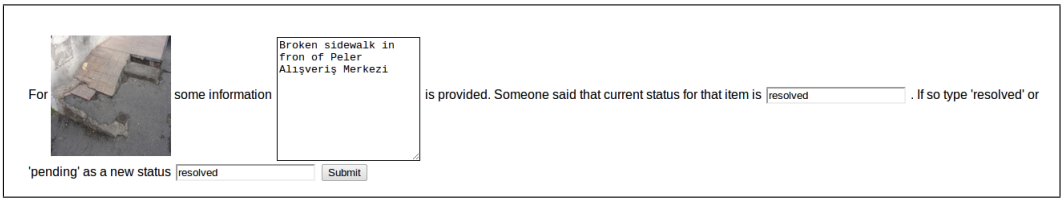


Figure 6.13. Rumelihisarustu – Control Item.

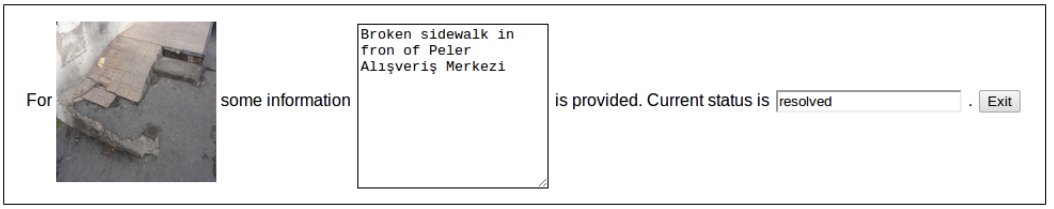


Figure 6.14. Rumelihisarustu – Show Item.

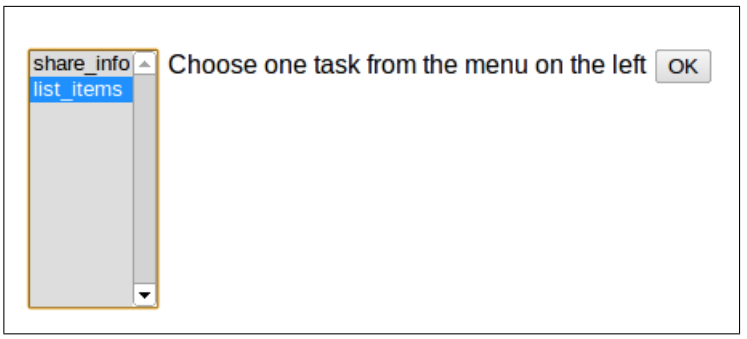


Figure 6.15. Rumelihisarustu – List Items.

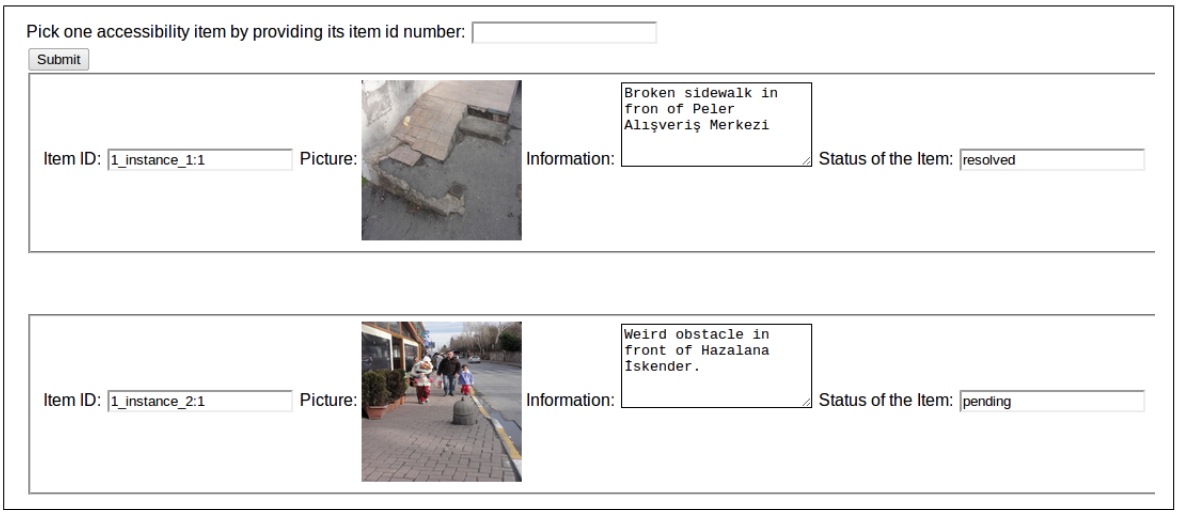


Figure 6.16. Rumelihisarustu – Show Items (after control check).

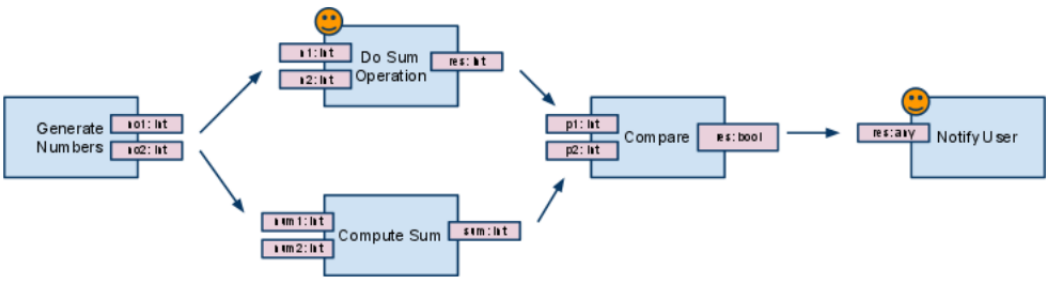


Figure 6.17. Jack – summing integers.

```
def generate_numbers(*args):
    import random
    a = random.randint(0,100)
    b = random.randint(0,100)
    return a,b
```

Figure 6.18. Code for Generate Numbers Task.

```
def compute_sum(*args):
    args = args[0]
    a= int(args[0])
    b= int(args[1])

    return a + b,
```

Figure 6.19. Code for Compute Sum Task.

```
def compare(*args):
    args = args[0]
    param_1 = int(args[0])
    param_2 = int(args[1])

    res = 'false'
    if param_1 == param_2:
        res = 'true'

    return res,
```

Figure 6.20. Code for Compare Task.

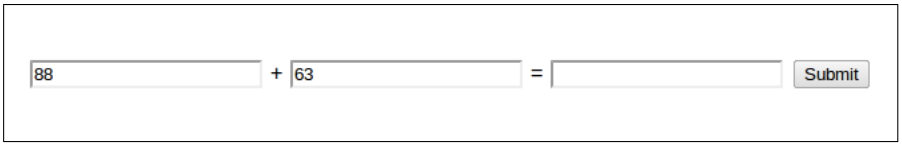


Figure 6.21. Jack – Do Sum Operation.

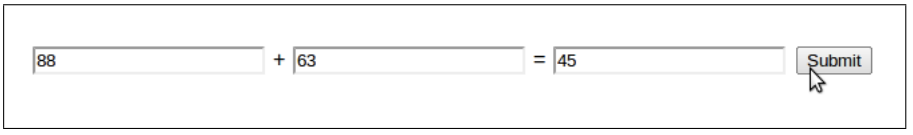


Figure 6.22. Jack provides an answer.

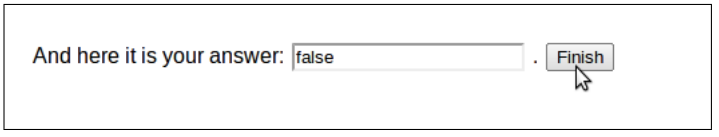


Figure 6.23. Jack – Notify User Task.

7. DISCUSSION AND FUTURE WORK

We are planning to extend this work in several directions.

First, we intend to develop a visual language preserving WeFlow specification semantics with a user friendly graphical user interface. An easy to use graphical user interface will be implemented and it will be highly customizable according to various user needs. As mobility is part of daily life, it is also important to provide WeFlow in mobile environments. Thus, we feel a need to develop a genuine mobile version of WeFlow, we believe that this approach would further increase the Web accessibility.

Second, we are planning to integrate Pantoto's robust data model which supports primitive and user defined data types. Pantoto manages structured data, therefore we intend to add Resource Description Framework (RDF) [53] data export functionality for tasks and/or workflows in order to enable data mobility. RDF export will also allow us to interchange data on Web, enable data import to be used by WeFlow tasks and data export to Semantic Web, Linked Data resources [54, 55]. In this context, WeFlow structured data will be shared across different applications and external structured data, coming from Semantic Web resources e.g. Linked Data, will be used by WeFlow framework.

Third, we intend to create a library of predefined tasks and workflows which can directly be used by specifiers, this approach will enable reusability of common tasks/workflows. WeFlow will offer a rich editor with access to useful, searchable libraries to define workflows. Rather then duplicating workflows and/or tasks by specifying them from scratch, providing a library of highly used workflows and/or tasks is important.

Fourth, we think we can further increase WeFlow efficiency by using model checking techniques to verify if a human computation web application is specified correctly and works exactly as specified.

Another interesting work is we want to bring together our work with Alipi Project. In Alipi project, the idea is enabling Web access for the print-impaired or for people who cannot read. In countries like India, majority of people are uncomfortable with text, either because they are not literate or because they are literate only in their localized language. In this work, people can contribute re-narratives on their sites which are re-rendered based on user profile and available alternative narratives out there on the Web [56–58]. WeFlow-Alipi project will provide a collaborative environment to create renarrations of web sites on Web.

This work opens the gate to several interesting questions such as how to enable perception of collaboration via others in a workflow, how a specification evolves through time and what happens to currently executing workflow instances using this specification, how to define and manage cancellation sets, how to introduce human computation aspects such identity, privacy, how to detect if a human abandons a computation, how to handle time issues, what are social aspects such participation, inclusion.

8. CONCLUSION

End-user programming is part of active research and will be a more important topic in few years as human-computer interaction gains more attention through years. In a recent talk of Clay Shirky [59], *Cognitive Surplus* idea is discussed where Shirky proposes to use the free time of the world's educated ordinary people as an aggregate and points out that "creating something personal, even of moderate quality, has a different kind of appeal than consuming something made by others, even of high quality". Community members, previously happy to spend most of their time consuming, would start making things. WeFlow is the tool empowering non IT savvy people to produce, rather than consuming something made by others, their own human computation web applications fitting to their community needs. As IT savvy people are not domain experts of specific fields, it is really difficult for them to create collaborative environments suitable and highly customizable for communities.

A specification language for end users to define human computation applications has been developed. Furthermore, an application generator that takes a human computation application specification and produces a web based application has been developed. The resulting web application can be used by participants via a web browser. The execution engine executes the participatory web application by distributing the tasks to participants as defined by the specification.

Thus far, we have been able to realize the basic aspects of our designs. Our prototype encourages us with respect to the development of a full fledged visual language for end user specifiers. Combined with a richer data specification we envision a productive application specification environment.

APPENDIX A: WEFLOW SPECIFICATION FOR COSTORY

```
<WeFlow>
    <name>CoStory</name>
    <description>
        Ready to create collaborative stories? Start using CoStory now!
    </description>
    <start_task id="create_story"/>
    <final_tasks> <final_task id="show_story"/> </final_tasks>
    <tasks>
        <task id="create_story" performer="human" type="basic">
            <name>Create Story</name>
            <description>
                Your story title is: \$title  And start your story: \$story
            </description>
            <params>
                <param type="input" name="title" datatype="text"/>
                <param type="input" name="story" datatype="textarea"/>
                <param type="output" name="new_story" datatype="textarea"/>
                <param type="output" name="s_title" datatype="text"/>
            </params>
        </task>
```

Figure A.1. Workflow Information for for CoStory.

```
<task id="rep_update_story" performer="auto" type="repetition">
<name>repeat</name>
<description>repeat</description>
<params>
  <param type="input" name="story" datatype="textarea"/>
  <param type="input" name="is_finished" datatype="yes/no"/>
  <param type="output" name="result_story" datatype="textarea"/>
</params>
<expr>
<condition operand1="\$is_finished" operator="!=" operand2="'yes'"/>
<then taskid="update_story"/>
<else/>
</expr>
</task>

<task id="update_story" performer="human" type="basic">
<name>Update Story</name>
<description>
    This is how the story is \$story Add your words now! \$entry
    Do you think that the story is finished? \$is_done
</description>
<params>
  <param type="input" name="story" datatype="textarea"/>
  <param type="input" name="entry" datatype="textarea"/>
  <param type="input" name="is_done" datatype="yes/no"/>
  <param type="output" name="result_story" datatype="textarea"/>
  <param type="output" name="is_finished" datatype="yes/no"/>
</params>
</task>
```

Figure A.2. Task Information for CoStory.

```
<task id="show_story" performer="human" type="basic">
    <name>Show Story</name>
    <description>
        Story title: \$story_title Your Story: \$story
    </description>
    <params>
<param type="input" name="story_title" datatype="text"/>
        <param type="input" name="story" datatype="textarea"/>
    </params>
</task>
</tasks>
<resourcing>
<task id="create_story">
    <groups>
        <group name="teacher_group"</group>
    </groups>
</task>
<task id="update_story">
    <groups>
        <group name="student_group"</group>
    </groups>
</task>
<task id="show_story">
    <groups>
        <group name="anyone"</group>
    </groups>
</task>
</resourcing>
```

Figure A.3. Resourcing Information for CoStory.

```
<controlFlow>
<task id="create_story"/>
<task id="rep_update_story"/>
<task id="show_story"/>
</controlFlow>
<dataFlow>
<outputMappings>
<mapping taskID="rep_update_story">
<outputParam name="result_story"
expression=":= \$update_story.result_story"/>
</mapping>
<mapping taskID="update_story">
<outputParam name="result_story"
expression=":= \$update_story.story + \$update_story.entry"/>
<outputParam name="is_finished" expression=":= \$update_story.is_done"/>
</mapping>
<mapping taskID="create_story">
<outputParam name="new_story" expression=":= \$create_story.story"/>
<outputParam name="s_title" expression=":= \$create_story.title"/>
</mapping>
</outputMappings>
<inputMappings>
<mapping taskID="update_story">
<inputParam name="story" expression=":= \$rep_update_story.story"/>
</mapping> <mapping taskID="rep_update_story">
<inputParam name="story" expression=":= \$create_story.new_story"/>
<inputParam name="story" expression=":= \$update_story.result_story"/>
<inputParam name="is_finished" expression=":= \$update_story.is_finished"/>
</mapping> <mapping taskID="show_story">
<inputParam name="story_title" expression=":= \$create_story.s_title"/>
<inputParam name="story" expression=":= \$update_story.result_story"/>
</mapping> </inputMappings>    </dataFlow></WeFlow>
```

Figure A.4. Mappings Information for CoStory.

REFERENCES

1. Rheingold, H., *The Virtual Community: Homesteading on the Electronic Frontier*, The MIT Press, Cambridge, MA, 2000.

2. Nardi, B. A., D. J. Schiano, M. Gumbrecht and L. Swartz, "Why we blog", *Communications of the ACM*, Vol. 47, pp. 41–46, 2004.

3. Leuf, B. and W. Cunningham, *The Wiki way: quick collaboration on the Web*, Addison-Wesley, London, 2001.

4. Java, A., X. Song, T. Finin and B. Tseng, "Why we twitter: understanding microblogging usage and communities", *WebKDD/SNA-KDD '07: Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis*, pp. 56–65, ACM, New York, NY, USA, 2007.

5. Zhao, D. and M. B. Rosson, "How and why people Twitter: the role that microblogging plays in informal communication at work", *GROUP '09: Proceedings of the ACM 2009 international conference on Supporting group work*, pp. 243–252, ACM, New York, NY, USA, 2009.

6. Costabile, M. F., D. Fogli, P. Mussio and A. Piccinno, "Software Environments for End-User Development and Tailoring", *PsychNology*, Vol. 2, pp. 99–122, 2004.

7. Wikipedia, *Wikipedia, the free encyclopedia*, 2011, `http://en.wikipedia.org/wiki/Main_Page`, accessed at November 2011.

8. von Ahn, L., "Human computation", *K-CAP '07: Proceedings of the 4th international conference on Knowledge capture*, pp. 5–6, ACM, New York, NY, USA, 2007.

9. Nardi, B. A., *A small matter of programming: perspectives on end user computing*,

MIT Press, Cambridge, MA, USA, 1993.

10. Myers, B., "Studying Development and Debugging to Help Create a Better Programming Environment", *CHI 2003 Workshop on Perspectives in End User Development*, pp. 65–68, 2003.

11. Prahofer, H., D. Hurnaus and H. Mossenbock, "Building End-User Programming Systems Based on a Domain-Specific Language", *6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06)*, 2006.

12. Spivack, N., *Radar Networks Blog*, 2007, `http://www.radarnetworks.com/`, accessed at November 2011.

13. Guess, A., *Imagining Web 4.0*, 2011, `http://semanticweb.com/imagining-web-4-0_b18176`, accessed at November 2011.

14. Facebook, *social utility that connects people with friends and others*, 2011, `http://www.facebook.com`, accessed at November 2011.

15. Facebook, *Statistics — Facebook*, `http://www.facebook.com/press/info.php?statistics`, accessed at November 2011.

16. Völkel, M., M. Krötzsch, D. Vrandecic, H. Haller and R. Studer, *Semantic Wikipedia*, New York, NY, USA, May 2006, `http://www.aifb.uni-karlsruhe.de/WBS/hha/papers/SemanticWikipedia.pdf`, accessed at November 2011.

17. Surgeons, D., *Facebook vs Twitter Infographic – A Breakdown of 2010 Social Demographics*, 2011, `http://www.digitalsurgeons.com/facebook-vs-twitter-infographic/`, accessed at November 2011.

18. Wellman, B., *Networks In The Global Village: Life In Contemporary Communities*, Westview Press, 1998.

19. Wikipedia, *Human-based computation — Wikipedia, The Free Encyclopedia*,

2011, http://en.wikipedia.org/wiki/Human-based_computation/, accessed at November 2011.

20. Chen, D. L.-J., *An Overview to Human Computation*, 2009, http://www.imi.ncku.edu.tw/images/seminar/091002.pdf, accessed at November 2011.

21. Malone, T. W., R. Laubacher and C. N. Dellarocas, *Harnessing Crowds: Mapping the Genome of Collective Intelligence*, 2009, http://cci.mit.edu/publications/CCIwp2009-01.pdf, accessed at November 2011.

22. Quinn, A. J. and B. B. Bederson, *A Taxonomy of Distributed Human Computation*, 2009, http://hcil.cs.umd.edu/trs/2009-23/2009-23.pdf, accessed at November 2011.

23. Wikipedia, *Optical character recognition - Wikipedia, The Free Encyclopedia*, 2011, http://en.wikipedia.org/wiki/Optical_character_recognition, accessed at December 2011.

24. Allen, R., *Workflow: An Introduction Rob Allen, Open Image Systems Inc., United Kingdom Chair, WfMC External Relations Committee*, 1998, http://www.futstrat.com/books/downloads/Workflow-An_Introduction.pdf, accessed at November 2011.

25. *Workflow Management Coalition*, 2011, http://www.wfmc.org/, accessed at November 2011.

26. Coleman, J. S., "Social Capital in the Creation of Human Capital", *American Journal of Sociology*, Vol. 94, pp. S95–S120, 1988.

27. Falk, I. and U. of Tasmania., *Human and social capital [electronic resource] : a case study of conceptual colonisation / Ian Falk*, Centre for Research and Learning in Regional Australia, Launceston, Tas. :, 2001.

28. Falk, I. and S. Kilpatrick, "What is Social Capital? A Study of Interaction in a Rural Community", *Sociologia Ruralis*, Vol. 40, No. 1, pp. 87–110, 2000.

29. Pantoto, *pantoto - mango*, 2011, `http://code.google.com/p/pantoto-mango/`, accessed at November 2011.

30. Uskudarli, S. and T. Dinesh, "Pantoto: A participatory model for community information", *DyD:02: The 2nd International Conference on Open Collaborative Design for Sustainable Innovation*, ThinkCycle and Media Lab Asia In Cooperation with ACM SIGCHI & ICSID, Dec. 2002.

31. Dinesh, T. and S. Uskudarli, "Community Software Applications", A. Venkatesh, T. Gonsalves, A. Monk and K. Buckner (Editors), *IF1P International Federation for Information Processing*, Vol. 241, pp. 103–112, Springer, Boston, 2007.

32. Oasis, *Web Services Business Process Execution Language Version 2.0*, 2011, `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`, accessed at November 2011.

33. IBM, *IBM Business Process Management*, 2011, `http://www-01.ibm.com/software/info/bpm/`, accessed at November 2011.

34. Oracle, *Oracle Business Process Management*, 2011, `http://www.oracle.com/us/technologies/bpm/index.html`, accessed at November 2011.

35. Vanderaalst, W. and A. Terhofstede, "YAWL: yet another workflow language", *Information Systems*, Vol. 30, No. 4, pp. 245–275, Jun. 2005.

36. *YAWL: Yet Another Workflow Language*, 2011, `http://www.yawlfoundation.org/`, accessed at November 2011.

37. Wohed, P., N. Russell, A. H. M. ter Hofstede, B. Andersson and W. M. P. van der Aalst, "Patterns-based evaluation of open source BPM systems: The cases of

jBPM, OpenWFE, and Enhydra Shark", *Inf. Softw. Technol.*, Vol. 51, pp. 1187–1216, August 2009.

38. jBPM, *jBPM Business Process Management Suite*, 2011, `http://www.jboss.org/jbpm`, accessed at November 2011.

39. Ruote, *Ruote Open Source Ruby Workflow Engine*, 2011, `http://ruote.rubyforge.org/`, accessed at November 2011.

40. Shark, E., *Enhydra Shark Open Source Workflow*, 2011, `http://www.together.at/prod/workflow/tws`, accessed at November 2011.

41. Aalst, V. D., "The Application of Petri Nets to Workflow Management", , 1998.

42. Initiative, W. P., *Workflow Patterns*, 2011, `http://www.workflowpatterns.com/`, accessed at November 2011.

43. Masoud Nosrati, R. K., "Occam: A primary parallel programming language", *World Applied Programming*, Vol. 1, No. 1, pp. 85–88, April 2011.

44. Ben-Ari, M., *Principles of concurrent and distributed programming*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

45. Hyde, D. C., "Introduction to the Programming Language Occam", , 1995, `http://www.eg.bucknell.edu/~cs366/occam.pdf`, accessed at November 2011.

46. Stefansen, C., "SMAWL: A small workflow language based on CCS", *Harvard University*, 2005, accessed at November 2011.

47. Choppella, V., B. Vamsikrishna, T. Dinesh and N. Kokciyan, "Towards a Declarative Workflow Model for Customizing Group Processes", *The 7th International Conference on Distributed Computing and Internet Technologies(ICDCIT 2011)*, 2011.

48. Dinmore, M. D. and C. C. Boylls, "Empirically-Observed End-User Programming Behaviors in Yahoo! Pipes", , 2010.

49. *W3C: HTML5*, 2011, `http://www.w3.org/TR/html5/`, accessed at November 2011.

50. Foundation, Z., *ZODB - a native object database for Python*, 2011, `http://www.zodb.org/`, accessed at November 2011.

51. Swartz, A., *Web Framework for Python*, 2011, `http://webpy.org`, accessed at November 2011.

52. W3C, *Extensible Markup Language (XML)*, 2011, `http://www.w3.org/XML/`, accessed at November 2011.

53. W3C, *Resource Description Framework*, 2011, `http://www.w3.org/RDF/`, accessed at November 2011.

54. W3C, *W3C Semantic Web Activity*, 2011, `http://www.w3.org/2001/sw/`, accessed at November 2011.

55. Community, L. D., *Linked Data - Connect Distributed Data across the Web*, 2011, `http://linkeddata.org/`, accessed at November 2011.

56. Janastu, *a11ypi*, 2011, `http://a11y.in/a11ypi/idea/`, accessed at November 2011.

57. Janastu, *a11ypi : Accessibility for the Print-impaired*, 2011, `http://a11y.in/`, accessed at November 2011.

58. Dinesh, T., *Alipi Report - Accessibility for the Print Impaired*, 2011, `http://www.scribd.com/doc/63174056/Alipi-Report-accessibility-for-the-print-impaired-Aug-2011`, accessed at November 2011.

59. Shirky, C., *Cognitive Surplus: Creativity and Generosity in A Connected Age*, Penguin Press, New York, 2010.