Sapienza Università di Roma

Dottorato di Ricerca in Ingegneria Informatica

XXII Ciclo – 2011

Incorporating Usability Evaluation in Software Development Environments

Shah Rukh Humayoun

Sapienza Università di Roma

Dottorato di Ricerca in Ingegneria Informatica

XXII Ciclo - 2011

Shah Rukh Humayoun

Incorporating Usability Evaluation in Software Development Environments

Thesis Committee

Prof. Tiziana Catarci (Advisor)
Prof. Giuseppe De Giacomo (Co-Advisor)
Prof. Massimo Mecella

Reviewers

Prof. Maristella Matera
Prof. Giuliana Vitiello

Author's address:
Shah Rukh Humayoun
Dipartimento di Informatica e Sistemistica "Antonio Ruberti"
"Sapienza" Università di Roma
Via Ariosto 25, I-00185 Roma, Italy.
e-mail: humayoun@dis.uniroma1.it
www: http://www.dis.uniroma1.it/∼humayoun/

To my late father Muhammad Yaqoob

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement and Motivation

High-level usability is acknowledged as a significant feature of software products. On the other side, poor usability and inefficient design of the end product are common causes, amongst others, for failed software products [4, 69, 86]. That's why, one of the challenges in software development is to involve end users in the design and development stages so as to observe and analyze their behavior and to collect feedback in effective and efficient manner and then to manage the ensuing development accordingly. One way to achieve this is by applying user-centered design (UCD) [25, 43, 110] philosophy. This philosophy puts the end users of the system at the centre of the design and evaluation activities, through a number of methods and techniques. The UCD philosophy is applied in software projects with the aims of increasing product usability, reducing the risks of failure, decreasing long-term costs, and increasing overall quality. The International Organization for Standardization (ISO) has also defined the standard guidelines to deal with different aspects of human-computer interaction (HCI) and UCD; in particular, ISO/DIS 13407 [116] provides the guidance on user-oriented design process. Other relevant ISO standard guidance are ISO 9241-11 [115] and ISO TR 16982 [117].

The agile approach [1, 2, 31] is one software development approach that has emerged over the last decade. This approach is used for constructing software products in an iterative and incremental manner; in which each iteration produces working artifacts that are valuable to the customers and to the project. This is performed in a highly-collaborative fashion to produce quality products that meet the requirements in a cost-effective and timely manner.

Generally, in software development practice, software teams hesitate to imply UCD activities due to their time-consuming and effort-intense nature. Using the agile approach, in which customers and product owners lead the

prioritization of the development, helps developers overcome these hesitations. By emphasizing the benefits common to both the end users and the developers, UCD and the agile approach can be dynamically integrated to get benefits from both, resulting in the development of high-quality and usable software products.

We identified that agile development teams were often lacking a properly-integrated approach that utilizes the UCD philosophy from end-to-end at all levels. To overcome this gap, firstly, this thesis proposes a three-fold integration framework that incorporates UCD philosophy into agile software development at three levels: the process life-cycle level, the iteration level, and the development-environment level. This proposed framework identifies ways to apply appropriate UCD activities alongside agile development activities, with the aim of developing high-quality and usable products.

Integrating UCD activities into software development activities fuses the user experience with the development process, attaining a high level of usability in the resulting product. One of the challenges of this integration is the management and automation of the usability evaluation along the development process. Usability evaluation aims at involving users, especially product end-users, in the evaluation process of a specific product to find usability flaws and errors and refine the product according to the feedback. Usability evaluation is performed using existing rigorous approaches and techniques that enable the process of defining and running experiments, collecting and analyzing results, and making decisions regarding which feedback to adopt and to what extent [25]. Unfortunately, in many cases these usability evaluation techniques are performed manually [63], and due to budget and schedule concerns sometimes they are neglected or poorly defined. Automating evaluation methods and techniques, and their application throughout the development process, provides several benefits, e.g.; reduced development costs and time, improved error tracing, better feedback, and increased coverage of evaluated features [63].

To handle this challenge, the thesis proposes a way to define and automate usability evaluation from within the integrated development environment (IDE). The motivation behind this approach is clear. Defining evaluation experiments and running them from within the IDE equips the software development team with the mechanisms to monitor and control a continuous evaluation process tightly coupled with the development process, thus receiving on-going user feedback while continuing development. This also enables them to automatically collect and analyze users and system behavior to recognize usability flaws and errors in an efficient and effective way.

> *It is important to note that we use the term usability evaluation for the evaluation of both product usability and functionality. We use experiments to find usability issues and system problems, and serve as a kind of acceptance test for the developed features.*

The thesis also carries in-depth focus on formalization of modeling user and system tasks and their behavior thus providing a mean of automatic analysis of user and system recorded data during evaluation experiments. This is achieved through structuring system activities, forming their relationship, and defining how users can achieve the desired goals through performing these set of activities. For this purpose, an expressive, dynamic, and well-defined (syntactically and semantically) formal task modeling language is required that gives us not only the facility to model user and system behavior appropriately but also provides the way to construct task models with properties; such as, precondition axioms for actions, postcondition effects on system states, and inclusion of any domain knowledge; that we require for the automated analysis of the recorded data. We achieve this by providing the definition of a formal task modeling language TaMoGolog. This language was created on the foundations of the Golog-family [18, 19, 20, 73, 107] of high-level programming languages. Conducting usability evaluation at the IDE level through TaMoGolog-based task models enables us to record user and system activities and behavior as per the defined mode, to analyze automatically the results by comparing the task models and the recorded data, and to draw conclusions to derive relevant design and development tasks for further improvement in developing product.

1.2 Research Contributions

The thesis comes up with contributions towards three directions for achieving the final goal, i.e., incorporating usability evaluation in software development processes and environments. These three directions are: (i) towards integrating UCD activities in software development process; (ii) towards managing and automating UCD activities at the IDE level related to usability evaluation methods in general, and particularly, task model-based usability evaluation approach for automatic analysis of the users and system data and behavior to highlight usability issues; (iii) and towards defining a formal way for task modeling to be used in task model-based usability evaluation. In this thesis, we discuss thoroughly our contributions in these three directions. Following we summarize these research contributions one-by-one.

Firstly, the thesis provides a three-fold integration framework to incorporate UCD philosophy into agile software development. This framework provides properly-integrated approach that utilizes the UCD philosophy from end-to-end at all levels. The framework integration approach works at three levels:

at the process life-cycle level for the selection and application of appropriate UCD methods and techniques in the right places at the right time; at the iteration level for integrating UCD concepts, roles, and activities during each agile development iteration planning; and at the development-environment level for managing and automating the sets of UCD activities through automated tools support.

Secondly, the thesis provides a generic conceptual framework for constructing task models at different abstract levels. This framework provides definition of a set of special concepts, such as task-types, view-type; to model user and system activities and behavior at different abstractions. These created task models are later used for conducting usability evaluation of the targeted application for highlighting usability issues from several abstractions.

Thirdly, the thesis provides the definition of a well-defined (syntactically and semantically) formal task modeling language, called **TaMoGolog** (**Ta**sk **Mo**deling **Golog**), on the foundations of Golog-family of high-level programming languages. This definition provides sets of predicates using situation calculus to define the domain theory of writing task models. TaMoGolog uses sets of constructs from Golog-family in addition to few its own defined constructs for structuring complex system behavior. Moreover, the thesis defines a framework for the realization of external nondeterministic constructs, based on the approach provided by GameGolog [20], a recent extension to Golog-family. TaMoGolog uses GameGolog semantics at higher-level but differs slightly when defining framework theory using situation calculus at lower level. The thesis also provides the formal semantics of its own defined external nondeterministic constructs using *transition semantics* [18, 102] approach. The support of making nondeterministic decisions by external entities (external application/systems or human users) is critical from task modeling and usability evaluation perspective, as it provides a way to explicitly model external entities' participation during tasks execution. This helps in analyzing users, external application/system, and the evaluated system activities and behavior separately during evaluation experiments.

Fourthly, the thesis provides a conceptual process for defining evaluation experiments and running these from within the IDE to equip the software team with the mechanisms to monitor and control a continuous evaluation process, tightly coupled with the development process. This consists of: defining experiment entity meaning adding a new kind of an object in the development area of software projects for creating and providing evaluation data; deriving development tasks meaning defining new development tasks for forthcoming iterations based on the analysis of the evaluation experiments results against the targeted usability criteria; keeping code traceability meaning automating the process of backward and forward traceability among different evolving parts; and developing evaluation aspects meaning to add automatic evalua-

tion hooks to the software under development for providing measures insights about the users' behavior.

Fifthly, the thesis provides a framework that defines an end-to-end life-cycle to manage and automate task model-based usability evaluation at the IDE level. For this, the thesis explains the role and effects of tagging tasks and variables at the code level, the reflection of usability scenarios in evaluation experiments through TaMoGolog-based task models, the role of TaMoGolog-based task models in recording users and system activities and behavior during execution of evaluation experiments and in performing automatic analysis of the recorded data.

Lastly, the thesis provides the realization of the development-environment level integration of our three-fold integration framework through two automated tools that support usability evaluation at the IDE level. The first tool UEMan, an eclipse plug-in, manages and automates UCD activities alongside the process of development in the development environment. The main capabilities include, creating the experiment object as part of the software project; deriving development tasks from the analysis of evaluation data; and tracing these tasks to and from the code. Further, it also provides a library to enable development of Java aspects for the creation of automatic measures to increase the breadth of the evaluation data. The second tool, called TaMUlator, works at the IDE level to provide the realization of end-to-end task model-based usability evaluation life-cycle by providing a set of APIs and interfaces. The thesis also presents case studies in which development teams used UEMan and TaMUlator to evaluate software projects they developed.

> *The concept of serving of evaluation experiments as the evaluation of both the product usability and functionality and as a kind of acceptance tests for the developed features is the novelty of our approach towards usability evaluation, whereas the previous ones considered only the product usability. The automation and management of UCD activities and usability evaluation at the IDE level to integrate it fully with the development process, the usage of a formal high-level language for task model-based usability evaluation, modeling user and system tasks at multiple abstractions, and the approach of tagging tasks and variables at the program code level for using them during evaluation experiments for recoding user and system data are the novel concepts provided by this thesis. Moreover, the usage of precondition axioms (that include all kinds of constraints to be true before executing a task) and post-condition effects to fluents (which explain how the fluents change their values in result of executing a task) for recoding user and system data during evaluation experiments and the automated analysis based on*

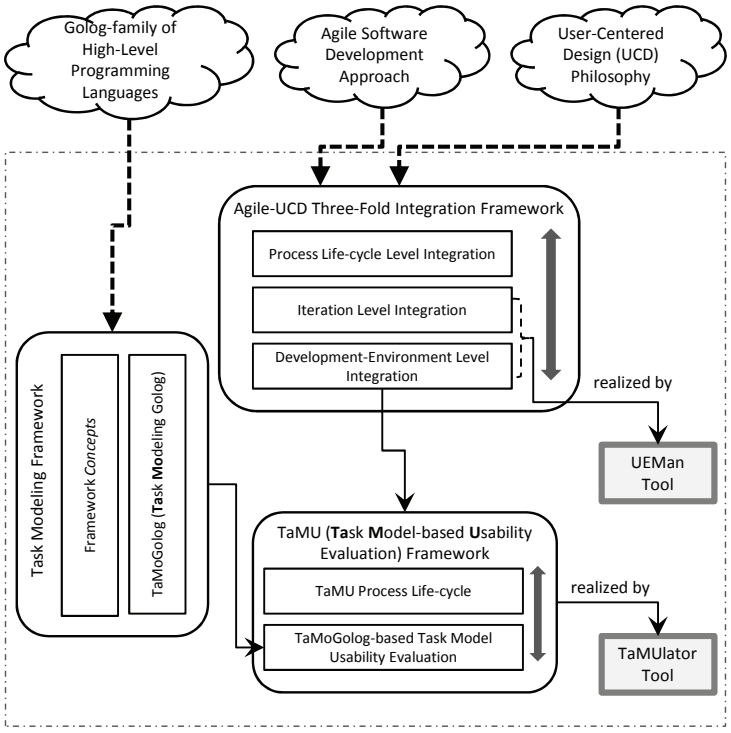


Figure 1.1: Thesis research contributions

this is also one of the key difference between our approach and the previous ones.

Figure 1.1 shows the thesis research contributions. The clouds represent state-of-the-art methodologies that we used for our frameworks, while inside the dashed-line box are the contributions by this thesis. The arrows show which contribution leads to the other one. The contributions listed above and basic ideas from which this work has started are partially contained in the following papers [26, 29, 30, 56, 57, 58, 59, 60]:

- **Y. Dubinsky, S. R. Humayoun, T. Catarci, S. kimani [30]**
  Integrating user evaluation into software development environments
  In Proceedings of the 2nd DELOS Conference on Digital Libraries, Pisa, Italy, December 5-7, 2007.

- **Y. Dubinsky, S. R. Humayoun, T. Catarci, S. kimani [26]**
  Managing User-Centered Design in Agile Projects
  In the workshop on "Optimizing Agile User-Centered Desig"during the 26th ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'2008), Florence, Italy, April 05, 2008.

- **Y. Dubinsky, S. R. Humayoun, T. Catarci [29]**
  Eclipse Plug-in to Manage User Centered Design
  In Proceedings of the 1st International Workshop on the Interplay between Usability Evaluation and Software Development (I-USED 2008), In conjunction with the 2nd Conference on Human-Centred Software Engineering (HCSE 2008), Pisa (Italy), September 24, 2008.

- **S. R. Humayoun, Y. Dubinsky, T. Catarci [58]**
  UEMan: A Tool to Manage User Evaluation in Development Environments
  In Proceedings of the IEEE 31st International Conference on Software Engineering (ICSE'2009), p. 551-554, Vancouver, Canada, May 16-24, 2009.

- **S. R. Humayoun, Y. Dubinsky, T. Catarci [59]**
  A Three-Fold Integration Framework to Incorporate User-Centered Design into Agile Software Development
  Lecture Notes in Computer Science, 2011, LNCS Volume 6776, M. Kurosu (Ed.): Human Centered Design, HCII 2011, p. 55-64, 2011.

- **S. R. Humayoun, Y. Dubinsky, E. Nazarov, A. Israel, T. Catarci [60]**
  TaMUlator: A Tool to Manage Task Model-based Usability Evaluation in Development Environments
  In Proceedings of IADIS Conference on Interfaces and Human Computer Interaction 2011 (IHCI'2011), July 20-26, Rome, Italy, 2011.

- **S. R. Humayoun, T. Catarci, Y. Dubinsky [56]**
  A Dynamic Framework for Multi-View Task Modeling
  In Proceedings of the 9th ACM SIGCHI Italian Chapter International Conference on Computer-Human Interaction: Facing Complexity (CHItaly 2011), Patrizia Marti, Alessandro Soro, Luciano Gamberini, and Sebastiano Bagnara (Eds.). ACM, New York, NY, USA, 185-190, 2011.

- **S. R. Humayoun, T. Catarci, Y. Dubinsky, E. Nazarov, A. Israel [57]**
  Using a High Level Formal Language for Task Model-Based Usability Evaluation
  M. De Marco, D. Teeni, V. Albano, S. Za (Ed.): "Information Systems: Crossroads for Organization, Management, Accounting and Engineering", Physica-Verlag Heidelberg - Springer, ISBN: 978-3-7908-2788-0, 2011.

Moreover, the role of UCD philosophy in the three-fold integration framework is partially influenced by the results of application of UCD activities in a project [54, 55]. The UEMan and TaMUlator tools and their evaluation case studies are author's collaborative work with *Dr. Yael Dubinsky*, who taught one year course "Annual Project in Software Engineering" in Computer Science Department at Technion, IIT from 2008 to 2010.

1.3 Thesis Organization

The outline of the thesis is follow:

In Chapter 2, we provide the preliminary background of the related areas. First, we provide an overview of agile software development approach. We then go through UCD philosophy and different UCD activities that are performed for carrying UCD in software development. We then focus on conducting usability evaluation in software development. Finally, we shed lights on few task analysis techniques and task modeling languages to give an overview of the task analysis area.

In Chapter 3, first we introduce our three-fold integration framework for incorporating UCD philosophy into agile software development. We survey each level one by one for describing them in depth. Then we present UEMan tool that manages and automates UCD activities at the IDE level along the development process. We also present two evaluation studies in which software development teams used UEMan for the software projects they developed. Then we describe other approaches that also integrate UCD and usability evaluation into software development processes. In the end, we conclude and provide directions for enabling task model-based usability evaluation at the IDE level.

In Chapter 4, we provide the theoretical foundation that we require for our task model-based usability evaluation framework. First, this chapter gives the background to those languages in Golog-family that are related to our work. Then we introduce our general framework for structuring task models following by detailed explanation of framework concepts, i.e., *task*, *task type*, *task model*, *view type*, and *view model*. Then we provide the definition of TaMoGolog task modeling language (the set of constructs, the syntax, and the semantics). Moreover, we provide a framework for defining the formal semantics of TaMoGolog external nondeterministic constructs based on the approach provided by GameGolog [20]. We provide low-level implementation details of the external nondeterministic constructs framework in the Golog-family interpreter platform `P-INDIGOLOG` [105] in Appendix A, Appendix B, and Appendix C.

In Chapter 5, first we introduce TaMU framework that provides our approach towards task model-based usability evaluation at the IDE level. Then we describe the role of TaMoGolog for constructing task models for usability scenarios in evaluation experiments and in recording user and system data during execution of evaluation experiments. We also explain the role of TaMoGolog-based task models in performing automatic analysis of the recorded data to highlight usability issues and to test product functionalities. Then we present TaMUlator tool and its evaluation study. TaMUlator works at the IDE level to provide the realization of our end-to-end task model-based us-

ability evaluation life-cycle. Finally, we describe other automated tools that also implement task model-based usability evaluation, and shed lights on the differences of our approach from the previous ones.

In Chapter 6, we present the conclusions of the thesis providing the outcomes and limitations, and sketch the directions for future works for improving the incorporation of usability evaluation in software environments in order to develop high-quality and usable software products.

Chapter 2

Preliminary Background

This chapter is devoted to provide the preliminary background knowledge about the four areas related to the thesis. These four areas are the agile development, the user-centered design (UCD) philosophy, the usability evaluation approach, and the task analysis. Each section provides the state-of-the-art in each one of these areas. This chapter helps to understand the concepts and the terminologies that are used in forthcoming chapters, where we provide our work. In each forthcoming chapter, we provide more specific related work to our approach.

The remainder of this chapter is structured as follows: In Section 2.1, we provide background of agile software development and brief description of few well-known agile development methods. In Section 2.2, we describe user-centered design philosophy and different activities that are performed for carrying UCD in software development. In Section 2.3, we describe usability evaluation, usability engineering approach for managing usability, usability evaluation process for conducting usability evaluation, and brief description of few usability evaluation methods. Finally, in Section 2.4, we provide few well-known task analysis techniques and task modeling languages.

2.1 Agile Software Development

The agile approach is one software development approach that has emerged over the last decade. This approach is used for constructing software products in an iterative and incremental manner; in which each iteration produces working artifacts that are valuable to the customers (a broad group that includes end users, stakeholders, shareholder, etc.) and to the project. This is performed in a highly-collaborative fashion to produce quality products that meet the requirements in a cost-effective and timely manner.

Agile Manifesto

Agile development broadly came to focus when defined by Agile Alliance in the *Agile Manifesto* [2] in 2001 with the twelve principles and four key values. The four key values are [2]:

> *"**Individuals and interactions** over processes and tools;*
> ***Working software** over comprehensive documentation;*
> ***Customer collaboration** over contract negotiation;*
> ***Responding to change** over following a plan"*

It is noted in the manifesto that while there is a value in the items on the right side, the items on the left side are valued more. The twelve principles that describe agile values in much large detail from Agile Manifesto [2] are:

1. "*Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*
2. *Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*
3. *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.*
4. *Business people and developers must work together daily throughout the project.*
5. *Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.*
6. *The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.*
7. *Working software is the primary measure of progress.*
8. *Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.*
9. *Continuous attention to technical excellence and good design enhances agility.*
10. *Simplicity–the art of maximizing the amount of work not done–is essential.*
11. *The best architectures, requirements, and designs emerge from self organizing teams.*
12. *At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.*"

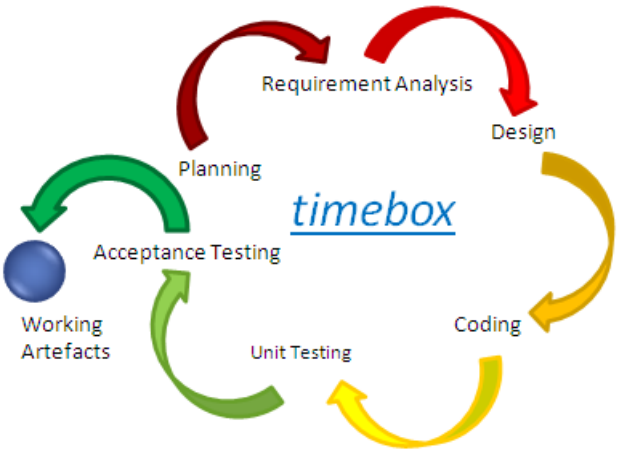


Figure 2.1: Agile iteration life-cycle

Agile Nature of Working

Agile software teams are mostly made up of small self-organizing and cross-functional teams [14], which work by breaking tasks into small increments with minimal planning. Iterations are normally constraint within short time frames, called ***timeboxes***, which typically last from one to four weeks [1]. In each iteration, software team works through a full software development cycle, as shown in Figure 2.1 including planning, requirement analysis, design, coding, unit testing, and an additional acceptance testing for that iteration when a working product is demonstrated to stakeholders/shareholders. It lets the project adapt to changes quickly and the benefits are the reduction of overall risks and time. Although, it is possible that single iteration may not add enough functionalities to make it for the market release, but the goal is to have an available release with minimum bugs at the end of each iteration. Therefore, a product release can take multiple of iterations.

2.1.1 Agile Development Methods

There exist a number of methods, which share the agile manifesto's principles and key values under the umbrella of agile development. Each methodology differs slightly from others, depending on different factors, but all are lightweight with short-term iterations in incremental fashion and emphasize continuous testing and planning with close teamwork. All agile methods focus on rich customers' involvement for establishing, prioritizing, and verifying the requirements throughout the whole development life cycle. Extreme Programming (XP) [7], SCRUM [109], Feature Driven Development (FDD) [89], Adaptive Software Development (ADP) [52], Crystal Clear [13], Dynamic Sys-

tem Development Method (DSDM) [112] are few well-known agile methods. Field surveys[1] in companies, which work with agile development, show the overall improvement in nearly all areas of product development. Following are brief description of few of the notable agile methods.

Extreme Programming (XP): Extreme Programming [7, 31] is one of the well-known and commonly used agile development method. It is based on extreme form of development like daily interaction, working parts of the system, testing, etc. XP emphasizes on groupware style of software development in which small team group (that include developers, managers, customers, etc) collaborates on daily basis. In this approach, the emphasize is laid on communication with customers, simplicity of the design, feedback through testing from the start, and the courage to deliver early the working parts of the system to customers and to makes changes as suggested by them. User stories, a short textual description that tells what the users want from the system, are used a way to plan requirements for the current iteration [7]. Usage of "Class, Responsibilities, and Collaboration"(CRC) cards is a common practice for designing the system by the team. Pair programming, code the unit test first, customer availability, collective code ownership, coding to agreed standards, all code must pass all the unit testing are common practices during the implementation and testing phases.

SCRUM: Scrum [109, 127] is one of the most popular agile methods. There are three main roles in Scrum: the *Scrum Master* that is responsible to maintain the processes, the *Product Owner* that represents the customer group and the business, and the *Team* that is made up of a small group (5-7 people) who performs all the activities in iteration (i.e., analysis, design, implementation, testing, etc). Each iteration lasts from two to four weeks and produces some working increment to the product. The project maintains a *backlog* that contains the high priority requirements to be carried. During the planning phase, the product owner decides which features from *backlog* will go for the current iteration. The team then commits how much they can implement in the current iteration. When it is decided then no one can make changes in the *backlog*, so requirements for the current iteration remains unchanged.

Dynamic System Development Methods (DSDM): The SDM [112] agile methodology is based on Rapid Application Development (RAD) approach. It divides project into three phases: *pre-project phase* that deals and resolves issues like identifying the candidate projects, project funding, project com-

[1]VersionOne, Inc., 1st to 4th Annual Survey: "The State of Agile Development". (2006 - 2009), available at http://www.versionone.net/

mitment, etc; *project life-cycle phase* consists of five stages (feasibility study, business study, functional model iteration, design and build iteration, and implementation) in which the work in the last three stages are done in iterative and incremental fashion; and *post-project phase* for ensuring the proper working of the developed system and deals with maintenance, enhancement, and fixing problems. DSDM provides nine underlying principles: "*user involvement, empowering the project team, frequent delivery, addressing current business need, iterative and incremental development, allow for reversing changes, high-level scope being fixed before project starts, testing throughout the lifecycle, and efficient and effective communication*"[112].

Feature Driven Development (FDD): The FDD agile development methodology [89] combines the agile development approach and the model-driven development approach, and it is suitable for developing critical systems. It is a short iteration in nature with regards to the process and consists of five basic activities. These five activities are: *developing overall model* for providing an overview of the developing system; *building feature list* where features are small pieces of client valued functions, decomposed to a level to be finished in normally two weeks duration; *planning by feature* for producing the development plan based on the created feature list, *designing by feature* for the production of design package for each of the feature and then allocating a set of features to a small team that will be able to finish it in two weeks duration; and *building by feature* for implementing and testing the developed features and then promoting the successful features to the main build.

2.2 The User-Centered Design (UCD) Philosophy

The user-centered design (UCD) philosophy puts the real end users of the system at the centre of designing and evaluation activities such as: by representing or modeling users in certain way through scenarios and personas; through user testing of prototypes; by involving users in making design decisions (e.g. through participatory design). UCD is applied in software projects with the aim of increasing product usability, reducing the risks of failure, decreasing long-term costs, and increasing overall quality.

The term "user-centered design" became widely known when Norman and Draper use it in their book [88]. They focused on the usability of the design and emphasized on the needs and interests of users during the development. Later, Normal developed further the concept in [87] and provided four basic suggestions for designing accordingly. The suggestions provided by Norman were: making it easy to determine the possible actions at a moment, making things more visible (e.g. alternative actions, results of actions), making it

easy to evaluate the current state of the system, and following natural mapping between intensions and the required actions; between actions and the resulting effects, and between the information that is visible and the interpretation of the system state. Norman also suggested seven principles of design that are necessary to facilitate the designer task. These seven principles are: *"use both knowledge in the world and knowledge in the head, simplify the structure of tasks, making things visible, get the mapping right, exploit the power of constraints, design for error, standardize"* [87].

Gould and Lewis [43] provided three principles, which are now accepted as basis for user-centered approach [76], to lead usable and easy to learn computer systems. These principles are later refined and extended by Gould in [41] and Gould et al. in [42]. These four principles are:

- *"Early – continuous – focus on users".* This leads to understanding of the users and their attributes such as; their needs, their behavior, working environment, etc.
- *"Early - and continual - user testing".* This continuously involves end users with the system prototypes (simulated or working) throughout the development and ensures measures are taken to measures, both qualitatively and quantitatively, their performance, behavior, reaction, etc.
- *"Iterative design".* This implements an iterative approach for the user oriented testing of the design, which leads to improvements, and carrying testing again to check the effects of fixed problems in the design accordingly. This provides an iterative loop of "*design, test, measure, and redesign*".
- *"Integrated design".* This leads to the evolution of all usability aspects in parallel, under one focus.

Gulliksen at el. [45] identified a set of definition of twelve principles for designing and developing systems with focus on UCD. These twelve principles, which based on existing theory are: *"user focus, active user involvement, evolutionary system development, simple design representations, prototyping, evaluate use in context, explicit and conscious design activities, a professional attitude, usability champion, holistic design, process customization, a user-centered attitude"*.[45] (p.401-403)

2.2.1 UCD Process Life-Cycle

The International Organization for Standardization (ISO) has defined the standard guidelines to deal with different aspects of HCI and UCD; in particular, ISO/DIS 13407 [116] provides the guidance on user-oriented design

process. It does not cover specific design approaches in detail and only provides how to incorporate UCD activities throughout the design process for achieving usability quality. The standard identified four principles of design process: *"the active involvement of users and a clear understanding of user and task requirements, an appropriate allocation of function between users and technology, the iteration of design solutions, and multi-disciplinary design"* [116]. The results of these principles are four design activities for developing system projects. These four activities are: understanding and specifying the context of use, specifying the user and organizational requirements, producing design solutions, and evaluating design against requirements. Other relevant ISO standard guidance are ISO 9241-11 [115], ISO TR 16982 [117].

Gulliksen at el. [45] also focused on process level and defined "user-centered system design" (UCSD) a process as: *"user-centered system design (UCSD) is a process focusing on usability throughout the entire development process and further throughout the system life-cycle"* (p. 401). They defined a life-cycle that consists of four phases: *analyze* for gathering requirements and observing user needs, *design* for usability through prototyping (both paper and working), *evaluate* for continuous evaluation while taking relevant measures and feedback for making plans for the next iteration. This life cycle phases are helped by two extra phases: *vision* for planning, initial concepts elaboration and *construct* for deploying the tested parts of target application.

Detweiler [23] provided a life-cycle for designing with UCD philosophy through three iterative phases followed by the development phase. Before any UCD phases, the first step is the innovation of the vision of the product. In the first phase *understand user*, emphasize is given to understanding the users and their requirements. As an iterative phase, it is recommended to start from the existing database of users, research their population, environment, needs and to involve them from the beginning. The theme of the second phase *define interaction* is to define the interaction before initiating design phase. The user research, conducted during phase one, is fed towards use cases that define the usage of end product. The completed use cases are also validated with user populations. The last phase in UCD life cycle is *Design UI* in which the user interface is defined directly from the use cases, defined in previous phases. This is performed in two stages: *low-fidelity prototyping* for rapid experimentation and evaluation, and *high-fidelity prototyping* for final and formal behavior preview of the product design. Finally, in the *development validation*, the design is validated with two reviews, one before the development and one after the development. If the design is successful in usability benchmark testing then it is deployed, otherwise it is sent back for improvements and redesigning.

2.2.2 User-Centered Design Activities

There UCD approach provides a variety of activities that are used in different development phases for different purposes. Normally, all of these suggest and recommend those methods that are common to Human-Computer Interaction (HCI) literature. The main idea is to promote informal methods where there is no need for intense training. As each method targets some particular context at a specific phase; so, it is normally recommended to use a mixtures of these to get maximum feedback. Here, we provide brief description of few of the selected methods that are commonly used in UCD practice.

Card Sort: In this method, the items of information are written on individual index cards and users are asked to sort these into different categories according to predefined criteria [110]. Users explain the reasons behind their sorted categories. This method is quick, cheap, and useful in early phases of development, but lacks to reveal the real interface problems. Normally, it is done with a group of 10-20 users.

Contextual Inquiry:[8, 53] This is a structural approach for gathering and interpreting data fieldwork. The designers and the team members visit the actual workplace to observe and analyze users' working in their actual environment and other encapsulating factors. This method is helpful in earlier stages and gives the opportunity to see the real environment. On the other side, it is often time consuming and informal in nature.

Focus Group: Normally group of three to ten users [110] participate with the team and discuss various activities. The participants provide their ideas and opinions about the target system. This method is good for analysis purpose and churns a large amount of data, but it requires a good facilitator and unbiased opinion to achieve accurate results.

Interview: This method is normally used in requirement and analysis phases. There are variations in types and in number of participants. Generally, it is categorized in *unstructured* for informal and open-ended questions, *structured* for predefined questions, and *semi-structured* that uses both open and closed questions [25, 110]. This method involves not only the end users but other stakeholders and shareholders of the system. It is low-cost method and provides an effective mean to identify users' need. On the other hand the results dependent on participants' memory, their opinion and sometimes can be time consuming due to busy schedule and other commitments of the participants.

Paper Prototype Testing: A group of usually 4 to 8 users are asked to evaluate prototype of the system in paper form. They also explain their choices and the possible task sequences for completing a scenario. This method is economical, swift and useful early in the development when there is no existing prototype.. However, due to its informal level, user can make such choices which are difficult to accommodate in the targeting system design.

Survey: In this method, users are asked standard set of questions through some channel such as paper, telephone, email, etc. Through this, it is easy to gather data but the drawbacks are the lack of reliable instrument, self reporting may not properly propagate users' behavior, and there is a possibility that most of the users may not give answer of open ended questions.

Log-File Analysis: The user activities are collected from log-files and are analyzed normally with the help of automated tools, which provide different patterns of users' behavior. This method provides an easy and quick way to gather users' behavior data. However conversely, log files usually lack information about the specific reasons behind user' actions.

Task Analysis: This method is normally used to observe an existing situation [25, 110]. The designers observe users and their work, and also talk with them to identify user tasks and specific goals attached to these tasks. The participating users are normally more than five. This method reveals new information and highlights the way users perform their tasks. But, it is time consuming and needs an expert to observe the process overall and requires expert users in order to give correct answers.

Usability Test: Users are asked to perform different tasks on the target system. The experts and designers observe and record users' activities, behavior, and performance. User can be asked to give any feedback during or after the experiment. Automated tools support is also used to save time and to get accurate data. There are many techniques that are used to check usability level of the targeted system [63]. Each technique highlights a portion of the usability issues from a specific perspective. Normally it is recommended to use more than one technique to highlight maximum usability issues [63]. This method helps to identify numerous real usability problems early in the development, but again, this method can be time consuming however, this can be reduced through automated tools support.

Heuristic Evaluation:[25, 110, 84] The developing/developed system is analyzed against a set of user-oriented heuristic principles, one such criterion is provided by Nielsen [84]. This method is performed by usability and UI ex-

perts. Generally, participation of multiple experts is recommended to identify prevailing problems. This method is inexpensive, quick, and useful to identify different/range of usability problems. The experts with preeminent knowledge of HCI and the working of system are recommended for this exercise.

Walkthroughs: This method involves walking through a task with the system and making note of the usability problems that occur [110]. In most of these techniques, such as Cognitive Walkthrough [110, 84], normally system and UI experts perform these. In few cases, such as Guided Walkthrough, users are also involved with the help of an expert/facilitator. In this case, users are asked questions during and after the walkthrough in order to gauge the user's understanding of the system.

Expert View: In this method, the experts (UI experts, system analysts, etc) view and analyze the prototype or the actual working system and provide feedback based on their own expertise. This method is quick and useful to resolve issues early, but largely depends on experts' own knowledge of the target system and expertise.

2.3 Usability Evaluation

Usability is defined by the International Organization Standardization (ISO) as:

> *"the extent to which the product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use"* [115]

Sharp et al. [110] defined usability as to achievement of six target goals:

> *"effective to use (effectiveness), efficient to use (efficiency), safe to use (safety), having good utility (utility), easy to learn (learnability), and easy to remember how to use (memorability)"*

While Dix, et al. [25] divided principles to support usability into three categories:

- learnability category handles principles (e.g., *"predictability, synthesizability, familiarity, generalizability, consistency"* [25]) that scales the level of learnability ease for the new user to interact with the system to get the maximum performance and output.

- *flexibility* category handles principles(e.g., *"dialog initiative, multi threading, task migratability, substitutivity, customizability"* [25]) that defines the possible ways for exchanging information between the user and the system.

- *robustness* category provides principles (e.g., *"observability, recoverability"* [25]) that assess the support provided to the users of the system for achieving targeted goals.

We can conclude from the above definitions that *usability* is a concept that is measured in a system against a set of predefined goals/principles. The purpose is to understand and conclude the extent to which the system possesses these goals/principles from user perspective. This is checked through some *usability criteria* that enable to assess the level of these goals/principles, e.g. time to complete a task (efficiency), in a particular system [110].

Usability Engineering

Usability engineering is a process of achieving usability goals in a particular system through applying a set of methods and techniques during different development phases. One of the main goals of usability engineering is to improve the user interface of the targeted system [71, 83]. The usability engineering process involves specifying the usability criteria, writing down formally this in usability specification, and then assessing the system against such criteria [25, 110].

Usability metrics are used for measuring quantitative usability aspects of the targeted system [25], such as Whiteside, Bennett, and Holtzblatt [125] provided a list of measurement criteria for assessing the usability metrics. This list includes criteria such as "*time to complete a task, percent of task completed, time spent in errors, number of commands use, number of good and bad features recalled by users, number of times user expresses frustration or satisfaction,.* ". This list can be used to quantitatively determine the level of usability a system provides to its users. Whiteside, Bennett, and Holtzblatt [125] also provided a list to set measurement levels to be used in usability specification that go from level 1 to level 8. ISO standard 9241 [115] also provides a list of usability metrics and categorizes these through their contribution towards three aspects of ISO usability definition, i.e., *effectiveness*, *efficiency*, and *satisfaction*. It is recommended to insert explicitly the usability criteria early in the development so as to judge the final product against these pre-defined criteria [25]. The target usability criteria and metrics' values differ from system to system [71]. Therefore, it is possible that the higher priority usability aspects for a system may become lower priority aspects in another system, for example, learnability and memorability is on higher priority for a

system for disable people while effectiveness, efficiency and safety is on higher priority for a payment system.

Deborah Mayhew [77] proposed a life-cycle for managing usability engineering. It describes how to perform usability tasks and their integration in software development life cycle. The Mayhew's usability life-cycle consists of three phases: *requirement analysis, design/testing/development* and *installation*. The usability goals are identified in the first phase through help of different activities such as user profiling, task analysis, platform capabilities constraints, and general design principles. These goals are then captured in the *style guide* that is used throughout the whole development to ensure these goals are achieved. The second phase deals with design, testing and deployment. This phase is divided into three levels where in the first level the major usability flaws are eliminated through several activities during analysis, in the second level it is ensured that usability goals are met in the developing system during design and implementation, while in the third level, the desired usability goals are ensured through detailed user interface design and its evaluation activities during the testing of the system [110, 77].

Usability Evaluation Process

A usability evaluation process evaluates the targeted system against the defined usability criteria using usability metrics. It checks whether the system, and especially its user interface, possesses all those required usability aspects and the extent of these. It tries to find-out through different approaches and techniques if the product is easy to learn and use, whether it is efficient and effective in achieving users' goals, and if it helps the users to perform their tasks [25, 82, 110]. The process aim is to find usability flaws and errors and refine the product according to the feedback.

Usability evaluation is performed using existing rigorous approaches and techniques that enable the process of defining and running experiments, collecting and analyzing results, and making decisions regarding which feedback to adopt and to what extent [25]. There are three main sources for performing usability evaluation: users, usability experts/designers, and models [25]. The activities in evaluation process can be separated into three phases; collecting usability data, analyzing the data to identify the usability flaws, and suggesting improvements [63]. The goal of usability evaluation varies in different phases of development; as in earlier analysis and design phase the purpose is to look into alternative user interface designs and then to identify and select the preferred one, while in later stages the purpose is to check whether the developed user interface and system meets the usability criteria defined earlier [71].

Usability Evaluation Terms

Following are few terms that are used in usability evaluation process [25, 83, 110].

Usability testing: To evaluates the usability level of a system against a defined usability criteria and to check at what extend the system posses these usability aspects.

User testing: In this evaluation, end users of the system are asked explicitly to perform different tasks using the system (paper prototype or working prototype). Users' performance and satisfaction are observed and recorded while they perform tasks, and then it is analyzed to find out usability issues and to suggest improvements.

Usability laboratory: The laboratory that is designed and used for performing usability experiments.

Controlled experiments: These are also called as experimental evaluation in which experiments are normally performed in laboratory environment and are controlled by evaluator(s). In these experiments, different aspects like performing tasks, time, and laboratory environment are controlled by evaluators [110].

Analytical evaluation: In this evaluation, experts (e.g., UI expert, system analyst, designers) participate in evaluation testing rather than the end users of the system. Walkthrough, heuristic evaluation, feature inspection are few examples.

Predictive evaluation: In this evaluation type, theoretical models are used to predict users' possible performance on a system.

2.3.1 Usability Evaluation Methods

Different usability methods are used in the evaluation process depending on the purpose of the evaluation and the phase of development [25, 80]. Dix at el. [25] distinguished eight factors for selecting appropriate method at right place in the development. These eight factors are: *"the stage in the cycle at which the evaluation is carried out, the style of evaluation, the level of subjectively of the technique, the type of measures provided, the information provided, the immediacy of the response, the level of interface implied, and the resources required"* [25] (p. 357). To discover maximum usability issues, more than

one method is needed, as each method highlights only specific usability flaws and errors. We already have described few of the usability methods (e.g., log-file analysis, heuristic evaluation, walkthrough, expert view) in Section 2.2.2. Following are brief descriptions of few other well-known usability evaluation methods.

Thinking Aloud Protocol: In this method, the users are asked to talk in loud voice about what they are thinking or what they are doing while they perform work on the system. An observer observes it and takes notes. This technique has the advantage of simplicity and does not require much expertise. This is useful in early phases to elaborate the system actual working [25]. A variation of this method is *cooperative evaluation* [79] in which user also see him/her as collaborator and can ask questions to the evaluator if anything is unclear.

Performance Measurement: In this method, the users are asked to perform different tasks on the target system. During this, their activities are observed and recorded through a number of techniques like through paper and pencil, audio recording, video recording, computer logging, or user notebook [25, 63]. Then the user performance is analyzed against some predefined criteria such as given by Whiteside, Bennett, and Holtzblatt [125].

Cognitive Walkthrough [96, 124]: This is a branch of walkthroughs technique and performed by system and UI experts [96, 84, 124]. The main focus is on establishing *learnability* usability aspect that talks how easy the system is able to learn through exploration [25]. There are four things that are needed to perform usability evaluation through cognitive walkthrough: a detailed specification of the system prototype (either working partially or fully), scenarios of the tasks that are supposed to be performed by users, sequences of actions that are supposed to be followed by the users to be performed in certain way for achieving each specific task, and the information about participating end users, their experience, and knowledge level [25, 96]. The evaluator (normally an analyst, UI expert, or designer) observes user while he/she performs actions to achieve each targeted task, and then evaluates user's behavior towards selecting right actions and progressing towards the solution. As the focus turns towards *learnability*, so a critical point to analyze and highlight is what the user knew before executing actions and what the user has learned after executing actions.

Questionnaires: In this method, users are asked to use the system openly (to perform different things) or closely (to perform given tasks) and then a set of questions is given to them to take feedback. This technique has the

advantage of taking feedback from a wider participation group in short time. There are several styles for questions that can be included in the questionnaire [25], such as *general* for taking information about the user data, *open-ended* for taking user own opinion, scalar to ask the user to rank a specific statement numerically (e.g., from scale 1 to scale 5 where scale 1 means disagree and scale 5 means agree), *multi-choice* to ask the user to select one from the listed choices, and *ranked* for ordering a list of choices according to user's own rank.

Model-based evaluation: In this evaluation, cognitive and design models are used that combine design specification and evaluation together [25]. Some model-based methods use task-models or user-models for evaluation [10, 71, 91, 97]. In task model-based usability evaluation technique, task models attached to evaluation provide a mean of user and system tasks. When user perform different tasks on the system, the user's and system activities are recorded and then analyzed against these created task models to highlight different issues such as incomplete tasks, users' performance, selection of a particular path, etc.

Automating Usability Evaluation

Usability evaluation is time consuming and costly. Unfortunately, in many cases the usability evaluation techniques are performed manually, and due to budget and schedule concerns sometimes they are neglected or poorly defined. Moreover, different techniques highlight different issues and it is also possible that the same technique produces notable variability in results when applied to different evaluation studies for the same system [78, 83]. Automating evaluation approaches and techniques, and applying these throughout the development process, provides several benefits, e.g.; reduced development costs and time, improved error tracing, better feedback, and increased coverage of evaluated features [63]. Support of possible automation for different usability approaches varies due to different requirements for each approach. Balbo [6] suggested taxonomy for categorizing automation approaches into four categories: *none* for approaches manually performed by experts of human factors, *capture* for approaches capturing users and system activities through help of automated tools, *analysis* for approaches identifying usability problems automatically, and *critique* for approaches that also provide solution to the identified usability problems. Ivory and Hearst [63] conducted a detailed survey on the state-of-the-art in the automation of usability evaluation techniques at different levels. They analyzed 132 evaluation techniques using the taxonomy suggested by Balbo [6], both for Web and WIMP (Windows, Icons, Pointer, and Mouse) interfaces, and found that only 33% of those techniques are supported by automated tools. The survey concluded that there is a great

under-exploration of usability evaluation methods automation and suggested focusing the research on automation techniques.

2.4 Task Analysis

Task analysis [24] is a way to analyze the system's working and then break it down to low-level activities to determine what users need to know to perform their jobs. The task analysis process emphasizes on three key elements that are the actions people do, the things on which people act on, and the things that people need to know [25]. Normally, task analysis is used for the existing systems and procedures, not for the new ones [5, 24, 25, 110].

A *task model* structures and forms relationship between the low-level activities, recognized in analyzing system's working during task analysis process, and defines how the user can achieve the desired goals through performing these sets of activities. Generally, task model can be categorized into two types [80]: *user task model* that describes the task structure from users' thinking to perform actions for the achievement of the desired goal(s), and *system task model* that describes the task structure from the perspective of system assumptions for performing the actions.

2.4.1 Task Modeling Techniques

Over time, various task analysis approaches have evolved such as Hierarchical Task Analysis (HTA) [5], Goals, Operations, Methods, and Selection rules (GOMS) [11], Groupware Task Analysis (GTA) [122], Task Analysis for Knowledge Description (TAKD) [24], Cognitive Task Analysis (CTA) [17], etc.

Dix et al. [25] distinguishes these approaches into three categories:

- *Task-decomposition-based techniques* that generally split tasks into subtasks and order them according to their supposed execution path.
- *Knowledge-based techniques* that look in to the objects and actions that user need to know to perform a task and the organization of this knowledge.
- *Entity-relation-based techniques* that look in to the actors and objects, their relationships, and the actions performed by them.

Following are brief descriptions of four well-known task analysis approaches.

Hierarchical Task Analysis (HTA): Hierarchical Task Analysis [5] is the most widely known and used technique for the task analysis. It breaks down

tasks in low-level actions and defines their order to achieve some goal. The tasks are divided into sub-tasks till we reach to the low-level basic actions. These low-level actions are then structured according to the user goal where *plans* describe in which order and how to achieve that specific goal [25]. The leaves in HTA task model are concrete low-level actions (tasks), while the parents are abstract tasks that normally manage the structure of sub-tasks. HTA focuses on those tasks and actions that are physical in nature and are observable; therefore, it also keeps those tasks that are not related to the software [110].

Goals, Operations, Methods, and Selection (GOMS): Goals, Operations, Methods, and Selection [11] technique is helpful for modeling procedural knowledge. It models tasks in terms of: a set of goals, a set of operators, a set of methods, and a set of selection rules. In GOMS, *Goals* are the representation of users' goals and describe what the users want to achieve. *Operators* are the representation of low-level basic actions. These must be performed by users when they interact with the system. These are of two kinds: first are those that affect the system state (e.g., by giving some input to system), and second are those that affect only user's mental state (e.g., reading an output) [11, 25]. *Methods* decompose goals into sub-goals and act like procedures for accomplishing specific goals. *Selection* provides the rules that help the user for selecting the appropriate method for achieving the desired goal. GOMS is also useful for predicting the quality of prototypes or existing system [110].

Groupware Task Analysis (GTA): Groupware Task Analysis [122] approach talks about collaborating environments. It provides a rich set of concepts; such as *role, agent, object, task, event, goal*; that are useful while modeling task structure for the systems that collaborate for performing some common goal. The aim is to model the situation in which collaborating tasks are performed [25].

Task Analysis for Knowledge Description (TAKD): Task Analysis for Knowledge Description [24] is a knowledge-based task analysis technique that involves building taxonomies based on objects and actions in the task. The taxonomies act as hierarchical descriptions aimed at understanding the knowledge required to perform a specific task. TAKD uses *Task Descriptive Hierarchy* (TDH)[24], a special form of taxonomy. TDH uses logical OR, AND, and XOR for the branches in the taxonomy. The hierarchy system in TAKD is based on genericity of knowledge where taxonomy is used to link similar tasks [25]. That is why it is more useful for producing the teaching material and the instructional manuals.

2.4.2 Task Modeling Languages

Different notations have been suggested for writing task models such as User Action Notation (UAN) [47] for in textual form and ConcurTaskTrees (CCT) [92] for graphical representation. Following are brief descriptions of three task modeling languages.

ConcurTaskTrees (CCT): ConcurTaskTrees [92, 90] is one of the well-known techniques for writing task models in HCI community. It provides graphical representation for different abstraction of tasks through a hierarchical based task tree, and specifies temporal relationships between tasks and sub-tasks using operators based on LOTOS [118] formal notations. It supports four kinds of tasks abstraction, called *user* tasks, *abstract* tasks, *interaction* tasks, and *application* tasks.
In CCT, a task is defined by a set of attributes. These are *name* to identify unique name, *type* for representing any one of the four task types, *objects* where each object has name, type, input and output object actions, *iterative* to indicate whether the task is iterative in nature, *first action* for representing the possible initial actions in the task, and *last action* for representing the possible last actions in the task [92]. The CCT set of constructs includes *"interleaving, choice, concurrency with information exchange, order independence, deactivation, enabling, enabling with information passing, suspend-resume, iteration, finite iteration, optional task, and recursion"* [92] (p. 812). This representation technique is supported by CCTE (ConcurTaskTrees Environment), a tool for creating task trees and building the relationship between different subtasks in the task-tree according to the semantics of task model [80]. Sinnig et al. [111] enhanced the set of temporal operators of CCT and added four constructs: *"stop, nondeterministic choice, deterministic choice, instance iteration operators"* (p. 45).

User Action Notation (UAN): User Action Notation [47] is a textual task modeling language that provides user- and task-oriented notations for describing the behavior of the user and the interface while both of these perform some tasks together. The primary abstraction in UAN is a *user* task. The basic concepts are *task* representing a target task, *user action* representing either a primitive user action or task, and *action set* containing the union of all user actions defined in the description of the attached task and is obtained by applying a projection function on the attached task [46].
The UAN uses modal logic for describing the temporal relationships between tasks. UAN uses the quasi-hierarchical structure of asynchronous tasks for representing an interface, where the sequences inside a task are independent of sequences inside other tasks. The lowest level describes the user actions, the

possible interface feedbacks, and any information related to changes in state. The UAN set of constructs consists of *sequence* $(A\ B)$, *waiting* $(A(t > n)B)$, *repeating disjunction* $(A \mid B)^*$, *order independence* $(A\ \&\ B)$, *interruptibility* $(A \rightarrow B)$, *one-way interruptibility* $(A \rightarrow B)$, *mutual interleavebility* $(A \leftrightarrow B)$, and *concurrency* $(A + B)$ [46].

Collaborative Task Modeling Language (CTML): CTML [128] is a recent task modeling language for capturing the behavior dynamics of collaborative environments. The collaboration task expression in CTML is a task tree with CTT-like notations. The task in CTML has a unique *identifier* for its name, a *precondition* for adding execution constraints as the task is considered to be in execution mode if the precondition is satisfied, and an *effect* representing the state change due to the execution of the task. CTML also provides a *domain model* for presenting domain specific concepts and to associate these relevant roles in the collaboration. The CTML set of constructs includes *choice, order independence, concurrent, enabling, disabling, suspend/resumes, iteration, instance iteration,* and *optional* [128].

Chapter 3

Integrating User-Centered Design into Agile Software Development

3.1 Motivation

This chapter is devoted to define a conceptual framework for the integration of user-centered design (UCD) philosophy [25, 43, 110] into agile software development approach [1, 2] at different levels. This proposed approach identifies ways to apply appropriate UCD activities alongside agile development activities, with the aim of developing high-quality and usable products.

One of the challenges in software development is to involve end users in the design and development stages so as to collect and analyze their behavior and feedback in an effective and efficient manner and to manage the ensuing development accordingly. One way to achieve this is by applying UCD philosophy. This philosophy puts the end users of the system at the centre of design and evaluation activities, through a number of methods such as involving them in Participatory Design or testing the Working Prototype [25, 110]. The philosophy is applied in software projects with the aims of increasing product usability, by involving the end users in design, development, and evaluation activities; to aim at reducing the risks of failure, decreasing long-term costs, and increasing overall quality.

The agile approach [1, 2, 31] is one software development approach that has emerged over the last decade. This approach is used for constructing software products in an iterative and incremental manner; in which each iteration produces working artifacts that are valuable to the customers and to the project. This is performed in a highly-collaborative fashion to produce quality products that meet the requirements in a cost-effective and timely manner.

Generally, in software development practice, software teams hesitate to imply UCD activities due to their time consuming and effort-intense nature. Using the agile approach, in which customers and product owners lead the prioritization of the development, helps developers overcome these hesitations. By emphasizing the benefits common to both the end users and the developers, UCD and the agile approach can be dynamically integrated to get benefits from both, resulting in the development of high-quality and usable software products. We identified that agile development teams were often lacking a properly-integrated approach that utilizes the UCD philosophy from end-to-end at all levels. To overcome this gap, we propose a three-fold integration framework that gives suggestions and recommendations for involving UCD in agile software development at different levels.

This chapter provides our work towards the following directions:

- A three-fold integration framework to incorporate user-centered design philosophy into agile software development. It provides integration at three levels: (*i*) at the process life-cycle level for the selection and application of appropriate UCD methods and techniques in the right places at the right times; (*ii*) at the iteration level for integrating UCD concepts, roles, and activities during each agile development iteration planning; and (*iii*) at the development-environment level for managing and automating the sets of UCD activities through automated tools support.

- A conceptual process for defining evaluation experiments and running them from within the integrated development environment to equip the software team with the mechanisms to monitor and control a continuous evaluation process, tightly coupled with the development process, thus receiving on-going user feedback while continuing development.

- The realization of development-environment level integration through a plug-in tool, called UEMan (User Evaluation Manager), for Eclipse development platform [32] to manage and automate UCD activities alongside the process of development at the integrated development environment (IDE) level.

The rest of the chapter is organized as follows.

Section 3.2 introduces our three-fold integration framework that offers integration of UCD philosophy in agile software development.

Section 3.3 describes the most abstract level of integration, i.e., the process life-cycle level integration. It provides suggestions and recommendations for the selection and application of appropriate UCD methods and techniques during development.

Section 3.4 provides detail for the second level of integration, i.e. the iteration-level integration. It describes how UCD concepts, roles, and activities can be integrated into agile development iteration activities. It also explains the resulting effects into agile development to support the continuity of UCD activities in the development iteration. These resulting effects are performing iterative design activities, taking measures, and defining UCD roles.

Section 3.5 describes the concrete level of integration, i.e., the development-environment level integration. It explains our understanding of the lack of UCD management practices in software development environments and provides the conceptual framework for the management and automation of UCD activities in the integrated development environment. This level integration resulting effects are the creation of experiments including the use of evaluation aspects, results analysis, and code traceability.

Section 3.6 presents the developed eclipse plug-in tool, called UEMan, and describes how it can be used to managed and automate user evaluation alongside software development. It also presents two evaluation studies of UEMan.

Section 3.7 discusses the related work that has been carried to provide the integrated approaches. It describes other approaches and techniques for integrating user-centered design in agile development.

Section 3.8 provides the concluding remarks and the directions and rationales for the work of forthcoming chapters.

The UEMan tool and the evaluation case studies presented in Section 3.6 are author's collaborative work with *Dr. Yael Dubinsky*, who taught one year course "Annual Project in Software Engineering"in Computer Science Department at Technion, IIT from 2008 to 2010. The UEMan tool was developed by one of the teams in the course session 2007/08 and an initial evaluation study was also done during the course, while the second evaluation study was done during the course session 2008/09.

3.2 The Three-fold Integration Framework

The UCD philosophy provides different activities as discussed in Chapter 2; for example, by representing or modeling users in scenarios and personas, having users test prototypes (either paper or working prototype), and involving users in design decisions (e.g., thorough participatory design); aiming at designing and developing systems with a focus on users' needs [25, 43, 45, 110]. Additionally, the International Organization for Standardization (ISO) has also defined standard guidelines to deal with different aspects of HCI (Human-Computer Interaction) and UCD [116, 115, 117]. One description of UCD that we find particularly motivating is: *"User-centered system design (UCSD) is a process*

focusing on usability throughout the entire development process and further throughout the system life-cycle" [45] (p. 401). One thread common to all methodological approaches come under umbrella of UCD is that the usability evaluation must be carried out throughout the whole development life cycles; it is more beneficial rather than the application of these techniques only during the early or some specific phases of development.

Poor usability and inefficient design of the end-product are common causes, amongst others, for failed software products [4, 69, 86]. Since software products are developed for the users, it's likely they will fail if the users find them difficult to operate, due to the lack of usability and inappropriate design. Normally in development practice, users are either involved at the very beginning of the project when defining the requirements or at the end of the project when testing the developing product. Furthermore, in the testing phase, the project teams focus more on checking the functionality of the product (e.g., performance, reliability, security, robustness, etc.) rather than its usability and design aspects. Checking usability or solving defects at the end of the development process requires more time, effort, and money; hence, they are not usually performed. Generally, in software development practice, and particularly, in iterative and incremental type of software development processes, software teams hesitate to imply UCD activities due to their time-consuming and effort-intense nature. Involving users from starting phases of development, especially in the case of iterative and incremental development methods, can help identify poor usability and design defects early in the development cycle, and prevent product failure at the end [4, 66, 98]. As consequences, UCD approach emphasizes on methods and techniques for involving end users from the early stages of development and guides the design of user interface (UI) and its evaluation by integrating user experience as part of software development process for improving overall product quality [123].

The agile approach [1, 2, 31] is used for constructing software products in an iterative and incremental manner; in which each iteration produces working artifacts that are valuable to the customers and to the project. This is performed in a highly-collaborative fashion to produce quality products that meet the requirements in a cost-effective and timely manner. When we look into agile development methods, normally all focus heavily on working code of the system rather than making documentation and other artifacts, which could result in conflicts for the future understanding of requirements. They emphasize on significant customer collaboration, but the compressed time scales for iterations sometimes make it difficult to get access to the right customer at the right time to get the feedbacks on previous iterations and prototypes [25]. Moreover, if the accessed customers are not the real end users then it could lead to misunderstanding of the demands of real end users of the future system. On the other side, UCD activities are normally time-consuming

and take extra efforts. Thus, there are risks of compromising attention on usability in a too quick agile development. Examining UCD limitations, it is sometimes perceived as waste of time and money when an intensive iterative and paper-producing UCD development takes place.

Although these two approaches, UCD and agile software development, are different approaches that were raised within different disciplines seems to contradict each other from the top view, but when we examine in detail the basic set of concepts and philosophy, we find no fundamental contradictions. In fact, when we go into the details we find more similarities than the perceived contradictions [9]; like, agile development approach focus on people is similar to UCD focus on end users, the agile approach of intense customers' collaboration is similar to the users' involvement in UCD activities, both approaches emphasize on communication with customers or end users for better understanding and proper designing of the product, both can change dynamically their process according to the target environment, agile focus on ease for customers' needs is same as UCD focus on usability for end users, both also emphasize on the design improvements based on customers or end users continuous feedbacks.

Thus, by emphasizing the benefits common to both the end users and the developers, UCD and the agile development approach can be dynamically integrated to get benefits from both, resulting in the development of high-quality and usable software products. Even the small investment of UCD activities in agile development gives the benefits in large [23]. For example during development, agile development team members participate in different roles and by integrating UCD activities; it will increase their knowledge and understanding about the domain and the end users [48], and as a result improves overall quality of the end product.

There are many challenges in the way to produce a proper working integrated approach; for example, due to the short iteration-time in agile development there is no time for field studies or making alternative user interfaces or testing prototypes and same is true for many other UCD activities [23]. Especially, in terms of agile development, more care is needed to be taken as inappropriate integration may lead away from the basic philosophy of agile approach. In spite of the growing research in both fields, we identified that agile development teams were often lacking a properly-integrated approach that utilizes the UCD philosophy from end-to-end at all levels.

The Three Levels of Integration

On the above findings to overcome the gap, we propose a three-fold integration framework that gives suggestions and recommendations for involving UCD in agile software development at three levels. Our approach emphasizes a tight

integration from top-to-bottom, in which shared ideas are combined at every level, from the process life-cycle to the development environment, gaining the benefits of both approaches. Our three-fold integration framework incorporates UCD into agile development at three levels: the process life-cycle level, the iteration level, and the development-environment level.

1) The Life-Cycle Level Integration – Selection and Application of UCD Methods and Techniques in the Agile Process Life Cycle:
We define life-cycle level integration as performing appropriate UCD methods and techniques in the right places at the right times, alongside the other development tasks. When integrating with agile methods, a careful selection of UCD techniques at each phase of life-cycle is needed to achieve the maximum benefits. We distinguish UCD methods into elicitation and evaluation, and suggest a number of attributes for selecting appropriate methods during different phases of development. On the base of these recommendations, we suggest a life cycle for integrating UCD into agile development.

2) The Iteration Level Integration – Integrating UCD Concepts, Roles, and Activities in Agile Development Iteration:
This level of integration helps to align UCD concepts, roles, and activities within the development iteration activities for maximum benefit. For example, our framework suggests that UCD tasks exist in addition to the development tasks in every iteration planning. Another example is integrating the role of the "design evaluator" in development team whose responsibilities are to plan the evaluation of the user interfaces design, gathering and analyzing evaluation data, and recommending UCD tasks for next iterations accordingly. The results of these tasks are presented in the iteration presentation. This level integration resulting effects into agile development are performing iterative design activities, taking measurements, and defining UCD roles.

3) The Development-Environment Level Integration – Managing and Automating User and Usability Evaluation in IDE:
UCD guides integrating user experience into the software development process. One of the challenges of this integration is to automate the management of UCD activities during development. When we analyzed current software design practices, we identified a lack of *UCD management*, which we define as the ability to steer and control the UCD activities within the development environment of the project. Defining evaluation experiments and running them from within IDE equips the software team with the mechanism to monitor and control a continuous evaluation process, tightly coupled with the development process, thus receiving ongoing user feedback while continuing development.

It is interesting to note that although our framework targets toward agile-based development approaches, but the given suggestions and recommendations are also applicable for other development approaches that come under the umbrella of iterative and incremental development methodology. The following three sections discuss the aforementioned three levels of integration in detail, followed by the tool that is based on the automation of the *UCD management* concept.

3.3 The Life-Cycle Level Integration

Our integration framework suggests tighten integration of UCD methods and techniques within the development life cycle. This means performing appropriate UCD methods and techniques in the right places at the right times, alongside other development tasks. Selection of appropriate UCD methods and techniques is critical due to the agile nature of development where there are short iterations and focus is towards the working part of developing product rather than investing much time on documentation. Following we categorize UCD methods into two groups, suggest a number of attributes for selecting appropriate methods, and a life-cycle for integrating UCD activities into agile development activities.

3.3.1 Method Categories

We distinguished the different types of UCD methods into two groups, elicitation and evaluation, based both on the way the methods perform and on their impact on software project development.

Elicitation Methods: These UCD methods are used for eliciting requirements and design of the software project. Normally, the end users or UCD experts are involved during the initial phases of development life cycle. They are useful to get requirements properly and to identify the drawback early in the design phase. We suggest using these in early activities of iterations to give more attention to eliciting requirements and design. Among the different elicitation methods, those that take less time, efforts, and give high feedbacks, such as focus groups and card sort methods [110], are more suitable are more suitable as they fit perfectly in agile development.

Evaluation Methods: These UCD methods involve end users, UCD experts, and automated tools and are used to evaluate developing/developed products by identifying usability issues. Each type of evaluation method highlights only parts of usability issues, and only to a certain extent, so using more than one method is recommended [63] for covering a higher rate of usability issues.

Attribute	Description
Automation	The automated tools support and the level of automation (e.g., None, Capture, Analysis, Critique (taxonomy by Balbo [6])
Effectiveness	Feedback effects (Low, Medium, High) on design or development
Dynamicity	The ability of the method to be changed according to the target environment (Low, Medium, High)
Time-cost	How much minimum time is needed to complete this method
Effort-cost	How much efforts are needed to perform this method (e.g., man power, equipments, experiment place, other resources)
Ease of Learning	How easy it is to learn the method, both for responsible persons on the development team and/or for the end users who will perform it
Results Accuracy	Accuracy of results
Coverage Area	The elicitation/usability issues covered by this method

Table 3.1: Set of attributes for determining the selection of appropriate UCD methods

The evaluation methods performed by UCD experts are useful in early design phases, in which only paper prototypes or only parts of working prototypes are available, as these methods take less time and effort, both of which are at a premium during these early phases of development. On the other hand, evaluation methods performed by end users give better results when used on working prototypes, as these prototypes help end users understand the system, taking better advantage of the methods. Heuristic evaluation, question-asking protocol, and performance measurement [25, 63] are all examples of evaluation methods.

3.3.2 Selection Criteria

Our framework provides a set of attributes for selecting appropriate elicitation and evaluation methods to apply during agile development phases. Table 3.1 shows these attributes and describes each one from the agile development perspective, i.e., short-time iterations, high-level of collaboration with customers, focus on working artifacts, and dynamic processes.

The *automation* attribute tells us the nature of automation supported by automated tools for the selected method. We use the taxonomy suggested by Balbo [6] for the nature of automation, which consists of: *None* if no automation is supported, *Capture* in which the automated tool records and captures users and system information, *Analysis* in which the automated tool provides the automatic analysis of the recorded data such as some usability problems

due to user interface, and *Critique* in which the automated tool also provides the possible solutions to the identified problems. The more automation the better, as it saves time and cost and provides much accurate results. The *effectiveness* attribute is to assess the method's impact on designing and/or development. A method with high impact gets more priority than the lower one, e.g., heuristic evaluation with usability experts on early prototypes can have more impact on design improvements compare to other evaluation methods.

The agile processes are able to adopt the target-working environment; therefore, methods with have higher degrees of dynamicity are natural alliance. The *dynamicity* attribute is to judge a method's ability to change its process of working according to the target environment. The *time-cost* attribute is to determine the time needed to perform a particular UCD method. This has a direct relation with the time frame of agile iteration. Therefore, we suggest choosing those methods that can fit properly in the time frame of the targeted iteration. The *effort-cost* talks about other resources (e.g., man power, equipments, money, experiment place, etc) that are needed to perform the selected method. This also has a direct relation with the targeted agile iteration. For example, normally agile teams are made up with small set of people and a particular method may not be suited well if it requires more man power than the team's capability.

The *ease of learning* attribute talks about a UCD method's understandability and learnability both for the responsible persons on the development team and/or for the end users who will perform it. A method with higher ease of learnability takes less time while performing it, hence compliment to agile short nature of iterations. A method's results act as feedbacks for the improvements in the design and development of the developing product. They also effect the forthcoming iterations' planning. Therefore, the *results accuracy* of a particular method can play a critical role in the success outcomes of agile development iteration. Each UCD method covers only parts of the problem, i.e., a usability method highlights usability issues from a certain perspective. The *coverage area* describes the perspective of highlighting eliciting issues or usability issues by a particular method. Therefore, it is better to select those two methods that highlight issues from different perspectives rather than those two who do from the same perspective.

Along these attributes, selecting an appropriate UCD method also depends on other factors, such as life-cycle stage, availability of participants, etc. For early design activities, we recommend an emphasis on paper-based or simple UI prototype-based evaluation methods to improve design early. While in later iteration activities, we recommend using formal evaluation methods, such as task model-based usability evaluation method, to get formal results. We also

recommend using a mixture of evaluation methods, preferably supported by automated tools and performed by end users and UCD experts for maximum results. Automation tools support gives more accurate results and save time and costs-all factors that complement agile development.

3.3.3 A Life-Cycle for Involving UCD in Agile Development Iteration

On the basis of the above recommendations, we suggest a life cycle of four UCD activities for involving UCD philosophy alongside the agile development iteration. Figure 3.1 shows the UCD activities (solid ovals), in which agile activities that are done per each user story or development task are represented by dashed lines ovals. Sometimes a UCD activity overlaps one or more agile activities.

UCD Involvement: The software team involves end users and UCD experts when appropriate, mostly through the use of elicitation methods during work on requirements, design, and early prototypes.

Design-Artifacts Evaluation: The early design and prototypes are then quickly evaluated through short-time consuming evaluation methods normally by UCD experts (i.e., system analysts, usability evaluators, UI designers) and sometimes by a small group of end users. The results of this phase become input for the third phase.

Design Improvement: The software team corrects and improves the design according to the feedback and performs implementation of the target modules.

Detailed Evaluation: The developed modules are evaluated in detail by end users and/or by UCD experts, normally through rigid evaluation methods with automated tools support. The results, feedback, and suggestions serve as input for making plans for improvements in the design and for the implementation of developing products in the upcoming iterations.

3.4 The Iteration Level Integration

This level integration helps to align UCD concepts, roles, and activities within the development iteration activities for maximum benefit. Due to the short time nature of agile iteration, a careful integration is needed in planning for each iteration; hence, the evaluation feedbacks can be gathered effectively and can be used to improve the product accordingly. In the following subsections, first we highlight how the end user role specified in UCD philosophy differ from

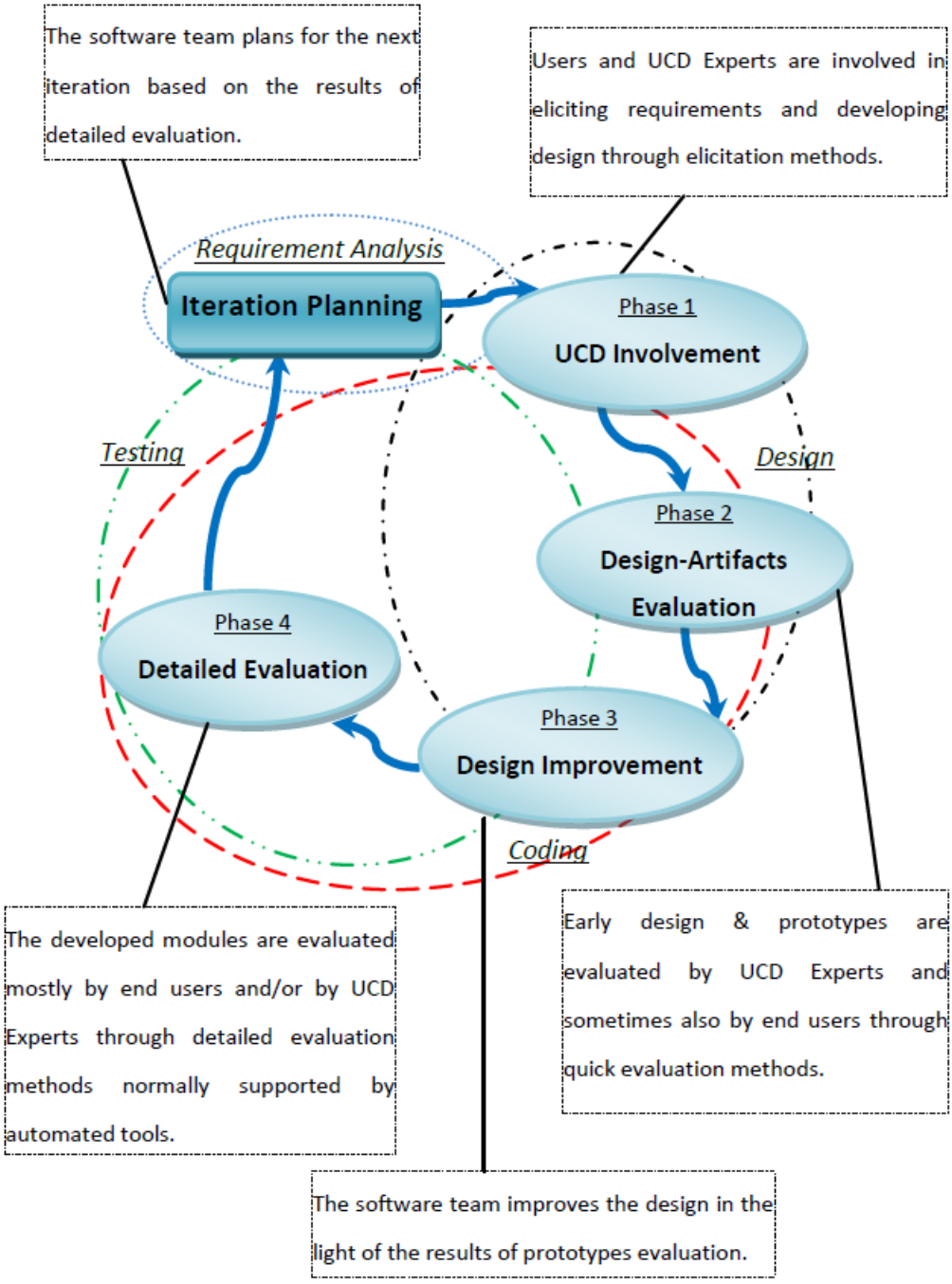


Figure 3.1: Life cycle for involving UCD philosophy into agile development iteration

the customer role in the agile development. Then we describe this level integration resulting effects into agile development through performing iterative design activities, taking measures, and defining UCD roles. Lastly, we present use cases as scenarios from the world of agile teams to represent the effects of integration in the form of users' involvement in development, evaluation of the product, and design improvements. These use cases scenarios form the foundation on which our framework lays the foundation of management and automation of user evaluation in development environment (Section 3.5) and the tool (Section 3.6) that we developed to realize it.

3.4.1 The Users' Role in Software Development Processes

The standard ISO 9241-11 [115] identifies the following as the most useful indicators for measuring the level of usability of a product:

- *Effectiveness in use*, which encompasses accuracy and completeness through which users achieve certain results.
- *Efficiency in use*, which is related with the resources utilized in relation to accuracy and completeness.
- *Satisfaction in use*, which includes freedom from inconveniences and positive attitude toward the use of a product.

In light of this standard, we bring the perspective of the customer and the user for whom the software is developed. We distinguish between the role of the customer and the user while focus on the way users should be involved in agile software development methods.

The Customer

The customer's position in software development processes is one of the main changes that the agile approach introduces into software development environments. The customer role in agile software development environments is central and is based on on-going communication between the customers and the team members, both with respect to the requirements and the way testing and checking of the developed product are performed. This communication is established with the aid of several practices, one of which is the planning session. During this session, the customer observes the current developed artifacts, gives feedback, and prioritizes the work for the next iteration.

The User

The agile approach for software development emphasizes '*individuals and interactions*' [2] during the process of software development [2, 7]. Asking soft-

ware practitioners who these individuals are, most of them mention roles like system analysts, developers, and testers. The agile approach increases the awareness to additional roles like the customers who are most important to collaborate with. There are roles schemes that are used in different agile methods and are discussed for different goals, e.g., in [28] for the sake of an academic project. The current agile development practice broads the customer role as a group, which includes a number of stakeholders and shareholders like business analysts, higher management, interaction designer, end users, QA persons, etc. In spite of all, still, the end users themselves and the design that follows their evaluation are somehow neglected. A common misconception and practice is not to include the end users in the customer group, and if included then many times the attention is more focused towards others in the group rather than end users, like higher management or business analyst, who have more influence or direct communication with the development team.

Given that the attention is normally paid more towards those who sometimes pay for the software development (shareholders) or have other kinds of interest with the development (stakeholders), the end users are the major group of individuals from stakeholders in the context of most software projects. But, generally in agile development practice, due to time constraints and other circumstance, end users are either completely ignored or receive little attention. Combining UCD philosophy and agile approach pays more attention towards end users' needs and puts them, along other customers, at the center of every interaction, and includes methods to deal with end users' evaluations and its implications to design and development.

3.4.2 User Experience and The Development Processes

The iteration level integration brings the following effects into agile development:

- ***Iterative design activities:*** In many cases, when UCD techniques are used (if at all), the design of the system is refined according to the users' evaluations mainly during the design phase. In the agile development approach, the design is updated regularly as the product evolves. When combining the UCD approach with agile development, the user evaluation is fostered by performing UCD activities in each iteration of two to four weeks, and the design is updated according to the evaluations' ongoing outcomes.

- ***Measures:*** Taking measurements is a basic activity in software development processes. When combining the agile and UCD approaches, a set of evaluation tools is built and refined during the development process and is used iteratively to complement the process and the product

measures.

- ***Roles:*** Different roles are defined to support software development environments. The agile approach adds roles for better management of the project [114]. Combining the agile and UCD approaches adds UCD roles, such as the design evaluator or the usability expert, to support and carry on UCD activities in the development.

The following are use cases as scenarios from the world of agile teams, in three categories to represent the effects of UCD activities integration in agile development iterations. In these use cases, the phrase User Perspective is used to refer to the perspective that supports UCD management.

Users' Involvement in Development Process

There is a need to involve end users in the process of development.

Following are examples for use cases that relate to this category:

- One of the tasks during the first planning session is as follows: "Explore the different kinds of users who should use the product that we develop; what are their characteristics; what are their needs; what are their expectations from the product." A customer is selected from a general customers' group. The customer explains that he/she is unable to represents all possible end users and even not certain of the exact requirements. However, the selected customer asserts that the proposed product has a great potential and is likely to attract customers. One of the teammates asks to be assigned to this task and estimates it as 10 hours of work for this iteration. Presenting her results after two weeks, she opens her development environment in the database of the *User Perspective* and shows the list of 20 users she talked with (names, titles, contact details, etc), main issues that were learned, and one new task that has emerged for future iterations: "Prepare and run a questionnaire that will enable us to extract users' needs." The customer sets high priority for this new task.

- The project manager reviews the subjects for the upcoming reflection session, and sees that one of the subjects is "ways to assess the usability of our product". She then sends invitations to seven users from the two different kinds of user groups to join this meeting. During the reflection session, one decision is made that two users will participate in each iteration planning session and their responsibility will be to give feedbacks on what is presented? In addition they will help in defining three measures

that will be automated thus enable teammates to receive an immediate feedback during development.

User Evaluation

There is a need to perform user evaluation and to manage it along the process of development.

Following are examples for use cases that relate to this category:

- The team leader browses over the details of the user experiment that is planned for tomorrow. He sees the number of participant users that will arrive, the names, and responsibilities of the two teammates that will take care of this experiment. He checks the variables that were set and the experiment flow.
- One of the teammates sees that the User Perspective flushes meaning new data has arrived. He clicks on it and sees that the results of the user experiment that was conducted yesterday are in. He is surprised to find a new problem with high severity ranking. Examining results from previous experiments, he observes that this is a new problem and adds a note about it in the discussion area. During the next iteration planning, the experiment results are presented and among others, a measure is presented that shows two problems that have emerged by evaluating end users, one in normal severity and one in high severity.

Design Improvement

There is a need to improve the design of the user interface or performance of a module based on the evaluation results.

Following are examples for use cases that relate to this category:

- The designer of the user interface views the latest design diagrams and tries different changes that adhere to the new task in this iteration. The task was added due to the last problem that was found during the evaluation by end users. Thinking of different options, she talks with two users from evaluating end users group and receives their feedback. She shows them the possible drawings of the new interface and asks them to simulate using it, while thinking aloud. She summarizes the results and sets her decision.
- One of the teammates browses over the system reports and sees for each user experiment, which was conducted in the last two releases. He/she

looks the results and their implications on design, and the development tasks created against each implication.

3.5 The Development-Environment Level Integration (*UCD Management*)

Our framework contributes the automation and management of UCD activities for user and usability evaluation in development environments to enable the creation of experiments; including the use of evaluation aspects, analysis of results, and code traceability. The motivation behind integrating and automating the evaluation process into the software development environment, i.e., into the Integrated Development Environment (IDE), is clear. Defining evaluation experiments and running them from within the IDE equips the software development team (especially in the case of agile development) with the mechanisms to monitor and control a continuous evaluation process tightly coupled with the development process, thus receiving on-going users and UCD experts feedbacks while continuing development. We argue that automating and managing these activities at the IDE level will improve the efficiency of the agile software development team so they are better equip to deal with schedule and budget constraints, and produce software projects with an adequate level of usability. This also helps in reducing the gap between the usability evaluation side and the development side.

Experiment Entity

Our approach towards automating the evaluation methods means that we can add a new kind of an object in the development area of a software project. These objects, following known as *experiments* because of the controlled environment in which they are performed, can be created and executed to provide evaluation data. Furthermore, an experiment's results can be associated with future development tasks as they emerge. We further divide these experiments into three categories: *expert-based* experiments, *user-based* experiments, and *system-based* experiments.

- ***Expert-based experiments:*** These experiments are based on the evaluation methods performed by either UCD experts or system experts. Normally in this case, these experts evaluate the usability of the system with respect to certain standards or guidelines. Most of these methods come under the *Inspection* type defined by [63] (p. 475). These kinds of experiments are good to highlight a large portion of usability drawbacks against defined standards and guidelines. Heuristic Evaluation

[83], Cognitive Walkthrough [74], Feature Inspection [84] are examples of methods whose experiments fall in this category.

- ***User-based experiments:*** These experiments are performed by end users of the developed/developing system. In these experiments, the evaluating users are selected from a pool of end users on different basis such as age, gender, expertise, etc. The evaluating users perform different tasks on target system or evaluate the system based on any given criteria. The resulting feedbacks; such as performance time, difficulties, completed tasks, etc; are then analyzed and provide a mean to highlight usability issues. In some cases, end users are asked deliberately to give feedbacks after analyzing the evaluating system. Question-Asking Protocol [67] and Performance Measurement [83] (p.191-194) are examples of methods whose experiments fall in this category.

- ***System-based experiments:*** These experiments use automated evaluation tools to record users' and system behavior while end users work on the system, and produce analysis of the recorded data. In these experiments, normally end users are not asked deliberately to perform any specific tasks. In many cases, users are even unaware of the type of data collected by these automated tools. These automated evaluation tools keep track of users' and system activities and behavior while users interact with the system. Few tools also analyze the recorded data and produce analysis of these results, while others present the raw results in a presentation form. Log File Analysis and Task-Environment Analysis are examples of methods where the supported evaluation tools analyze the recorded behavior of users' log.

Figure 3.2 shows the overall structural division of UCD methods as described above and in Section 3.3.1. The above described three categories come under the evaluation methods type. The figure also shows the names of few standard methods in each category, and our recommendations for using these during specific development phases.

Derived Development Tasks

Each kind of evaluation experiment has its own criteria for judging the usability level of the product. Support for the analysis of the experiments' results enables the comparison of these results against the targeted usability criteria. If the results show a failure to achieve the target usability level, then new development tasks can be defined accordingly. For example, if the system response was slower than the expected time then a new suggested development task might be to make improvement in it. Each development task is associated with the relevant data, thus providing its rationale.

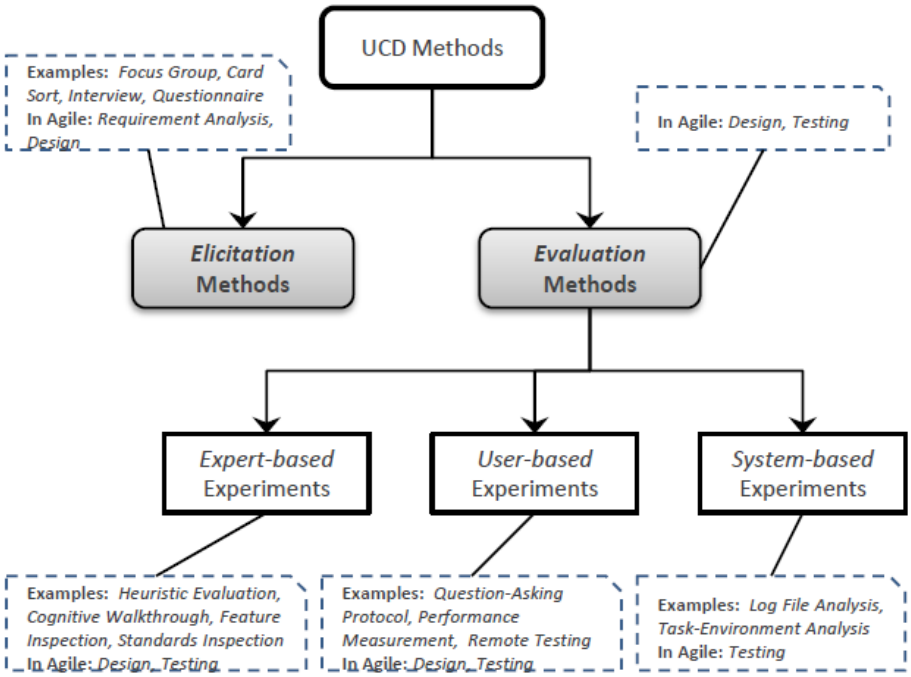


Figure 3.2: Structural division of UCD methods

Code Traceability

Generally, in all iterative and incremental development approaches and specifically in agile development approach, the software product evolves through iterative steps in which the design of the software project improves gradually. Automating the process of backward and forward traceability among different evolving parts, at the development-environment level provides a better traceability of the refinement carried out in the design to improve the product. Such parts could include code parts, experiments, and derived development tasks. One of the benefits of this mechanism is that it helps to learn about the impacts of the evaluation.

Developing Evaluation Aspects

Automating UCD activities in the development environment can enable developers to add automatic evaluation hooks to the software under development. For example, an aspect could be created to control the use of a specific button or key that is part of the developing software. These system-based methods that include such measures provide insights about the users' behavior.

A practical example of this is the usage of Aspect-Oriented Programming (AOP) [68] to facilitate such automatic evaluation. In brief, AOP introduces language constructs called *aspects* that separate cross-cutting concerns from an object-oriented system and provide a mechanism to weave these aspects into the underlying system.

3.6 UEMan: A Tool for UCD Management in Integrated Development Environment

Our approach towards automating UCD activities helped us to shape a set of requirements for the creation of a tool as well as guidelines and techniques to accompany it. We found that no existing tools support *UCD management* as part of the integrated development environment (IDE). We argue that UCD management at the IDE level will improve the efficiency of the software development team so they can better deal with scheduling and budget constraints.

We developed **UEMan** (**U**ser **E**valuation **Man**ager), a tool for managing and automating UCD activities, especially related to the usability evaluation, alongside the process of software development. The UEMan is an Eclipse [32] plug-in, developed in the Eclipse IDE using its Plug-in Development Environment (PDE) [119] facility to extend and be integrated into the Eclipse IDE. After becoming a part of the Eclipse IDE, the project team can view and use its facilities from within the Eclipse IDE. The main capabilities include, creating the experiment object as part of the software project; deriving development tasks from the analysis of evaluation data; and tracing these tasks to and from the code. Further, it provides a library to enable development of Java aspects for the creation of automatic measures to increase the breadth of the evaluation data.

The UEMan Evaluation Life-Cycle

In a nutshell, UEMan evaluation life-cycle consists of four phases that occur iteratively.

- ***Phase 1 – Evaluation Definition:*** The evaluation is defined through the following: different types of experiments (such as questionnaires, heuristic evaluations, etc); experiment tasks (which end users/UCD experts need to perform during experiments); role holders who are involved in experiments (e.g., end users, UI experts, evaluators); and other management considerations (such as experiment details, time to execute, etc).
- ***Phase 2 – Evaluation Execution:*** The evaluation is executed by running the experiments either locally at the evaluation site or remotely

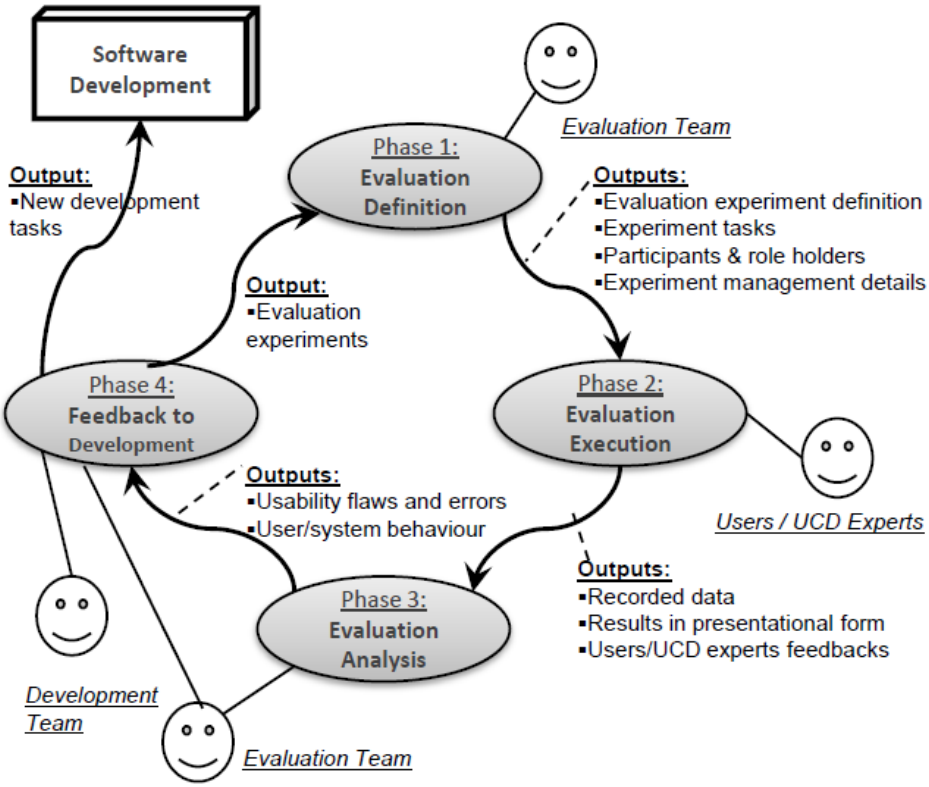


Figure 3.3: UEMan evaluation life cycle

at the participant's site; thus, tasks in the experiments can be performed by participants and the results can be stored in the system.

- ***Phase 3 – Evaluation Analysis:*** The results are analyzed by software development team and/or evaluation team as per evaluation experiment and any cross experiments to identify users' and system behavior as well as usability issues.
- ***Phase 4 – Feedback to Development:*** New development tasks for upcoming iterations are derived according to the evaluation analysis for improving the product, whereas connectivity is kept between the evaluation results and the relevant code parts during development.

Figure 3.3 shows the evaluation life-cycle as described above, managed in UEMan. It also shows that the evaluation team works in the first, third and fourth phase, users or UCD experts work in the second phase, while the development team work with the evaluation team in the fourth phase to finalize the new development tasks for further improvements. Sometimes, the development and

evaluation teams decide to perform further experiments based on the current results as shown by the outputs going from phase four to phase one.

The UEMan Evaluation Experiments Support

The UEMan supports different experiments in three categories as described in Section 3.5. The detail of supported experiments is presented below:

- ***Heuristics-evaluation experiment:*** In this experiment, UCD experts analyze the developing software to judge its usability level according to the guidelines of Nielsen heuristic evaluation [83].
- ***Task-type experiment:*** In this experiment, users perform tasks from a given list while using the developing/ developed software, without any direct help from the evaluation team. During this experiment, the UEMan measures different performance times and then the evaluation/development team analyzes the results to judge the system's usability level. The evaluating user can also give feedback during or at the end of experiment.
- ***Questionnaire-type experiment:*** The purpose of this experiment is to evaluate the system according to the end users' level of agreement, after they use the developing/developed software, with the presented statements.
- ***Logging-aspect experiment:*** The purpose of this experiment is to record users' behavior against the selected criteria (e.g., mouse clicks, key presses, timings, etc) while they use the targeted software. The UEMan then shows the recorded data for analysis purpose in different forms.

3.6.1 The UEMan Process for Managing and Executing Experiments

Following we describe UEMan capabilities in detail that can be used by the software team to manage and automate UCD activities for a particular software project.

UCD Role Holders:

UEMan provides the facility for creating and managing the team responsible for usability evaluation and assign them related UCD roles. It manages the evaluation team members into three categories: the users who participate in the user-based and system-based evaluation experiments, the external UCD

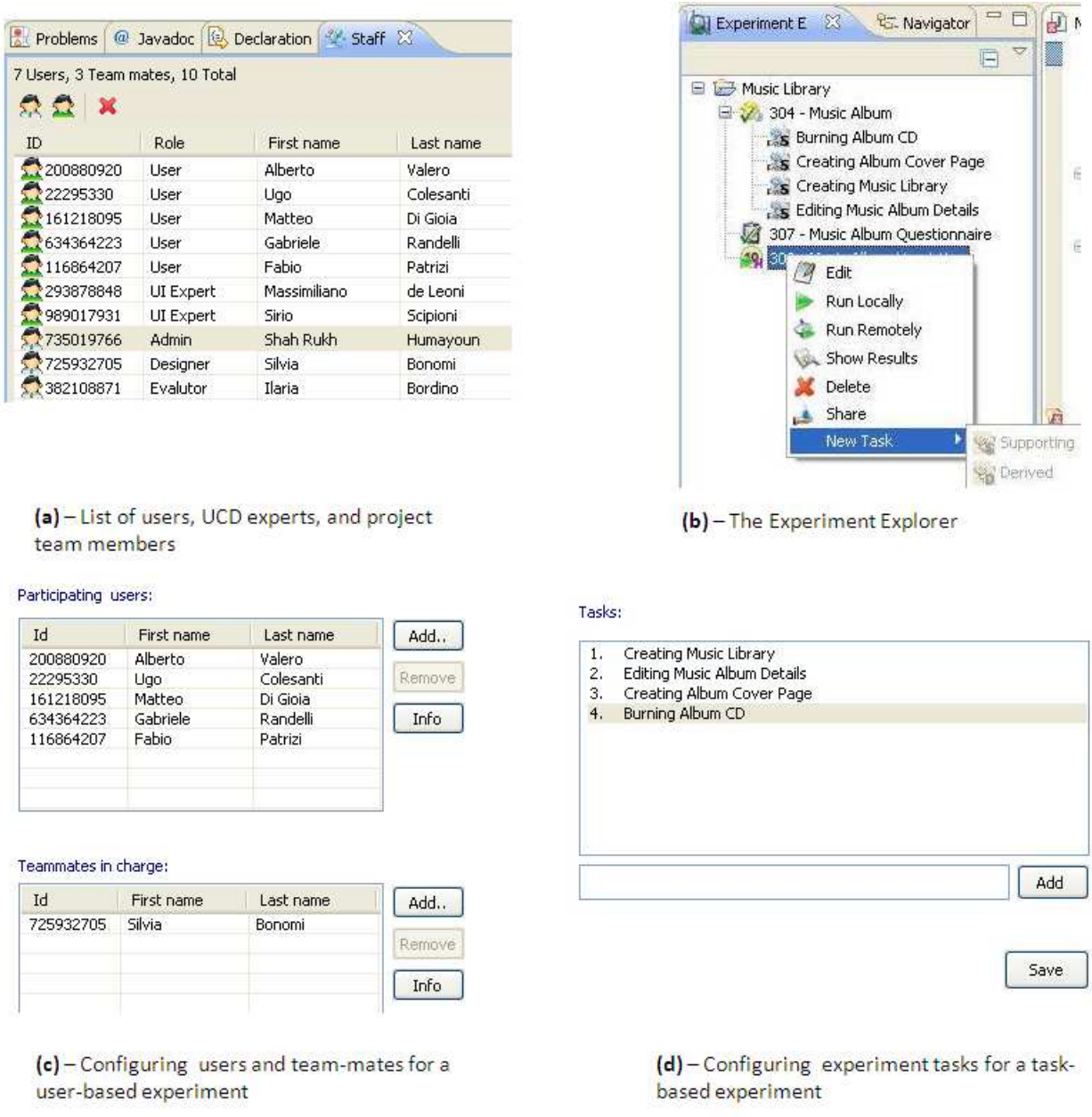


Figure 3.4: UEMan experiment management

experts (e.g., UI Expert, System Analyst, etc) who participate in the expert-based evaluation experiment, and the project usability team members (e.g., Designer, Evaluator). Figure 3.4.(a) shows a project view displaying a number of users, usability experts, and team members.

Management of Evaluation Experiments:

UEMan provides an *Experiment Explorer* that helps to create, manipulate, and automate evaluation experiments for a specific project (shown in Figure 3.4.(b)). It also facilitates sharing the data from these evaluation experiments among different projects.

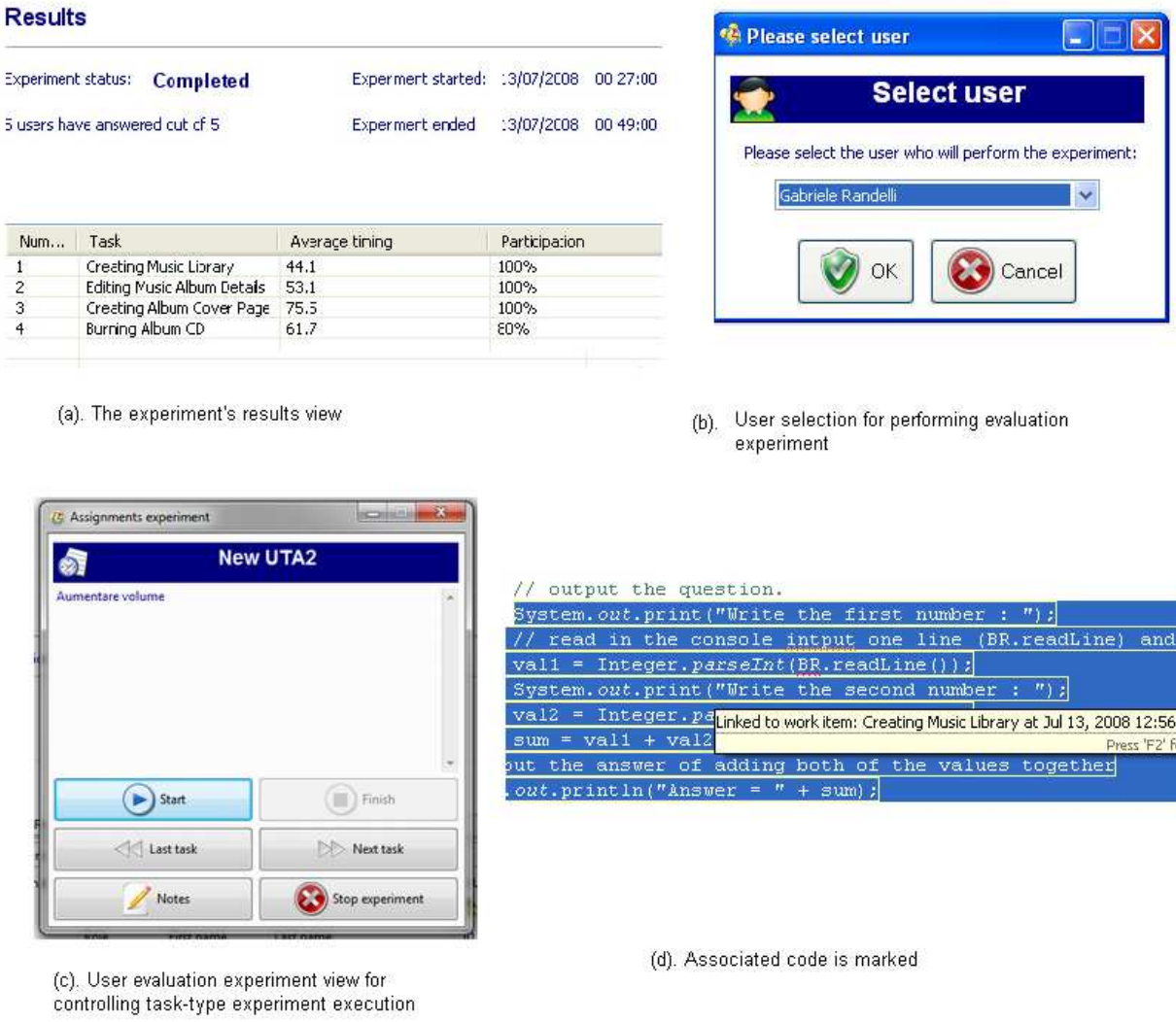
Using the Experiment Explorer, the software team can run the evaluation experiment either locally, i.e., on the server on which the data is stored; or remotely, such that the enlisted users receive an email with the evaluation experiment files attached and instructions for running the evaluation experiment so that the final results cab be directed back to the server for storage. Configuring an evaluation experiment is performed using the experiment's Configuration Wizard. Figure 3.4.(c) (part of the wizard) shows the option for adding participating users and teammates responsible for the evaluation experiment. Figure 3.4.(d) (part of the wizard) shows the list of tasks the participating users have to perform while executing that task-type evaluation experiment.While a user performs the evaluation experiment, UEMan measures different performance times to evaluate the usability level of the related parts of the software product. Figure 3.5.(a) shows the results view of a task-type evaluation experiment. You can see the average time (in seconds) and the level of the users' participation in performing each task.

Execution of Evaluation Experiments by Participating Users:

The user-based category of evaluation experiments are executed by participating users, who use the developing system to judge its usability level. Figure 3.5.(b) shows a user selecting his/her name for executing the evaluation experiment. An individual user at a remote site has only one user to choose from. The user starts experimenting. Figure 3.5.(c) shows which task is currently executing in a task-type experiment and different options for controlling the execution. For example, if the user wants to give feedback during or after the experiment then he/she can do it through giving notes, or if the user is unable to complete the current task in the experiment then he/she an skip it and move on to the next task.

Derivation of New Work Items and Code Traceability:

New work items (development tasks or evaluation experiments) can be derived using the facility provided by the *Experiment Explorer* if the usability



(a). The experiment's results view

(b). User selection for performing evaluation experiment

(c). User evaluation experiment view for controlling task-type experiment execution

(d). Associated code is marked

Figure 3.5: UEMan experiment execution

evaluation team is unsatisfied with the results of the evaluation experiments, such as results showing that the parts of the developing product are below the targeted usability level. UEMan also enables associating code files or code parts to related work items, and vice versa, for the purposes of traceability. These associations are important because on the basis of progress in the next iteration, UEMan can associate the newly created code parts to the previously derived work items, thus enabling continuous traceability. Figure 3.5.(d) shows a code part highlighting its association with a specific experiment.

3.6.2 Automatic Measures using Development Aspects

UEMan provides an AspectJ library, called *AutoMeasurement*, for adding AspectJ[1] aspects to the software under development to support automatic measures that fit the developing product. These measures, as part of a logging-aspect experiment, enable the developers to add automatic evaluation hooks in the software under development to record different kinds of user behavior, while using the evaluated software product. It uses Aspect-Oriented Programming (AOP) [68] to facilitate such automatic evaluation. In a brief, AOP introduces language constructs called *aspects* that separate cross-cutting concerns from an object-oriented system and provide a mechanism to weave these aspects into the underlying system. An aspect contains an action called *advice* and a definition of different events during the execution of the program called *join-points*, where the action should be executed. AspectJ, which is an extension to Java defines join-points such as method executions, method calls, and assignments to variables. An AspectJ aspect defines relevant join-points using *pointcut* descriptors. For example, we can define an aspect that logs each method call during the execution of the program; its *pointcut* is the set of all method calls in the program, and its advice simply prints a log entry to a file. An AspectJ aspect also defines whether the advice should execute *before* the advised join point executes or *after* it. Furthermore, an advice can be of type *around*, meaning that it can be executed instead of the advised join-point where the advice can then include a *proceed* statement that executes the join-point.

Using the *AutoMeasurement* library in UEMan, the software developers can create and implement AspectJ aspects that are customized for the specific software. An aspect can be created to measure time of user activities or to control the use of a specific button or key that is part of the developing software. Running an experiment that includes such a measure provides insights about the users' behavior. The aspects in the library are abstract, so to utilize these, corresponding concrete sub-aspects must be implemented. While an abstract aspect generally defines the kind of measurement to handle, the

[1] AspectJ project, see http://www.eclipse.org/aspectj/

concrete implementing aspect connects the abstract measurement to a base system by specifying the exact locations within the system's code to which the measurement is targeted. An example from the UEMan library for an abstract Timer aspect that measures the duration of a defined time interval is presented is presented in Listing 1.

Listing 1. *An abstract Timer aspect that measures the duration of a defined time interval.*

```
1 public abstract aspect Timer {
2   private long time = 0;
3   public abstract pointcut startPoint();
4   public abstract pointcut endPoint(Object obj);
5
6   after() : startPoint() {
7      time = Calendar.getInstance().getTimeInMillis();
8   }
9   after(Object obj) : endPoint(obj) {
10     time = Calendar.getInstance().getTimeInMillis() - time;
11     log(obj.getClass() + ": " + time/1000);
12     time = 0;
13  }
14  ...
```

The *Timer* aspect measures the duration of a time interval defined by a start point and an end point. It can be used, for instance, to measure the time a user takes to complete a certain task or to locate a certain feature. As seen in the listing, the aspect defines two abstract pointcuts *startPoint*( ) and *endPoint*( ). As explained, a *pointcut* construct defines a set of system events (join-points). An abstract *pointcut* does not declare any join-points, and it is the responsibility of the concrete *pointcut* within the implementing aspect to specify these (e.g., a method call corresponding to completion of a user's task for the *endPoint*( ) pointcut). The advice actions in lines 6 and 9 declare the action taken by the aspect when the pointcuts are matched. The code within the first advice sets the current time after the start point is notified. Upon activation of the end point, the second advice is activated and sets the time difference and logs it, including the type of a system's object that identifies the operation.

Figure 3.6.(a) shows the creation wizard, including the type of the abstract aspect to be implemented. Figure 3.6.(b) shows the code of the created concrete aspect and the pointcuts to be implemented. Using the *AutoMeasurement* Library, the development team can analyze automatically recorded evaluation data to assist with the future design of the product. The recorded

data can be viewed as text or using visual graphs. For example, Figure 3.6.(c) shows the UEMan graphical view of the time the user has spent in different windows when evaluating the Lobo[2] Java web browser. This kind of logging-aspect experiment provides a deeper insight of user involvement and shows how UCD activities can be incorporated into software development.

3.6.3 UEMan Evaluation Studies

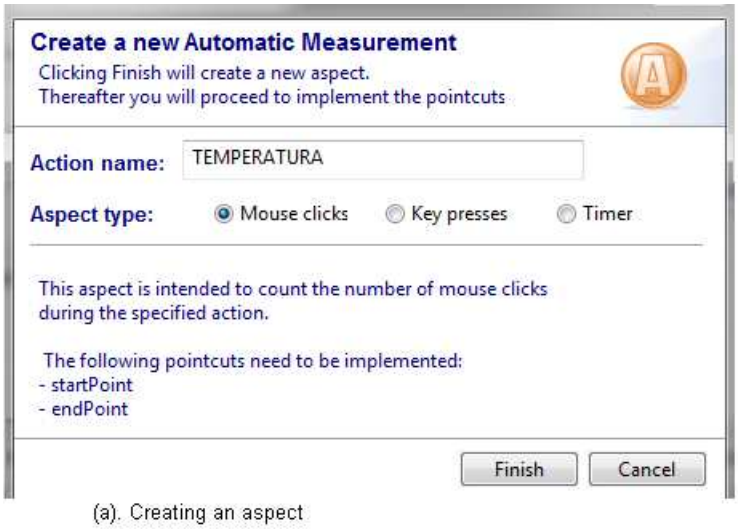
Evaluating UEMan is a challenge since UEMan itself is used to evaluate an application under development. This implies that the goals of this evaluation study are to assess UEMan usability and to check if and how it contributes to the evaluation of the developed application. In the following we present two evaluation studies. At first, during preliminary evaluation study, the team who developed UEMan was asked to evaluate its own product using itself ("eating its own cookies"). While in the second evaluation study, six software development teams (each developed their version of the same application) in academia conducted the evaluation (by other development teams) of the developed application using UEMan.

Preliminary Evaluation Study

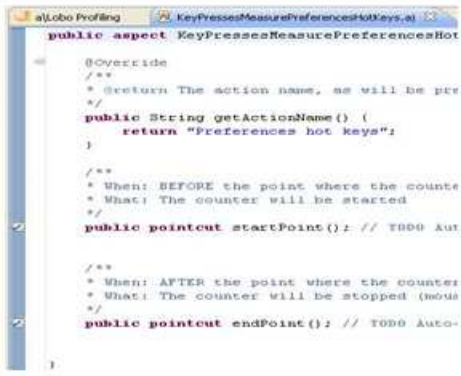
The evaluations goals were:
– Examining suspicious issues like adding new users to the system and analyzing the experiments' results (specifically for the questionnaire-based experiments).
– Receiving feedback on the graphical user interface (GUI) and how intuitive it is.
– Examining the plug-in on a large scale project.

The team defined two experiments: a task-based experiment and a questionnaire based one. The participants were three students from another team in the same course, and in addition all the six developers performed both kinds of experiments. Each participant performed the experiment by himself/herself while one observer was sitting aside for writing notes. We focus on the questionnaire-based experiment and comments of the observers and show an example of a derived task that emerged for further development. Figures 3.7.(a) and 3.7.(b) show the results of the three participants from another group and the results of the developers themselves respectively. Following are few comments, for example, that were noted by the observers:
1). "In the questionnaire-view that is presented to the participant, long tasks appear truncated."

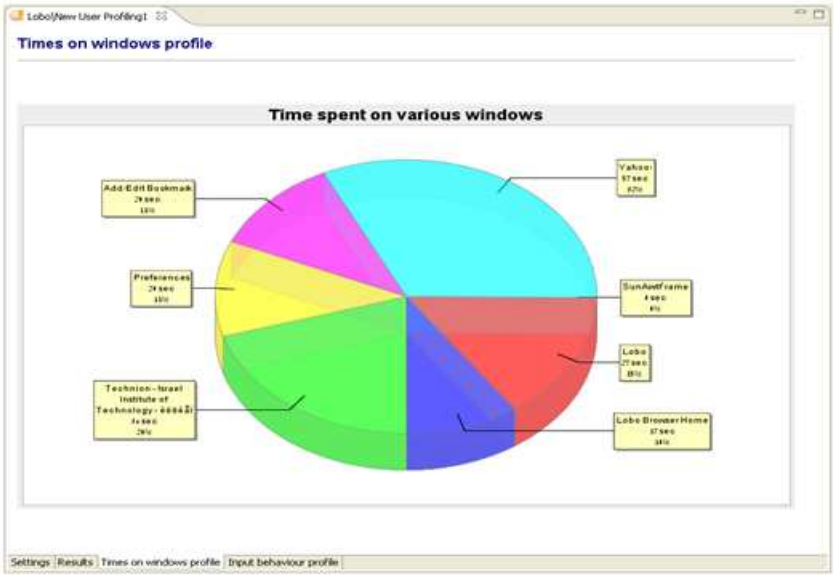
[2]Lobo web brwoser at http://www.lobobrowser.org/



(a). Creating an aspect



(b). Implementing the pointcuts



(c). UEMan view showing the timer results using the Lobo browser

Figure 3.6: UEMan automatic measures using Development Aspects

2). "The participant did not know how to save the changes in the result page. He searches for a save button like appears in other screens."
3). "The names of the operations in the menu of the experiment view are not clear."

Analyzing the results of both experiments, associations to the specific results were presented for each conclusion, and then suggested development tasks were associated to the conclusions. One of the finding, for example, was detailed as follows: "It was found that there is a difficulty in identifying problems in the product out of the information that is presented in the 'results page'. Participants find it hard to associate the results (as presented in the 'results page') to the experiment goals and to the practical problems that were discovered."

Association to the results:
– In the questionnaire-based experiment, the two teams marked 'Disagree' for statement 6 [The questionnaire result page displays the usability problems discovered in a clear way].
– In the task-assignments experiment, it took long time, 84 and 177 seconds on average for the two groups, to complete task 5 [According to the experiment goals, try to assess the number of usability problems indicated by the results, and write that number as a conclusion to this experiment].

The development task that was defined using the plug-in is as follows: "Enable determining thresholds for success and failure in an experiment and present them clearly in the 'results page."

Detailed Evaluation Study

In the second case study, six development teams developed an application, named FTSp[3] (Follow the Sun plug-in), to support synchronization between distributed teams that have no synchronous communication between them due to large time zone differences. Six development teams (each developed their version of the same application) conducted the evaluation (by other development teams) of the developed application using UEMan by defining and executing the heuristics evaluation experiment and logging-aspect experiment.

The evaluation study was conducted as a team exercise. The members of each team *i*, in addition to running their evaluation exercise, served as inspectors for team *i-1* in the heuristics evaluation experiment and as users for team *i-2* in the logging-aspect experiment. This way, each team worked with twelve participants, six for each experiment. Teams were asked to summarize the

[3]The project was developed by thirty-seven students working into six groups as part of the course 'Annual Project in Software Engineering' during the session 2008/09 in Computer Science Department at Technion, IIT.

		Strongly Disagree	Disagree	Agree	Strongly Agree
1.	Logging in to the system is simple.	0	0	0	6
2.	Adding a user or a teammate to the system is simple.	1	3	1	1
3.	Switching between teammates is fast and simple.	3	2	0	1
4.	The configuration page is intuitive.	0	0	2	4
5.	The Questionnaire result page displays the level of agreement (p...	0	0	1	5
6.	The Questionnaire result page displays the usability problems disc...	3	2	0	1
7.	The different editors and views of the plug-in are uniform and foll...	0	1	4	1
8.	The different editors and views of the plug-in blend seamlessly in...	0	0	3	3
9.	I would use this plug-in to test the usability of an application in de...	0	1	2	3

(a). Results of questionnaire-based experiments – participants from another team

		Strongly Disagree	Disagree	Agree	Strongly Agree
1.	Logging in to the system is simple.	0	0	0	6
2.	Adding a user or a teammate to the system is simple.	1	3	1	1
3.	Switching between teammates is fast and simple.	3	2	0	1
4.	The configuration page is intuitive.	0	0	2	4
5.	The Questionnaire result page displays the level of agreement (p...	0	0	1	5
6.	The Questionnaire result page displays the usability problems disc...	3	2	0	1
7.	The different editors and views of the plug-in are uniform and foll...	0	1	4	1
8.	The different editors and views of the plug-in blend seamlessly in...	0	0	3	3
9.	I would use this plug-in to test the usability of an application in de...	0	1	2	3

(b). Results of questionnaire-based experiments – team members are the participants

Figure 3.7: UEMan preliminary evaluation study results

results including their own evaluation and severity ranking, group brainstorm, and final results. Based on the final results, teams were asked to suggest three specific development tasks for forthcoming iterations. A total of twelve experiments were conducted to evaluate FTSp.

We focus on the heuristics evaluation experiments. The FTSp teams defined the heuristics evaluation experiments and participants executed the experiments, using the product freely and providing evaluation comments, according to these heuristics: visibility of system status, matching between the system and the real world, user control and freedom, consistency and standards, error prevention, recognition rather than recall, flexibility and efficiency of use, aesthetic and minimalist design, helping users recognize, diagnose, and recover from errors, and Help and documentation. A total of 206 comments were reported for the six products (lowest 25, highest 51).

After a brainstorming session to discuss the final comments, participants ranked the severity of the comments. Figure 3.8 shows the results of average ranking of participants for each problem per heuristic; for example, two problems were found concerning heuristic no. 10 each is represented with a dot. Each of these problems was ranked according to severity by all participants and the average (in this case between 1.5 and 2) sets the dot in the vertical 'y' coordinate. The team summarized that the highest severity ranking was given

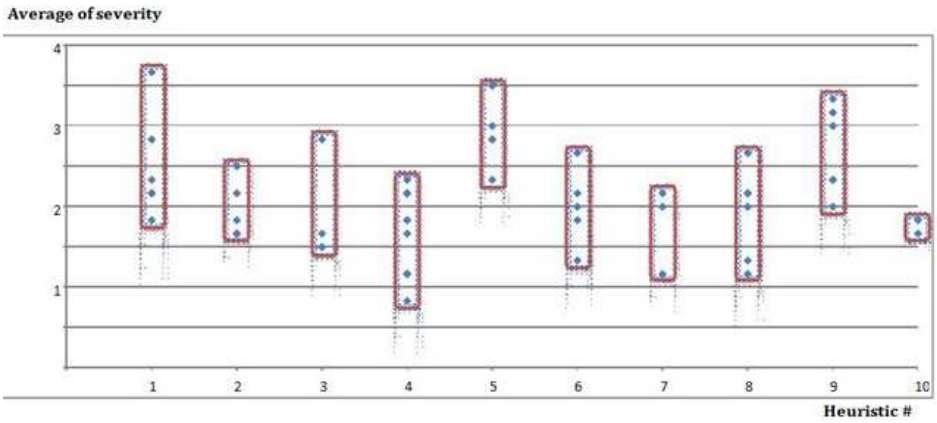


Figure 3.8: Average ranking of participants for each problem per heuristic

to problems related to heuristics 1, 5, and 9 (as shown in Figure 3.8), which are, respectively, *visibility of system status, error prevention*, and *helping users recognize, diagnose*, and *recover from errors*. Accordingly, they derived three development tasks from the problems: the first relates to working with the baton object, the second is adding indications to the sites' communication status, and the third relates to error prevention in case of communication problems.

Conclusions of the case study:
Different teams provided feedback on the contribution of the evaluation process using UEMan. Among other comments, teams mentioned the good collaboration between the team and the participants, the benefit of recognizing the new issues raised that had not been seen before, and the ability of UEMan to automate the results summary, enabling them to identify significant problems and define development tasks accordingly. As part of the evaluation study, we found that software team members engage in UCD management activities in a natural and intuitive manner. They can easily analyze experiments' results for their project and derive significant development tasks accordingly.

3.7 Related Work

Integrating UCD philosophy into software development processes, and specifically into the agile development approach, is not a new idea. Some earlier studies focus on applying several UCD techniques to agile development, while others focus on the benefits a particular technique can bring to agile development. However, the literature lacks a consolidated approach to cover the integration from all aspects.

Gulliksen et al. [45] proposed a definition of user-centered system design (UCSD) that is "*User-centered system design (UCSD) is a process focusing on usability throughout the entire development process and further throughout the system life-cycle*" [45] (p. 401). They identified 12 key principles for the adoption of user-centered development process from the existing theory and research. Gransson et al. in [40] and Gulliksen & Gransson in [44], defined a usability design process that integrates their UCSD approach in [45] with software development processes. Their defined process is iterative-in-nature and works with well-planned iterative and incremental development approaches, such as one provided for Rational Unified Process (RUP). Their process is divided into three phases: requirements analysis, growing software with iterative design, and deployment. Their proposed approach is not very well suited for agile development, in which emphasis is given to the working artifacts and small iterations.

Chamberlain et al. [12] described a framework for integrating UCD practices into agile development and provided a field study. They identified a set of five principles as being significant in integrating the two approaches–*user involvement* through a number of roles within team, *collaboration and culture* through daily basis communication and customer as an active member of the team, *prototyping* for giving rapid feedback to developers, *project lifecycle* for giving enough time to incorporate UCD practices, and *project management* for working both approaches together.

Constantine and Lockwood [16] described methods for usage-centered design and provided models to integrate usage-centered design practices [15] into software engineering practice while focusing on agile development approach. Pattern [94] proposed using of interaction design based on Constantine and Lockwood's usage centered design approach for Extreme Programming method of agile development.

Sy [113] provided an approach for incorporating usability in agile development. They added a cycle zero in the development for performing usability investigation activities. The approach is based on the concept that for each development cycle i there is need to perform usability testing of prototypes at least a cycle *i-1* and then the evaluated design is passed to developers in the iteration i for the implementation. The approach also suggests conducting the contextual inquiry for workflows at least two cycles ahead, and recommends the usability testing of the implemented working version.

Blomkvist [9] investigated the possible support of usability-enhancing activities in agile development, provided a general integrated model, and recommended a balanced integration for achieving benefits from both.

Hussain et al. in [61] proposed an approach of integrating UCD and Extreme Programming [7] development method based on evaluating the usability of user interfaces of developing application in small iterative steps. The ap-

proach is based on the concept of designing prototypes of user interfaces and evaluating them throughout the development process, thus evolving the design gradually.

Fox et al. provided a study [38] that researched participants experienced in combining these two approaches and concluded that there can be a common model from the existing models.

Ungar and White [121] provided a case study for merging the UCD and the agile development while using one-day design studio approach. The targeted agile development in their case study was Scrum [108] method.

There are also approaches with regard to integrating usability evaluation into different phases of the software development processes. Several of them suggest the integration of usability evaluation techniques in well-known processes while the others give their own version of the processes that accommodate usability evaluation according to their approach.

Juristo et al. [65] and Rafla et al. [99] deal with the implications of applying usability techniques during the requirements phase, and note the resulting problems and impact on the development.

Anderson et al. [4] as well as Ferr [35] suggest approaches for integrating usability techniques into the iterative nature of software development processes, while Ferr et al. [36] give a general usability process that can be applied by software teams after making slight variations to accommodate a *design-evaluate-redesign* cycle.

Many problems and challenges arise when software teams work with integrated usability evaluation development processes, and many suggestions have been presented to cope with these challenges and problems. For example, Lohmann et al. [75] highlight the key issues and design concerns that need to be considered while integrating users and their feedbacks during software development. Hellman et al. [50] suggest the changes that are needed for related development processes to support User experience (UX), specifically for the mobile industry. Based on the research of three case studies, Uldall-Espersen et al. [120] suggest that during software development, the usability of the product should be considered at the user interface level and in relation to organizational usability problems.

Few studies have investigated the impact of usability testing and evaluation on development. Alshamri et al. [3] explore the effects of task design on usability evaluation, since it can fundamentally influence the usability evaluation results, while Law [70] defines an approach to determine the effectiveness of user tests.

3.8 Summary and Further Directions

In this chapter, we have presented our three-fold framework for utilizing the benefits of user-centered design philosophy while developing software projects with agile approach. The proposed framework integrates the UCD philosophy at three levels, i.e., the process life-cycle level, the iteration level, and the development-environment level. We described in detail each of the integration level and explained the steps that are needed to achieve equal benefits from both approaches for enabling the development of high-quality and usable software products.

We suggested a set of attributes for selecting appropriate UCD methods during different phases of development. We suggested a life-cycle for performing UCD activities alongside agile development activities. We also provided suggestions for aligning UCD concepts, roles, and activities with the development iteration activities for maximum benefits. At the lower level, we provided the concept of *UCD management* for managing and automating UCD activities at the IDE level. This mechanism enables the software development team to monitor and control a continuous evaluation process, thus receiving ongoing user feedback while continuing development. The concept of UCD management at the IDE level is one of the main differences between our approach and the existing ones.

The framework recommendations and suggestions helped us to shape a set of requirements for creation of a tool UEMan, which enables the software development team to manage and automate the UCD activities at the IDE level alongside the development activities. We also provided two evaluation case studies of UEMan where in the first case study the UEMan was evaluated by using itself, while in the second case study six software teams used it to evaluate the software projects they developed. As part of the evaluation study, we found that software team members engage in UCD management activities in a natural and intuitive manner. They can easily analyze experiments' results for their project and derive significant tasks accordingly.

One of the main challenges we found in usability evaluation is to provide automatic analysis of the evaluation results so to effectuate maximum benefits from the automation process. One way to achieve this is by using task model-based usability evaluation approach in which formal task models are used to model user and system tasks. The created task models are then used as a mean for producing automatic analysis of the recorded users' and system data. Even though UEMan provides an effective framework to perform and automate the evaluation process, it lacks the ability to model user and system tasks and behavior. Thus, the experiment itself is not formalized in a way that can enable automatic analysis. To enable the automatic analysis of the evaluation experiment data, we need to define user and system tasks in a formal way

through task models and then to use these as a basis to record users and system data. We also need to provide the criteria for performing automatic analysis to highlight usability issues. This leads us toward two goals. The first one is to define a way to write the user and system tasks and behavior in a formal, unambiguous, and accurate way. The second one is to perform usability evaluation based on these task models and the automatic analysis of the recorded data through comparing it with the created task models.

In the forthcoming chapters, we deal with these two goals one by one. Firstly, we provide a way to write formal task models through a task modeling language, called TaMoGolog, which works on the foundations of the Golog-family [18, 19, 20, 73, 107] of high-level programming languages. Secondly, we provide a framework for performing usability evaluation through TaMoGolog-based task models. This framework also provides the automatic analysis of the recorded users' and system data by comparing the attached TaMoGolog-based task models. We also provide the realization of this framework through a tool, called TaMUlator, which works at the IDE level. This approach provides a way to highlight usability issues in an efficient and effective manner, and helps to produce software product with an adequate level of usability.

Chapter 4

Framework for Task Modeling Formalization

4.1 Motivation

This chapter is devoted to define a general conceptual framework for constructing task models from multi-view perspectives and to provide the definition of **TaMoGolog** (**Ta**sk **Mo**deling **Golog**) task modeling language that was created on the top of the foundations of Golog-family [18, 19, 20, 73, 107] of high-level programming languages. This work provides a foundation that is needed for our approach of automated task-model based usability evaluation at the development-environment level.

Our three-fold conceptual integration framework, defined in Chapter 3, emphasizes automating usability evaluation at the development-environment level to collect and analyze users and system behavior and to recognize usability flaws and errors in efficient and effective ways. Even though our tool UEMan, also described in Chapter 3, provides an effective framework to perform and automate the evaluation process, it lacks the ability to model user and system tasks. Thus, the experiment itself is not formalized in a way that can enable automatic analysis. We require to formally model, user and system tasks and behaviors for enabling automatic analysis on recorded data collected during evaluation experiments. This is achieved through structuring system activities, forming their relationship, and defining how users can achieve the desired goals through performing these set of activities. For this purpose, an expressive, dynamic, and well-defined (syntactically and semantically) formal task modeling language is required that gives us not only the facility to model user and system tasks and behaviors appropriately but also provides the way to construct task models with properties; such as, precondition axioms for tasks/actions, postcondition effects on system states, and inclusion of any

domain knowledge; that we required for automated analysis of the recorded data.

The existing task modeling languages in Human-Computer Interaction (HCI) area are at high abstract levels and most of them lack well-defined formal semantics. Another problem is the unavailability of characteristics, such as properly defined precondition axioms for actions or postcondition effects on system states, which we required for our usability evaluation approach. These reasons motivated us to provide a foundation, while utilizing the work done in Artificial Intelligence (AI) field, for task modeling in order to use it later for our task model-based usability evaluation framework. We first provide a conceptual framework that defines how to model properly task structures from different views perspectives. The use of task models from different views perspectives in evaluation experiment helps us to figure out usability issues at different layers. Then we provide a well-defined (syntactically and semantically) formal language TaMoGolog, on the foundations of Golog-family, that fills the gap found in existing task modeling frameworks by enabling precondition axioms and postcondition effects to tasks, option to define domain knowledge in task models, and by providing a rich set of operators for constructing complex system behavior in the resulting task models.

Following is brief summary of each section:

Section 4.2 introduces some brief preliminary background of situation calculus [102] and Golog-family of high-level languages. It also provides an overview of the Golog-family interpreter platform `P-INDIGOLOG` [105], in relation to our work here. The purpose is to provide an overview to those readers who are not much familiar with these topics.

Section 4.3 introduces our general framework for constructing task models at different abstraction levels. The framework suggests it at the conceptual-level through *framework-concepts* and at the representation-level through the definition of a *formal task modeling language*.

Section 4.4 describes details of *framework concepts* for providing a conceptual foundation to model and structure system activities from different abstraction levels. It provides definition and details of a set of concepts; i.e., *task, task type, view type, task model*, and *view model*; to understand properly the proposed framework suggestions for constructing task models from different views perspectives.

Section 4.5 provides the definition of TaMoGolog task modeling language. TaMoGolog uses constructs and semantics from the Golog-family in addition to few of its own defined constructs. This section first discusses the set of constructs and then provides syntax framework using predicate structure of situation calculus and the semantics. TaMoGolog evaluation semantics is also based on the Golog-family.

Section 4.6 discusses the framework for external nondeterministic constructs. GameGolog [20], a recent extension in the Golog-family, provides a way to model external agents' participation in making decisions for nondeterministic constructs. TaMoGolog uses GameGolog constructs and semantics at higher-level but differs slightly when defining framework theory using situation calculus at lower level. This section also provides the *Trans* and *Final* definition of TaMoGolog own defined external nondeterministic constructs. The support for making nondeterministic decision by external entities is very important from task modeling perspective, as it provides a way to model explicitly *end users*' participation during tasks execution that helps during usability evaluation to analyze users and system behavior separately, while making nondeterministic decisions.

Appendix A and Appendix B provide low-level implementation details of external nondeterministic constructs, that are discussed in Section 4.6, in two forms respectively: Golog-family based high-level program syntax, and Prolog-based syntax targeting IndiGolog [19] interpreter implementation platform `P-INDIGOLOG` [104, 105]. In Appendix C, we propose a *labeling framework* to implement external nondeterministic constructs through passing labels, representing program-parts, as parameters to external entities for making nondeterministic decisions.

4.2 Preliminary Background

This section provides a preliminary background about the Golog-family of high-level programming languages [18, 19, 20, 73, 107] that works on the theoretical foundation of situation calculus [102]. Our proposed task modeling language TaMoGolog is built on top of the Golog-family and uses situation calculus as a foundational framework, so we go through the situation calculus and the relevant languages in the Golog-family in the following sub-sections. The Golog-family of high-level programming languages aims to define and reason on complex actions and processes by finding executable atomic actions. The first language created in this family was Golog (alGOl in LOGic)[73], and since then, several extensions have been proposed where each extension further enhanced certain constructs or functionalities. The extensions, we are interested in, are: ConGolog (Concurrent Golog) [18], IndiGolog (incremental deterministic Golog) [19], GameGolog (Game Structure Golog) [20], and an extension that formalizes the BDI (Belief-Desire-Intention) action theory [107].

4.2.1 Situation Calculus

Situation Calculus (SitCalc) [102] is a first-order logical language with second-order features for representing and defining reasoning about the dynamic

world. In SitCalc, changes in the world are due to the execution of *actions* [18, 102]. As a result of executing these actions, the system moves from one *situation* to another *situation*. A *situation* is represented by a first-order formula that tells what is true in the current situation through the values of *fluents* (functional and relational). The system, in fact, moves from one situation to another when values of these fluents change due to the execution of some action α. Situation calculus defines a constant S_0 to denote the initial situation where there is no action has executed so far.

A *history* tells a sequence of execution of actions so far, and a point in the history represents the situation at that point. The binary term $do(\alpha, s)$ represents the next situation after executing the action α in the current situation s where the action may be parameterized; for example, *print*(*X,N*) may say to the printer for printing file X for N number of times. In SitCalc, actions are denoted as function symbols while situations as first-order terms. The abbreviation $do([a_1, a_2, ..., a_n], s)$ is used for the term $do(a_n, do(a_{n-1}, , do(a_1, s)))$ where the current situation is obtained from situation s while performing the actions in sequence as $(a_1, a_2, ., a_n)$ [18, 102].

The properties, called *fluents*, that hold in a situation are categorized into two kinds: *relational fluents*, which are predicates and take a situation term as their last argument, for example, $on(x, s)$ tells whether in the current situation the specific bulb x is on or not; and *functional fluents*, which are function symbols with additional situation argument, for example, "$balance(acc, s) = amount$" tells the balance of a specific account in situation s. The changes in the values of these fluents are specified through special set of axioms called *effect axioms* or *successor state axioms*, which specify how the action execution affects the value of these fluents. For each fluent F there is a *successor state axiom* that is of form [102]:

$$F(\vec{x}, do(\alpha, s)) \Leftrightarrow \Phi_F(\vec{x}, do(\alpha, s), s)$$

Where $\Phi_F(\vec{x}, do(\alpha, s), s)$ is a formula with free variables $\vec{x}$, α is an action, and s represents a situation. For example, following the successor state axioms for $on(x,s)$ brings the bulb in an *on* state if that is not its current state already and the action $pushButton(x)$ is executed by someone.

$$on(x, do(a, s)) \equiv \neg on(x, s) \wedge a = pushButton(x) \vee on(x, s) \wedge a \neq pushButton(x)$$

SitCalc also specifies how to define action in some domain and under what conditions the action is possible to be executed through set of precondition axioms. A special predicate $Poss(\alpha, s)$ [102] is provided that states the set of axioms that are needed to be true to execute the action. These action precondition axioms have the form:

$$Poss(\alpha, s) \Leftrightarrow \Pi_\alpha(s)$$

Where formula $\Pi_\alpha(s)$ defines under what conditions the action α can be executed. For example, the following precondition axiom for an action *deposit* says that the amount to be deposited in a bank account must be greater than zero.

$$Poss(deposit(acc, amount), s) \equiv amount > 0$$

To specify a domain application through situation calculus, we need action theories in the following form [18, 102]:

- Set of axioms describing the initial situation S_0.
- For each primitive action α, the set of precondition axioms characterizing $Poss(\alpha, s)$.
- For each fluent F, set of *successor state axioms* describing under what conditions the fluent $F(\vec{x}, do(\alpha, s))$ holds as function of what holds in situation s.
- Set of unique name axioms for the primitive actions in the domain theory.
- Set of foundational, domain independent axioms.

4.2.2 Golog-family of High-level Programming Languages

Golog-family provides a set of logic programming languages for specifying high-level programs to reason and control complex actions and processes execution. The first language was Golog [73] and there are many variants currently available that provide some extensions to the existing ones. The extensions on which we focus are: ConGolog [18] for handling concurrency in Golog, IndiGolog [19] for providing the look-ahead search operator, GameGolog [20] for handling nondeterministic choices by external agents, and IndiGolog adoption for BDI [107] to provide failure handling construct. Following are brief details of constructs provided by each of them:

Golog

Golog [73] provides a set of constructs to control the complex action execution. Table 4.1 summarizes those constructs and their meanings. The first construct is a that stands for a situation calculus action; the ϕ construct stands for a situation-suppressed formula thus evaluated in the current situation when the program reaches to it; while the sequence construct $\delta_1; \delta_2$ describes that after finishing certain action(s) successfully in program δ_1 the action(s) in program δ_2 are next in the queue.

Constructs	**Meaning**
a	primitive action
$\phi?$	wait for a condition
$(\delta_1; \delta_2)$	sequence
$(\delta_1 \mid \delta_2)$	nondeterministic choice between actions
$\pi x.\delta(x)$	nondeterministic choice of arguments
δ^*	nondeterministic iteration
proc $P(\vec{(x)})\delta$ **end**	procedure definition
$P(\vec{(\Theta)})$	procedure call

Table 4.1: Golog set of constructs [73]

Golog provides few nondeterministic constructs. The first one is the choice construct $(\delta_1 \mid \delta_2)$, where the system chooses nondeterministically either program δ_1 or program δ_2. The second construct is the choice of argument $\pi x.\delta(x)$, where the system nondeterministically chooses a variable x and binds the program δ for that variable x and then performs the program $\delta(x)$ accordingly. The last nondeterministic construct is the iteration δ^*, where the system performs program δ zero or more times nondeterministically.

Golog also provides a mechanism for procedure definition and procedure call through a second-order formula definition. A procedure is defined as:

proc $P(\vec{v})$ δ **end**

Where P is the name of the procedure, $\vec{v}$ are its formal parameters, and δ is the procedure body. $P(\vec{\Theta})$ is used for procedure call. Golog uses *call-by-value* approach for parameter passing and *lexical* (or *static*) scope as the scoping rule [18, 73].

ConGolog

ConGolog [18] (Concurrent Golog) is an extended version of Golog and provides the concurrent execution in resulting high-level programs that was missing in Golog. It models concurrent processes as interleaving of the primitive actions in the component processes. It includes all constructs of Golog in addition to the set of constructs listed in Table 4.2.

The (**if** ϕ **then** δ_1 **else** δ_2) and (**while** ϕ **do** δ) are the synchronized version of the usual *if-then-else* and *while-loop*. They are synchronized in the sense that testing the condition ϕ does not involve a new transition, instead testing the condition and executing the first action in the selected program is considered as unit transition. These can be achieved in Golog through $(\phi?; \delta_1 \mid \neg\phi?; \delta_2)$ and $([\phi; \delta]^*; \neg\phi?)$ but here the condition part and the program parts are considered as separate transitions.

Constructs	Meaning
if ϕ **then** δ_1 **else** δ_2	synchronized conditional
while ϕ **do** δ	synchronized loop
$\delta_1 \parallel \delta_2$	normal concurrency
$\delta_1 \rangle\rangle \delta_2$	concurrency with priority
$\delta^{\parallel}$	concurrent iteration
$< \phi \rightarrow \delta >$	interrupt

Table 4.2: ConGolog set of constructs [18]

ConGolog provides three constructs for handling concurrency. The first construct ($\delta_1 \parallel \delta_2$) is for normal concurrent execution (interpreted as interleaving) of two processes δ_1 and δ_2. A process may be in the *blocking* condition if it reaches to a state where the preconditions of one of its primitive actions are not met or there is a wait action $\phi?$, which hasn't yet reached true condition. In this case, the other program in the concurrency keeps its execution. The second construct ($\delta_1 \rangle\rangle \delta_2$) is for priority concurrency, where the first process δ_1 has a higher priority than the second process δ_2 and the second process can only start execution either the first finishes its execution or if it is in the blocking state. The concurrent iteration construct ($\delta^{\parallel}$) is like the nondeterministic iteration where instances of process δ are executed concurrently.

The interrupt construct $< \phi \rightarrow \delta >$ has a trigger condition ϕ and a body δ. When the interrupt gets control from the higher priority processes, it checks the condition ϕ and if it is true in that situation then it executes its body process δ. The interrupt gets ready to trigger again after the body part finishes execution.

IndiGolog

IndiGolog (incremental deterministic Golog) [39, 19, 106] provides to the programmer the control of planning/lookahead in ConGolog high-level programs. It supports online execution, sensing the environment, and execution monitoring. Golog and ConGolog provide the *off-line* execution, so before executing a program, they execute the entire program off-line and execute in reality only if it is successfully executed off-line; otherwise, they give failure error. IndiGolog provides a special construct, the *search operator* $\Sigma(\delta)$, which specifies that lookahead should be performed in such a way that the nondeterministic choice must be resolved so to guarantee the successful execution of the program.

The programmer can define in the program the search blocks in which the program will be executed through lookahead like as in the cases of Golog and ConGolog, while outside of these search blocks the nondeterministic choices

Constructs	Meaning
$[agt\ \rho_1 \mid \rho_2]$	nondeterministic branch
$[agt\ \pi x.\rho]$	nondeterministic choice of argument
$[agt\ \rho^*]$	nondeterministic iteration
$[agt\ \rho_1 \parallel \rho_2]$	concurrency

Table 4.3: GameGolog set of constructs [20]

are resolved externally from the program executor.

GameGolog

GameGolog (Game Structure ConGolog) [20] is a recent extension to ConGolog for specifying game structures in which nondeterministic choices can be made by some agent that has the control of the situation, where these decisions are also recorded in the situation. Table 4.3 shows the set of constructs of GameGolog.

The construct $[agt\ \rho_1 \mid \rho_2]$ specifies that the agent agt, which has control over the current situation, will choose whether to execute the program ρ_1 or ρ_2; the construct $[agt\ \pi x.\rho]$ specifies that the agent agt decides the binding for variable x and then the program ρ executes according to the binding; the construct $[agt\ \rho^*]$ specifies that the agent agt decides the moment to stop the iteration; while the construct $[agt\ \rho_1 \parallel \rho_2]$ states that the agent agt chooses how to interleave the execution of ρ_1 and ρ_2 during each step of execution.

Golog-BDI Adoption Approach

The **BDI** (**B**rief, **D**esire, **I**ntention) approach uses event happening for selecting a plan from the plan library and then places it into the intension base in order to commit the plan for achieving the goal [95]. There are agent-oriented programming languages and platforms that support this approach, such as AgentSpeak, Jason [100], and SPARK [81]. An approach described by Sardiña and Lespérance [107], let say it Golog-BDI adoption approach, provides a way to use IndiGolog for implementing BDI programming paradigm. We are interested in a construct provided by this approach for managing failure handling. The construct $(\delta_1 \rhd \delta_2)$ means that first the program δ_1 should be executed; and, in case of failure to finish it, the alternative program δ_2 will be executed leaving δ_1 where it was. Following is the *Trans* and *Final* definition from [107].

$$\begin{aligned} Trans([\delta_1 \rhd \delta_2], s, \delta', s') \equiv\ & (\exists \delta_1'.Trans(\delta_1, s, \delta_1', s') \wedge \delta' = \delta_1' \rhd \delta_2) \vee \\ & \neg\exists \delta_1', s''.Trans(\delta_1, s, \delta_1', s'') \wedge Trans(\delta_2, s, \delta', s'). \end{aligned}$$

$$Final([\delta_1 \rhd \delta_2], s) \equiv Final(\delta_1, s) \vee \neg\exists \delta_1', s''. Trans(\delta_1, s, \delta_1', s'') \wedge Final(\delta_2, s).$$

Execution Semantics of Golog-family-based High-level Program

The execution semantics of the Golog-family-based high-level program is expressed in terms of *transition semantics* (or *computation semantics*) [51, 85] based on single-step execution [18], which defines how a program evolves after executing an *action* in a step. It provides two predicates *Trans* and *Final* to specify program transitions.

- The predicate $Trans(\delta, s, \delta', s')$, given a program δ and a situation s, defines the semantics of how the program evolves to the remaining program δ' with the latest situation s'.
- The predicate $Final(\delta, s)$ specifies when the program δ can be considered as finishing successfully in the situation s.

The predicate *Trans* for programs for the constructs of above Golog-family languages without procedures is characterized by the following set of axioms [18, 19, 20, 73]:

1. Empty program:

$$Trans(nil, s, \delta', s') \equiv \texttt{false}.$$

2. Primitive actions:

$$Trans(a, s, \delta', s') \equiv Poss(a[s], s) \wedge \delta' = nil \wedge s' = do(a[s], s).$$

3. Golog wait/test action:

$$Trans(\phi?, s, \delta', s') \equiv \phi[s] \wedge \delta' = nil \wedge s' = s.$$

4. Golog sequence:

$$Trans(\delta_1; \delta_2, s, \delta', s') \equiv$$
$$\exists \gamma. \delta' = (\gamma; \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s').$$

5. Golog nondeterministic branch:

$$Trans(\delta_1 \mid \delta_2, s, \delta', s') \equiv Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s').$$

6. Golog nondeterministic choice of argument:

$$Trans(\pi v. \delta, s, \delta', s') \equiv \exists x. Trans(\delta_x^v, s, \delta', s').$$

7. Golog nondeterministic iteration:

$$Trans(\delta^*, s, \delta', s') \equiv \exists\gamma.(\delta' = \gamma; \delta^*) \wedge Trans(\delta, s, \gamma, s').$$

8. ConGolog synchronized condition

$$Trans(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s, \delta', s') \equiv \\ \phi[s] \wedge Trans(\delta_1, s, \delta', s') \vee \neg\phi[s] \wedge Trans(\delta_2, s, \delta', s').$$

9. ConGolog synchronized loop:

$$Trans(\textbf{while } \phi \textbf{ do } \delta, s, \delta', s') \equiv \\ \exists\gamma.(\delta' = \gamma; \textbf{while } \phi \textbf{ do } \delta) \wedge \phi[s] \wedge Trans(\delta, s, \gamma, s').$$

10. ConGolog concurrent execution:

$$Trans(\delta_1 \parallel \delta_2, s, \delta', s') \equiv \\ \exists\gamma.\delta' = (\gamma \parallel \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee \exists\gamma.\delta' = (\delta_1 \parallel \gamma) \wedge Trans(\delta_2, s, \gamma, s').$$

11. ConGolog prioritized concurrency:

$$Trans(\delta_1 \rangle\rangle \delta_2, s, \delta', s') \equiv \\ \exists\gamma.\delta' = (\gamma \parallel \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee \\ \exists\gamma.\delta' = (\delta_1 \parallel \gamma) \wedge Trans(\delta_2, s, \gamma, s') \wedge \neg\exists\varsigma, s''.Trans(\delta_1, s, \gamma, s'').$$

12. ConGolog concurrent iteration:

$$Trans(\delta^{\parallel}, s, \delta', s') \equiv \exists\gamma.\delta' = (\gamma \parallel \delta^{\parallel}) \wedge Trans(\delta, s, \gamma, s').$$

13. GameGolog nondeterministic branch:

$$Trans([agt\ \rho_1 \mid \rho_2], s, \rho', s') \equiv \\ s' = do(left(agt), s) \wedge \rho' = \rho_1 \vee s' = do(right(agt), s) \wedge \rho' = \rho_2.$$

14. GameGolog choice of argument:

$$Trans([agt\ \pi x.\rho], s, \rho', s') \equiv \exists x.s' = do(pick(agt, x), s) \wedge \rho' = \rho.$$

15. GameGolog iteration:

$$Trans([agt\ \rho^*], s, \rho', s') \equiv \\ s' = do(continue(agt), s) \wedge \rho' = \rho; [agt\ \rho^*] \vee s' = do(stop(agt), s) \wedge \rho' = \texttt{True?}.$$

16. GameGolog concurrency:

$$\begin{aligned} &Trans([agt\ \rho_1 \parallel \rho_2], s, \rho', s') \equiv \\ &\quad s' = do(left(agt), s) \wedge \rho' = [agt\ \rho_1 \langle \parallel \rho_2] \vee \\ &\quad s' = do(right(agt), s) \wedge \rho' = [agt\ \rho_1 \parallel \rangle \rho_2]. \end{aligned}$$

where:

$$\begin{aligned} Trans([agt\ \rho_1 \langle \parallel \rho_2], s, \rho', s') &\equiv Trans(\rho_1, s, \rho_1', s') \wedge \rho' = [agt\ \rho_1' \parallel \rho_2]. \\ Trans([agt\ \rho_1 \parallel \rangle \rho_2], s, \rho', s') &\equiv Trans(\rho_2, s, \rho_2', s') \wedge \rho' = [agt\ \rho_1 \parallel \rho_2']. \end{aligned}$$

The Golog-family uses *Trans* and *Final* predicates to provide a new definition to the $Do(\delta, s, s'')$ predicate which defines that given the initial situation s and a program δ, holds if possible to repeatedly single-step the program δ, obtaining a program δ' and a situation s' such that δ' can legally terminate in s' [18]. Formally, it can be written as:

$$Trans(\delta, s, s') \Leftrightarrow \exists \delta'. Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s').$$

Here $Trans^*$ is the reflexive transitive closure of *Trans* and it is possible to get more than one s' since programs based on ConGolog and extended versions can be nondeterministics.

The IndiGolog Interpreter Platform: P-INDIGOLOG

`P-INDIGOLOG` [105] platform, originally developed at University of Toronto based on `LeGolog` [72], is a *logic-programming implementation* of IndiGolog for the incremental execution of high-level Golog-like programs [19, 104]. The platform is developed in modular form and is extensible in order to deal with external applications/systems through providing suitable interfacing modules. The code of `P-INDIGOLOG` is mostly written in `Vanilla Prolog` but the overall architecture was developed in open source `SWE-Prolog` [126], which provides mechanism for interfacing with other programming languages such as Java or C, allows multi-thread application development, and supports socket communication and constraints solving [19]. The `P-INDIGOLOG` platform provides a way for the real execution of Golog-based high-level programs on concrete platforms or devices (e.g, a robot device), collects sensing output information, and detects the exogenous actions generated by external applications/systems. On the top level, the platform architecture is divided into six modules that are:

- **The Top-level Main Cycle:** The top-level main cycle works as *sense-think-act* loop [105] through three steps: first checks for exogenous actions occurred, then calculates the next program step, and finally executes *action* if the step involves any action. The platform updates the

history during the execution of actions. It provides predicate `indigo/2`, where `indigo(E, H)` states that a high-level program `E` is to be executed in history `H`.

- **The Language Semantics:** This module deals with the semantics of the language by using two main predicates `final/2` and `trans/2` that implement *Final* and *Trans* relations for giving single-step semantics to each program construct. When executing the actual program, this module first checks whether the current program is terminating in the current history through `final/2` predicate, and if so the top-level predicate `indigo/2` succeeds. Otherwise, in the case if the program evolves single step through `trans/4` predicate then the history remains same if no action execution is needed or is updated with the action execution and sensing information.

- **The Temporal Projector:** This module maintains the agent's belief about the world and evaluates a formula relative to a history. The module implements a realization of predicate `eval/3`, where `eval(+F, +H, ?B)` states the value `B` (either `true` or `false`) of formula `F` at some history `H`.

- **The Environment Manager (EM):** The EM is responsible to interact with external devices/application/systems through providing a complete interface. Through this interaction, it manages the execution of actions in real world, collects sensing outcome information, and detects the generation of any exogenous action by external entities. The EM manages action execution in three steps: firstly it decides which target external entity will execute the action, then orders the selected external entity to execute action, and finally collects the outcome sensing information. At top-level main cycle it provides predicate `exec/2`, where `exec(+A, -S)` states the execution of action `A` with returning sensing value `S`.

- **The Set of Device Managers:** Each external entity (device / application / system / environment / etc) that interacts with EM has a *device manager* (a piece of software) that understands the infrastructure of the external entity and knows how to talk with it. A device manager interacts with the environment manager through TCP/IP sockets and instructs the external entity to execute actions, and sends this back to the EM sensing information and any event occurring.

- **The Domain Application:** A domain application provides three key objectives for implementing a complete system: a domain axiomatization of dynamics of the world, a Golog-family based high-level agent program to show the agents' behaviors, and the information required to execute

the action in external entities; e.g., the external entity device manager, how the action actually is to be executed, the translation between high-level symbolic actions to low-level device manager representation, etc.

4.3 A Dynamic Framework for Multi-View Task Modeling

Task Modeling provides a way to model the structure of sets of activities and establish their relationships for defining how to achieve the desired objectives through performing these sets of activities. These sets of activities and their relationships are observed and obtained through different task analysis approaches. Over time, various task modeling techniques have evolved such as Hierarchical Task Analysis (HTA) [5] for breaking down low-level actions and defining their order to achieve some goal. Goals, Operations, Methods, and Selection rules (GOMS) [11] are helpful for modeling procedural knowledge, while Groupware Task Analysis (GTA) [122] is useful for modeling collaborative work. Different notations have been suggested for writing task models such as User Action Notation (UAN) [64] for in textual form, while ConcurTaskTrees (CCT) [92] and JSD (Jackson Structured Design) diagram [64] for graphical representation.

When we investigate previous work, we find lack of a generic framework to structure task models at different abstraction levels. Each task model structures the system activities and behavior from a certain view and a system can be described from several views at different abstraction levels. The differentiation in notations and providing details at different granularity from one technique to others, while modeling system activities from different viewpoints at different abstraction levels, creates obscurity in describing system structure and behavior accurately and unambiguously. Hence, it could lead to misunderstanding and confusion amongst the stakeholders. To overcome this lack, we propose a generic and dynamic framework for structuring and modeling system activities and behavior. This framework provides the solution at two levels: *at conceptual-level* through framework concepts for providing a conceptual foundation to create and structure task models at different abstraction levels; and *at representation-level* through a formal task modeling language with the ability to write dynamic and rich task models accurately and unambiguously. The framework can be used for multiple purposes, from system analysis to performing usability testing, due to its customizable and extensible nature. Such a framework, which provides a uniform way to create task models of system activities at different abstraction levels, gives an unambiguous understanding of created task models to all stakeholders of the system. The two main pillars of our proposed task modeling framework are:

- ***Framework Concepts:*** A set of special framework concepts that provides a conceptual foundation to create and structure task models at different abstraction levels from multi-view perspectives. The created task models provide a comprehensive picture of the external interactions, the internal structures and the behavioral responses from those perspectives. We provide the definition of a set of concepts; i.e., *task, task type, view-type, task-model,* and *view-model*; to understand properly how to create and structure task models at different abstraction levels from multi views perspectives.

- ***Task Modeling Formal Language:*** An expressive, dynamic, and well defined (syntactically and semantically) language is required for creating rich and powerful task models in order to model system activities and behavior from all perspectives. For this reason, the framework provides the definition of a formal language for task modeling that fills the gap existed in previous languages, which usually are unable to fully express the effects of executing tasks on the system and unable to cope with the complexity of the targeted system behavior. This definition provides a substantial foundation for defining complex system activities and behaviors in an appropriate, accurate, and unambiguous form, and provides designers/analysts with the freedom to express domain knowledge representation in rich task models. In the forthcoming Chapter 5, we use this language to create task models for using it in our model-based usability evaluation approach.

4.4 Framework Concepts

The behavioral response of a system depends on the external interaction and the internal structure of the system. To provide a comprehensive picture of this interaction, the internal structure and the behavioral response the system can be modeled at different abstraction levels from multi-view perspectives. For this, we suggest to model system activities from different views perspectives, where each view reflects system structure and behavior with some context at a specific abstraction level, which in turn provides a comprehensive picture of the system from all perspectives.

The motivation behind creating task models from multi-view perspectives at several abstraction levels is clear. Firstly, it decouples the complex structural and behavioral aspects as it suggests to model task structures and forming their relationships from different views perspectives where each view deals with only the related tasks interested to that view perspective. For example, a task model from the interaction viewpoint may highlight and emphasize only those system activities that interact with external entities. The following sub-

section provides details of these special framework concepts. Secondly, it gives a comprehensive picture of the scenario by modeling it from many perspectives for achieving the desired goals. In this case, it also provides the way to understand how the underlying business logic is accomplished from each view perspective.

Keeping the above idea, our task modeling framework provides the definition of five concepts: *task, task type, view type, task model*, and the *view model*.

- A *task* is an abstract entity that hides its internal functionality, performs one or more operations/actions, and provides an overall behavior for achieving some goal. It can be defined at different abstraction levels where a task at one abstraction level may contain a set of defined tasks at lower abstraction level.

- A *task type* represents the kind of behavior provided by the task at some abstraction level. For example, the tasks associated with the *user* type are executed by human users.

- A *view type* specifies the modeling of system activities and behavior, from a certain view perspective, for achieving some goal(s). The view guides what kind of task types and associated tasks will be participated in the resulting task model. For example, a *user-interaction* view may focus on modeling those system tasks and activities where the system interacts with external users.

- A *task model* provides the realization of system activities, their relations, and the behavior to achieve some goal(s) from a certain view perspective, and highlights the properties in that specific context.

- A *view model* combines a set of views where these views present task models from different abstraction levels for achieving the same set of goal(s). Each view model combines task models from those views that are special interest to it.

Our framework suggests and implements the task model structure based on hierarchical task decomposition method [5, 25], where the leaf nodes represent atomic or executing tasks while the parent nodes handles the structure of the task model. Figure 4.1 shows the relationship among above defined framework concepts. The background of the relationship between these concepts is inspired from the conceptual model of architectural description by IEEE-P1471 [62]. The figure shows that a system structure and behavior can be described by one or more view models where each view model contains task models from different views perspectives for achieving a particular set of goals. It highlights

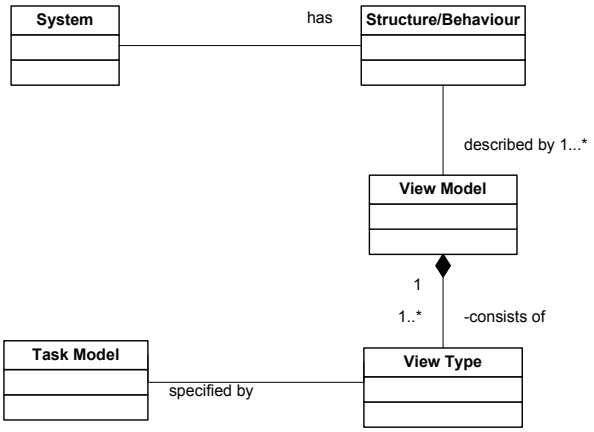


Figure 4.1: Relationship between framework *concepts*

that each task model, in fact, is a special representation of system activities and behavior from a certain view perspective at some abstraction level.

The *Task* Notion

In our framework, a *task* is an abstract entity that hides its internal functionality, performs one or more operations/actions, and provides an overall behavior for achieving some specific goal. It is possible to decompose this abstract entity at lower abstraction levels until we reach the point where it actually performs some action in an atomic way; like checking variable truth value, sending command. Generally, when we speak about an atomic task while residing at some abstraction level, we refer to the overall functionality it provides in atomic way at that level even though it can be decomposed into more atomic tasks at lower abstraction levels. For example, a *"print-page"* command at the application level can be decomposed into a series of atomic tasks at the device level; such as, entailing the request, checking whether printer is available, etc; in order to perform the required printing functionality. Therefore, we refer *"print-page"* task as performing in atomic way at the application level.

4.4.1 Task Types

A *task type* associated to a task is a representation of the behavioral or structural aspects of the task. It is also used to group tasks with similar kinds of abstraction. This categorization helps in constructing task models from some specific view perspective. In our framework, a task type represents not only the contextual behavior of the task but also highlights the kinds of properties

it may possess. At higher level, our task modeling framework differentiates task types into two main categories, i.e., the *basic* category and the *behavioral* category.

The Basic Category:

The *basic* category contain types based on hierarchical task decomposition approach in which leaves of a task model are atomic tasks, while the upper layers have complex tasks to handle the structure of the task model. In *basic* category, we distinguish tasks into three types: *unit*, *waiting*, and *composite*. In our framework, all tasks are supposed to belong to only one of these three basic types at a certain abstraction level. It is noteworthy that a task can belong to two different basic types at two different abstraction levels. For example, a *"print-page"* unit task at application level can be a composite task at device level. Unit and waiting tasks normally reside at leaf levels while composite tasks reside at upper layers in the task hierarchy. Here we give details of each basic type:

Unit Tasks: These tasks, at the current abstraction level, cannot be decomposed further and are considered to perform in atomic manner to achieve some particular goal. At lower abstraction level, they can be combination of more than one task. Generally, when modeling system activities, all basic level operations and functionalities are reflected through unit tasks. Unit tasks normally reside at leaf levels in task hierarchy.

Waiting Tasks: These are the tasks that wait either for a particular event to happen or for some set of conditions to be fulfilled. They do not perform any operation by themselves, but wait and as any event happens or the set of conditions is fulfilled these are supposed to complete execution. For example, a *waiting* task in a printer can be to wait for a printing request event or in the case of an application it can be to wait for a free printing request slot in the printer queue before sending print request. In our framework, these tasks also reside at leaf levels in task hierarchy.

Composite Tasks: These tasks have complex behavioral structure and are composed of sub-tasks (*unit*, *waiting*, or *composite*), so can be divided further at the current abstraction level. In fact, these tasks do not perform operations/actions by themselves; instead, these handle the task model structure to provide successful paths in order to reach the targeted goal. These use fluent values, conditions checking, temporal operators, etc; to determine which direction to follow next to achieve the targeted goal. For example, a *composite*

task *"payment"* may decide which payment method to follow from credit card, bank or cash methods through applying some condition and then the selected method will perform the payment transaction. It is interesting to note that at certain instances, at upper abstract level, these may act as unit tasks.

The Behavioral Category

The *behavioral* category contains those types that represent the behavioral characteristics and possible functionalities of tasks. A task can have more than one associated type in this category as it is possible that the behavioral characteristics of a task may overlap to more than one type. In a task model, normally, task to behavioral type association depends on the targeted view perspective as the task models are created accordingly. We provide the definition of a set of types in this category where each type possesses some characteristics. A new type can be added in this set according to the model requirements and the target environment. Here we give brief introduction of few behavioral types:

System Task-Type: This type groups those tasks that are performed by the system in question. Grouping tasks into system type is helpful to understand which actions and operations are executed by the system in question and which are by the external entities; especially, if the task model is being created from the internal and external view perspective.

Services Task-Type: This category contains those tasks that are services provided by external applications/systems. The system uses these to achieve some goals. For this, the system requests to external applications/systems to perform these tasks and then it is up to that external application/system how to perform these. Normally, from the system perspective they are considered as unit tasks, but at the external application/system level these can be any type of the basic category. As an example, a Book Store system may ask an external application to complete the payment transaction. From the Book Store system perspective, this represents a unit task, but from the external application perspective it may be a combination of more than one unit tasks where each task is responsible for some functionality.

User Task-Type: This type belongs to those tasks that are performed by human users either through direct interaction with the system or with some external application/system, generally in both environments: the electronic and the physical one. For example, writing credit card information in payment page, pressing some button in cars, or washing the tea kettle, etc. Through these tasks, users also give inputs to the system to achieve the desired goal(s).

Interaction Task-Type: This type belongs to those tasks that interact outside the system boundaries with certain external entities (application/systems or human users). The responsibility of these tasks is to handle the communications forward and backward to and from the external entities. These act as a channel between the system and the external entities. For example, if a system is interacting with some external system through a device manager and system is using *sendRequest* task of the device manager to send the request to the external system or a system is asking to some human user to select an item from a list through *selectItem* task.
This type has two sub-types: *service-interaction* and *user-interaction*. We divide external entities into two categories. The first one contains those external applications/systems that provide services to the system or ask for services from the system for themselves; e.g., a Book Store system may ask to an external application to complete the payment transaction. The second category contains those external applications/systems that act as a channel between the system and the human users. These provide an interface through which human users interact for receiving or sending inputs to the system; e.g., A Book Store system may use a Java applet to take user input for searching a specific book or selecting a book from a list. The *service-interaction* type deals with first category while the *user-interaction* type deals with the second.

- **Services-Interaction Task-Type:** This sub-category of the *interaction* type contains those tasks which are responsible to interact with external applications/systems where these applications/systems provide services to the system or ask for services from the system. When the external application/systems receive request through these tasks, they execute services tasks in order to achieve the requested goal. On the other side, the system executes some *system* task(s) when it receives request from the external application/systems.

- **User-Interaction Task-Type:** This is a sub-category of both the *interaction* type and the *user* type and contains the tasks whose targeted interactive external entity is the human users of the system. This interaction can be directly from the system or through some external application/system, but in all cases these external applications/systems act just a channel between the human user and the system. If the external application/system manipulates the user's input and afterwards sends it to the system then we consider the interaction is between the human user and that external application/system, while the interaction between that external application/system with the system in question in this case is considered as *service-interaction* type. The *user* type contains all those tasks that are performed by the human user of the system

irrespective of interaction, while the *user-interaction* is a sub-type and considers only those tasks that are performed by the human user but through interacting with the system such as filling a form.

Exogenous Tasks: This type belongs to those tasks that are generated by external applications/systems in order to provide some inputs or services to the system behind the scene, where normally the system does not have control over generation of these. These tasks are not direct part of the system structure and may be generated by the underlying operating system to provide basic services or by some external applications/systems.

So far, we have provided just few of the selected types and a new type can be added to the above set or any type can be redefined according to the targeted system and environment. For example, there can be some types that target some specific framework; like in the forthcoming sections, we define a new type *choice* that deals with the decisions of external entities about the selection of nondeterministic choice paths.

4.4.2 View Type

A *view type* presents task model from a specific view perspective at a particular abstraction level for achieving some targeted goal(s). The created task model provides the task execution scenario from that certain view perspective. For example, a view type from building the user interface perspective may emphasize on those tasks that interact or executed by human users. Each view type, in general, specifies the purpose, the level of abstraction, the focusing view perspective, and the interested task types to be used in modeling the task structure for achieving some specific goal(s). For the same set of goal(s), there can be multiple views where each view highlights the task structures and task relationships from a certain view perspective at some abstraction level.

Here, we provide examples of few view types; i.e., *system* view, *general-interactive* view, *services-interactive* view, *human-interactive* view, and *complete* view. It is possible to create a new view type or can redefine these view types in order to fit into the targeted environment.

System View: This view emphasizes on creating task models from the system perspective; hence, all other tasks, outside the system boundaries, are considered external to the system and are neglected or described at higher abstraction level. The purpose is to show how the system achieve some goal(s) through performing a set of activities.

General-Interactive View: This view emphasizes on constructing task models from interaction perspective in which those tasks are highlighted that interact with external entities. The purpose is to focus on how the system in question behaves while interacting with external entities for executing some operations, giving feedbacks, or taking inputs. This kind of task models are important during analysis and design phases of development as these focus more on the critical points of interactions.

- **Services-Interactive View:** This is sub-category of *general-interactive* view as it emphasizes the interaction with external applications/systems.
- **User-Interactive View:** This is also sub-category of *general-interactive* view, and the purpose is create task model from the perspective of how the users of the system perform tasks while interacting with the system to achieve the desired goal(s). The users' interaction with the system can be through some user interface provided by the system itself or by any external application/system.

Complete View: This view emphasizes on creating task models while keeping every kind of tasks and their details. The purpose is to get an overall picture of the task structure across the system boundaries for achieving some targeted goal(s).

4.4.3 View Models

A *view model* combines a set of views where these views present task models at several abstraction levels for achieving the same set of goal(s). Each view model combines task models from those views that are of special interest to it. Our framework leaves it to the responsible team to decide how many views are needed within a view model in order to get a comprehensive picture. For example, for user interface analysis and design, the team may combine *general-interactive*, *services-interactive*, and *user-interactive* views. The basic theme here is to organize and combine a set of task models from multiple perspectives for achieving the same targeted set of goals while providing a full understandability of these activities across different abstraction levels.

Here, we give a brief example of a view model for performing usability evaluation, let say it *usability-eval* view model. Our objective is to check the usability level of a targeted application through the performance of usability evaluation experiments by the end users of the application; where in each experiment, end users perform a number of tasks to achieve a certain set of targeted goals. In this case, the view model may contain task models created according to the *user-interactive* and *system* views; as it will help to understand, during evaluation, the users' interaction with the system and the

system response. This kind of view model is useful in model-based usability evaluation techniques where users perform a list of tasks for achieving the targeted set of goals while using the evaluating application, and then the automated evaluating tool records the users and the system activities and behavior and produce results after analyzing these with actual task models created according to the *user-interactive* and *system* views.

4.4.4 An Example for Task Modeling

In this subsection, we discuss an example and provide task models as suggested by our proposed task modeling framework. The system in this example is an "Online Book Store" that provides the facility to its users to buy book(s) online. The user can select a book either by browsing the book-catalog or by searching with some specific search criteria (such as author name, book title, keyword) or by enquiring the system For the list of recommended books, where the system recommendations are based on user's history and some context criteria. The system provides several other functionalities, such as user profiling, writing reviews, but here we concentrate on one of the main scenario, i.e., book purchasing.

Book-Purchasing Scenario: The user selects a book through one of three available options: by browsing the book-catalog, or by giving some specific search criteria, or by asking the system to recommend a book for the user. The user can repeat this until he/she finds the desired book. After that, the user adds the selected book into the shopping cart and can search again another book. The user can repeat this process (selecting a book and then adding it into the shopping cart) as many times as he/she wants. At any time, when the user has the desired list of books in the shopping cart, he/she can start checkout. When the user starts checkout, firstly, he/she fills the shipping details form (address, email, etc). The system generates the total price depends on VAT and the shipping address. The user then selects a payment method from three options; i.e., credit card, bank transfer, or voucher payment; and provides the detail of the selected payment option. The system then moves to the final part where it asks the user to check all the details. The user checks all details and finalizes the order. The system completes the payment transaction by communicating to the selected payment system (Credit Card Company, Bank, or Voucher System). After successful completion of the payment transaction, the system sends a confirmation email to the user's email account given previously in shipping details form. At any moment before completion of payment transaction, the user can select the cancelation of the order. In this case, the system asks the user for saving the shopping cart list in the user profile for future use or removing the list. The system acts as wished by the user.

Tasks and Task Types

The following is a possible list of tasks taken from the given scenario, where we represent each task in the form of *task-name* [*basic-type*, (*behavioral-types*)].

```
Book-Purchasing [composite, ()]
Book-Selection [composite, ()]
Catalog-Browsing [composite, (user, user-interaction)]
Catalog-Navigation [unit, (user, user-interaction)]
Picking-Book [unit, (user, user-interaction)]
Book-Searching [composite, ()]
Search-Criteria [unit, (user, user-interaction)]
Search-Results [unit, (system)]
Book-Suggestion [composite, ()]
Generate-Book-Suggestion [unit, (system)]
Adding-Book-To-Catalog [unit, (user, user-interaction)]
Making-Catalog [unit, (system)]
Checking-Out [composite,()]
Shipping-Details [unit, (user, user-interaction)]
Asking-Book-Suggestion[unit, (user-interaction)]
Price-Generation [unit, (system)]
Payment-Method [composite, (external-choice)]
Pay-By-Card [unit, (user, user-interaction)]
Pay-By-Bank [unit, (user, user-interaction)]
Pay-By-Voucher [unit, (user, user-interaction)]
Finalize-Order [composite,()]
Confirm-Order [composite, ()]
Confirm-Payment [unit, (user, user-interaction)]
Payment-Transaction [unit,(services, services-interaction)]
Transaction-Completed [waiting, (system)]
Transaction-Notification [unit, (exogenous)]
Send-Order-Confirmation [unit, (system)]
Cancel-Order [composite,(external-choice)]
Cancelation [unit, (system)]
Save-Details [unit,(system)]
```

We are not considering much lower abstraction level tasks. It is possible that some of these unit tasks at this abstraction level become composite tasks when we go to any lower abstraction level. For example, *Pay-By-Card* unit task may become a composite task that contains more than one unit tasks where each unit task deals with some specific kind of credit card. In the above list, most of the tasks are related to the *user* behavioral category, as these are performed by human users. Because these user-category tasks also provide a

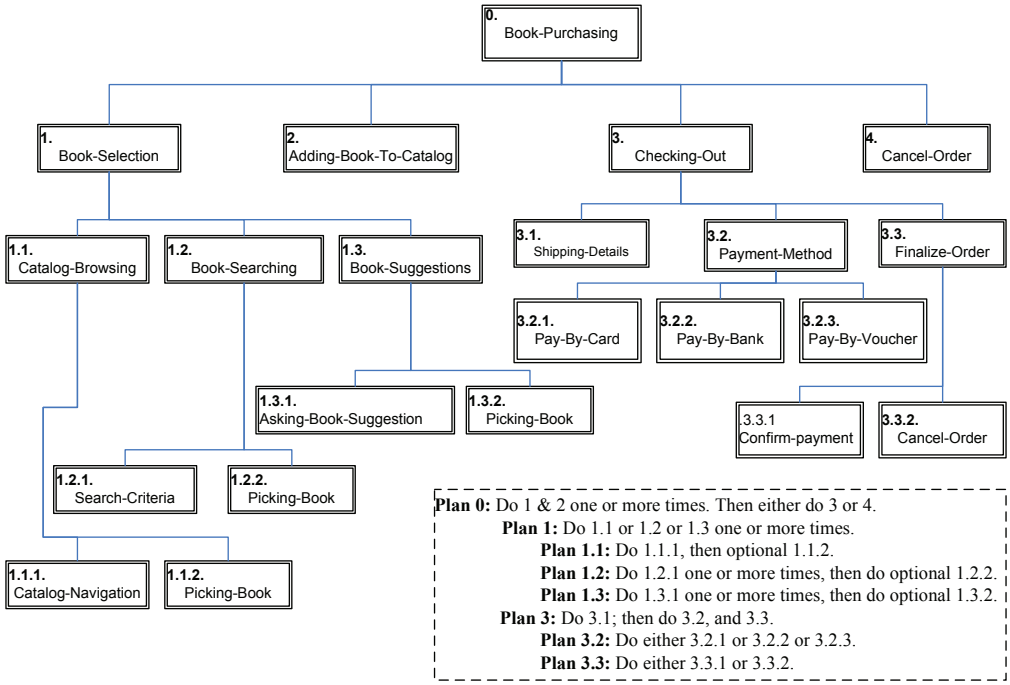


Figure 4.2: A task model of Book-Purchasing from *user-interactive* view perspective

channel between the user and the system, so these belong to *user-interaction* category. Few composite tasks, such as *Book-Searching*, have no behavioral type, as these contain mixture of sub-types. This kind of composite tasks can be associated to all the behavioral types of its sub-types. For example, *Checking-Out* task can be associated to *user*, *system*, and *services* types.

Task Models, View Types, and View Model

If we look into the given scenario, we notice that most of the tasks are are related to the categories where user interacts with the system , such as giving the search criteria or filling the shipping details, or where the system performs some operations, such as generating total price. Therefore, obviously we are interested in a task model from the *user-interactive* view. Figure 4.2 shows a task model, created in HTA [5, 25] form that describes the achieving of buying book goal from the user perspective in the *user-interactive* view. While Figure 4.3 shows the task model from the *complete* view perspective, which contains all the tasks involved in this scenario. So, we have:

View-Model: *book-purchasing*
View-Types: *user-interactive, complete*

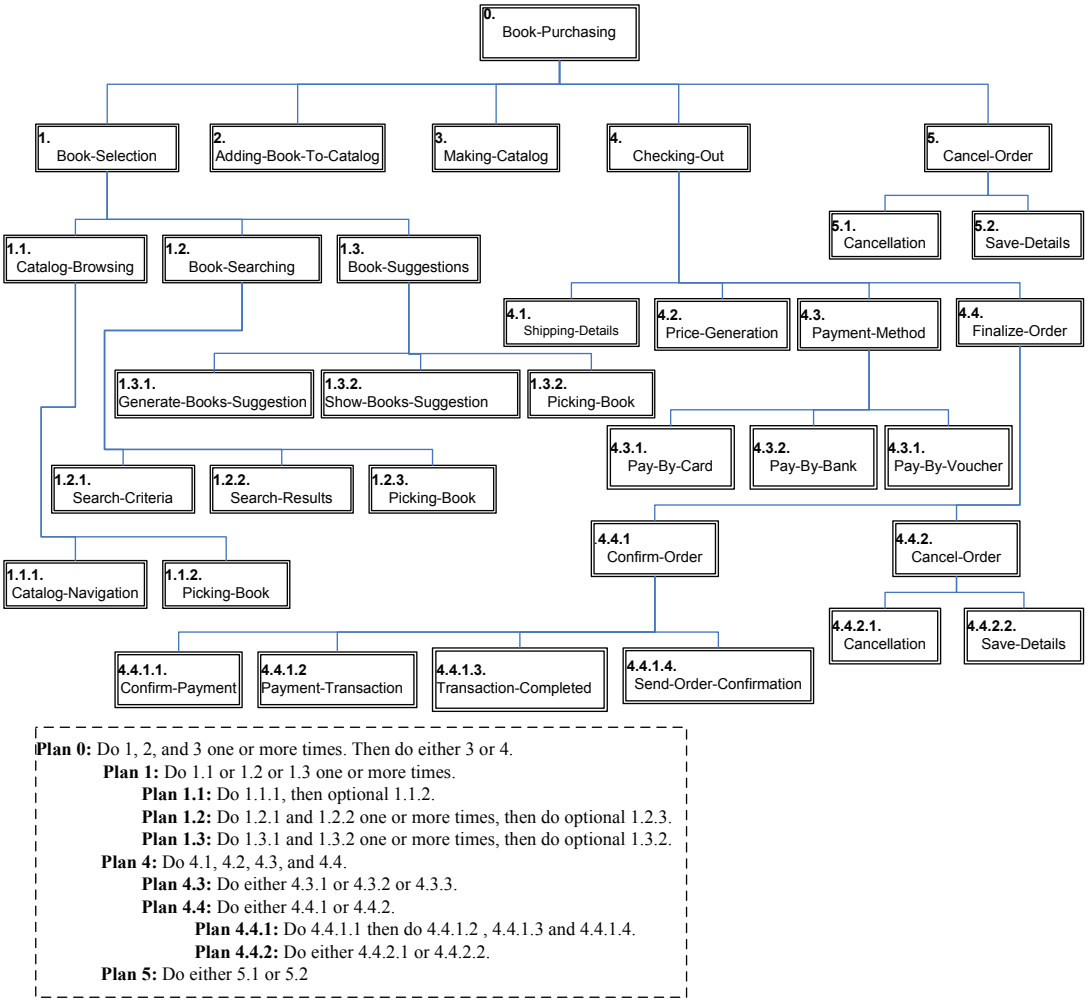


Figure 4.3: A task model of Book-Purchasing from *complete* view perspective

Task-Models: *book-purchasing-user, book-purchasing-complete*

This describes that in the *book-purchasing* view model, we have task models from two different views to show the achieving of the same goal of this view model, i.e., purchasing a book. We can have more task models in this view model, depending on the scenario's requirements. We keep this example in the forthcoming sections to write its parts in our formal task modeling language.

4.5 TaMoGolog – A Formal Task Modeling Language

An expressive, dynamic, and well-defined (syntactically and semantically) language is required for creating rich and powerful task models in order to model system activities and complex behavior accurately and unambiguously. We require a task modeling language that supports knowledge representation to provide domain knowledge in the created task models. Other than aforementioned qualities, we also want this language to be customizable or extendable in order to fit for creating task models for different purposes and environments; e.g., from system analysis to usability evaluation and from system design to support of task-centric environment.

The existing task modeling languages in Human-Computer Interaction (HCI) area are at high abstract levels and do not provide a comprehensive mechanism for representing system structure, activities, and behavior in task models from different viewpoints. Most of them lack well-defined formal semantics; hence, can create ambiguity and misunderstanding. Another problem is the unavailability of the domain knowledge representation, which results in task models being unable to describe system domain behavior properly.

On the other side, in Artificial Intelligence (AI) there exist approaches and languages; such as Hierarchical Task Network (HTN) [34, 33], STRIPS [37], high-level agent programming languages [18, 73, 19, 102]; to construct systems for autonomous agents in order to achieve the predefined set of goals. These have many benefits; like, formally well-defined syntax and semantics, precise definition of action rather than more abstract notion of task used in HCI, powerful set of constructs for handling complex system structure and behavior, facility to write knowledge representation, mature planning and inference techniques, etc. However, normally they ignore the role of human users' interaction with the system as they focus more towards autonomous agents. Sometimes, due to their complexity it is difficult to use them for creating task models for other purposes (e.g., for analysis or designing interface).

To fill the gap, found in previous task modeling languages in HCI, by utilizing the benefits of approaches in AI, we conclude that a better solution is to use some existing AI approach to make it suitable for constructing task models

as per our framework defined mode. This helps us to provide the definition of a task modeling language having the following abilities:

– a well-defined formal representation (syntactically and semantically)
– the ability to write dynamic, complex, and rich task models at different abstractions
– an expressive way to write the properties of tasks and task models as per defined by our task modeling framework
– representing unambiguously domain knowledge in task models
– involving human users and external applications/systems interaction in task models
– creation of executable task models
– customizable and extendable in order to suit for different purposes and environments

4.5.1 Task Modeling Golog (TaMoGolog)

Due to many of the characteristics, we fond the Golog-family [18, 19, 20, 73, 107] of high-level programming languages a worthy and suitable candidate for the foundation of a task modeling language. The reasons for selecting the Golog-family are:

– provides a rich foundation for reasoning about actions and complex processes
– expresses knowledge representation through situation calculus, thus enabling the facility to provide domain knowledge and the definition of system behavior in accurate and unambiguous way
– provides powerful sets of constructs useful for describing complex system structure and behaviors
– enables explicitly writing action precondition constraints and action execution effects on fluents
– provides a formal *evaluation semantics* and *transition semantics* for giving an unambiguous definition of the system behavior
– provides the facility to write high-level nondeterministic programs where a programmer can control the nondeterminism to control the execution
– extendable and customizable for different purposes

In spite of the above qualities, all previous versions, except a recent extension GameGolog [20], lack the mechanism for defining explicitly the external entities' role. Defining external entities role explicitly, especially the *human user* role, for making decision on nondeterminism is important for engineering task models in order to reflect how the external entities participate in per-

forming tasks or making decisions to achieve the desired goals. GameGolog provides an approach through which external entities, called *agents*, can decide the nondeterministic branches of the program during the execution. The approach provided by GameGolog is useful for describing external entities participation in task models for making nondeterministic branch decisions or for providing some input to the system.

On the basis of the above characteristics, we provide the definition of a task modeling language, called **TaMoGolog** (**Ta**sk **Mo**deling **Golog**), on the top of the foundations of the Golog-family of high-level programming languages. TaMoGolog gives a solid foundation for writing task models of complex system structures and behaviors in an appropriate, accurate, and unambiguous form; and provides the freedom to designers/analysts to express domain knowledge representation in the created task models. TaMoGolog tries to fill the gaps found in existing task modeling languages in HCI in order to write well-defined, rich, and unambiguous task models.

The following subsections describe TaMoGolog set of constructs, mainly obtained from Golog-family in addition to few own defined constructs, and the syntax and semantics of the language. While the forthcoming section (Section sec:ExtNDConstructs) discusses our approach for the framework of external nondeterministic constructs based on GameGolog approach. The resulting implementation of this framework is provided in Appendix A, Appendix B, and Appendix C.

4.5.2 TaMoGolog Set of Constructs

For defining complex and dynamic system structures and scenarios in task models, TaMoGolog provides a rich set of constructs mostly obtained from the Golog-family along with some additional constructs. It contains all the constructs of Golog [73] and ConGolog [18], the off-line search block construct from IndiGolog [19], the failure handling construct from Golog-BDI adoption approach [107], and the external nondeterministic choice constructs from GameGolog [20]. Along with these sets of constructs, it also provides the definition of some additional constructs on the basis of the constructs provided by GameGolog and Golog-BDI adoption. Table 4.4 shows the set of all these constructs (obtained from Golog-family along with TaMoGolog own defined constructs). In Table 4.4 υ represents a *unit* task, ϕ represents a condition or a conjunction of conditions, ϖ represents a *waiting* task (in TaMoGolog semantics, waits either for an event to happen or for some condition(s) to become true), and Γ represents a *composite* task that may have one or more sub-tasks.

In Table 4.4, TaMoGolog constructs are categorized into nine categories

Constructs	**Meaning**	**Origin**
	1 - Basic task category	
υ	*unit* task	GL
ϖ	*waiting* task	GL/TG
Γ	*composite* task	GL
	2 - Condition	
$\phi?$	waiting/testing action	GL
	3 - Sequence	
$\Gamma_1;\Gamma_2$	sequence	GL
	4 - Optional category	
$[\Gamma_1 \mid \Gamma_2]$	internal nondeterministic choice	GL
$[agt\,\Gamma_1 \mid \Gamma_2]$	external nondeterministic choice	GG
[**if** ϕ **then** Γ_1 **else** Γ_2]	*if-then-else* condition	CG
$[\pi x.\Gamma(x)]$	internal nondeterministic choice of arguments	GL
$[agt\,\pi x.\Gamma(x)]$	external nondeterministic choice of arguments	GG
	5 - Cycling/iteration category	
$[\Gamma]^*$	internal nondeterministic iteration	GL
$[agt\,\Gamma]^*$	external nondeterministic iteration	GG
[**while** ϕ **do** Γ]	conditional (*while-do*) loop	CG
	6 - Time-sharing category	
$[\Gamma_1 \parallel \Gamma_2]$	normal concurrency	CG
$[\Gamma_1 \rangle\rangle \Gamma_2]$	concurrency with priority	CG
$[\Gamma]^{\parallel}$	concurrent iteration	CG
$[agt\,\Gamma_1 \parallel \Gamma_2]$	external every-step decision concurrency	GG
$[agt\,\Gamma_1 \langle\rangle \Gamma_2]$	external selected priority concurrency	TG
$[agt\,\Gamma_1 \langle\mid\rangle \Gamma_2]$	external first-step decision concurrency	TG
$[agt\,\Gamma]^{\parallel}$	external selected concurrent iteration	TG
	7 - Off-line search	
$\Sigma(\Gamma)$	off-line search block	IG
	8 - Interrupt	
$< \phi \rightarrow \Gamma >$	interrupt call to a task	CG
	9 - Failure handling	
$[\Gamma_1 \rhd \Gamma_2]$	online alternative execution	GB
$[\Gamma_1 \rhd_\Sigma \Gamma_2]$	off-line alternative execution	TG

Table 4.4: TaMoGolog set of constructs

due to their nature of handling tasks structure; and the last column shows the origin of the construct where GL stands for Golog, CG stands for ConGolog, GG stands for GameGolog, IG stands for IndiGolog, GB stands for Golog-BDI adoption approach, and TG stands for TaMoGolog. In most cases, the meaning of constructs is the same as defined by the Golog-family discussed early in Section 4.2. In few cases, TaMoGolog provides its own customized definition of those obtained constructs. Following are brief description of these construct from the TaMoGolog perspective.

The *basic* task category represents three basic task types: *unit* task, *waiting* task, and *composite* task. TaMoGolog treats *waiting* task as a separate task where the start of execution, it waits for some particular event to happen or checks some set of conditions. As the particular event happens or the set of conditions becomes true, it finishes its execution. In fact, it does not perform any action by itself and does not change any fluent value.

The *waiting/testing condition* ($\phi?$) and *sequence* $[\Gamma_1;\Gamma_2]$ constructs are from Golog with the same definition; where *waiting/testing condition* also provides realization of a *waiting* task and *sequence* describes that the second task starts execution when the first task finishes its execution.

The *optional* category contains those constructs that deal with choice option between two or more tasks either nondeterministically (internally or externally) or through applying some condition.

- The *internal nondeterministic choice* construct $[\Gamma_1 \mid \Gamma_2]$, obtained from Golog, represents nondeterministic system choice where the system chooses any one of these in nondeterministic way. For example, the choice between two credit cards where the system decides nondeterministically which card to be used for the payment transaction.

- The *external nondeterministic choice* construct $[agt\,\Gamma_1 \mid \Gamma_2]$, from GameGolog, involves an external agent for making the nondeterministic choice and then the system executes the selected task. This is important from task modeling perspective as it involves human user or external application/system in making the choice decision. In this case, the *agt* is the external entity (human user or some external application/system) that makes this choice decision. TaMoGolog assumes that there will be some communication channel between the system and the external entity through which the system asks and receives this decision. If we take the previous example of payment by credit cards then this time the selection of the card will be done by some external entity at run time.

- The construct [**if** ϕ **then** Γ_1 **else** Γ_2] is a synchronized *if-then-else* construct from ConGolog where the condition checking is considered as part of the first action execution of the selected task.

- The *internal nondeterministic choice of argument* construct $[\pi x.\Gamma(x)]$ from Golog and *external nondeterministic choice of argument* construct $[agt\ \pi x.\Gamma(x)]$ from GameGolog have same definitions as described early in Section 4.2.2; that is, the system or agent chooses binding of variable x and the task Γ is executed after binding it to this variable x.

The *cycling/iteration* category deals with repetition of same task (or set of tasks) zero or more times either nondeterministically or through some condition checking.

- The *internal nondeterministic iteration* construct $[\Gamma]^*$, from Golog, represents the execution of task Γ zero or more times nondeterministically decided by the system. It is useful in those places where there is need to execute the same task more than once.

- The *external nondeterministic iteration* construct $[agt\ \Gamma]^*$, from GameGolog, is the same as the previous one, the only difference is that the external entity *agt* decides how many times the task will be executed. Each time after finishing execution, the system asks to the external entity whether to execute it again or not; e.g., [*human-user* `addBookToCart`]* says that the external entity *human-user* can add book to shopping cart as many times as it wants.

- *The conditional loop* construct [**while** ϕ **do** Γ] from ConGolog is a synchronized version of usual *while-do* loop where the task (or set of tasks) repeats execution till the condition is true. For example, [**while** *requestInQueue* **do** *printRequest*] says that till there is some request for printing in the queue, the printer keeps printing the request.

The *time-sharing* category provides many constructs to deal with concurrency. As the Golog-family supports the interleaving form of concurrency [18]; hence, TaMoGolog also assumes the same support of concurrency where tasks may interleave through *unit* or *waiting* tasks. It assumes that at one time only one unit or waiting task keeps execution. This category provides seven constructs where the first three are from ConGolog, fourth one is from GameGolog, while the last three are provided by TaMoGolog on the foundations of GameGolog approach.

- The construct $[\Gamma_1 \parallel \Gamma_2]$ deals with normal form of interleaving concurrency where the system nondeterministically takes a *unit* or *waiting* task

(action) from either of tasks and then executes it. The system repeats this till one of them finishes execution and then the remaining task is executed as normal.

- The construct $[\Gamma_1 \rangle\rangle \Gamma_2]$ provides priority form of concurrency where the task Γ_1 has more priority over task Γ_2. Task Γ_2 only starts execution either after completion of task Γ_1 or if the task Γ_1 is in some blocking state. This is useful where there is need to show different priority between a set of tasks. For example, in the case of $[\mathit{fireAlarm} \rangle\rangle \Gamma_1 \rangle\rangle \Gamma_2 \rangle\rangle$ ... $\rangle\rangle \Gamma_n]$ the task $\mathit{fireAlarm}$ has higher priority than any other task.

- The construct $[\Gamma]^{\|}$ represents the concurrent iteration of task, which means the same task participates in concurrency if the system selects it for more than once. So, it acts like $[nil \mid \Gamma \mid [\Gamma \parallel \Gamma] \mid [\Gamma \parallel \Gamma \parallel \Gamma] \mid ...]$. This construct is useful where there is need to create concurrent instances of the same task to handle the same thing within different instances; such as creating sessions for different users at the server side.

- The *every-step external decision concurrency* construct $[agt\ \Gamma_1 \parallel \Gamma_2]$ from GameGolog asks to the external entity at every step of interleaving to decide which task will be the next one amongst the concurrent tasks to perform some action. The system keeps asking this from external entity till only one task remain.

The following three constructs in *time-sharing* category are TaMoGolog own defined constructs on the foundations of GameGolog approach.

- The *external selected priority concurrency* construct $[agt\ \Gamma_1 \langle\rangle \Gamma_2]$ asks to the external entity to decide the priority amongst given tasks, and then the system executes tasks in concurrency with that priority level.

- The *first-step external decision concurrency* construct $[agt\ \Gamma_1 \langle|\rangle \Gamma_2]$ is different from the *every-step external decision* construct in the regard that in this case, the external entity decides the task to perform the first step and then the remaining parts execute with normal concurrency. So, if the external entity chooses Γ_1 then the first action will be from this task. If the selected task is in blocking state then the other task will not perform any action till the selected task finishes the first action; and after that remaining execution will be with normal form of concurrency and will act like $[\Gamma_1' \parallel \Gamma_2]$ or $[\Gamma_1 \parallel \Gamma_2']$ where Γ_1' or Γ_2' are remaining parts in case of selection. This is useful in those cases where other tasks depend on the starting of any particular task in concurrency.

- The *external selected concurrent iteration* construct $[agt\ \Gamma]^{\|}$ represents the nondeterministic concurrent iteration but here the external entity

decides rather than system that how many instances of the task will participate in the concurrent iteration.

The *off-line search* construct $\Sigma(\Gamma)$, from IndiGolog, deals with off line execution. TaMoGolog follows IndiGolog for the normal execution as online execution; especially, those parts that deal with external entity decisions because they cannot be executed off-line due to the nature of making decision at run time. So, TaMoGolog uses IndiGolog off-line search operator $\Sigma(\Gamma)$ to provide off-line search block, which indicates that the task(s) inside this block will be executed after checking if it is possible to finish these successfully. If it is not possible to finish any possible path successfully then the system does not start its execution. This is very useful in many cases where there is need to check whether the task(s) will be able to finish successfully before starting execution. For example, if we have a scenario with three tasks that are: the user selects a book, the system automatically transfers money from user's bank account to the credit card, and then the system completes the payment transaction automatically. It is possible that the payment system may not accept the given credit card and as the first two tasks were executed successfully before reaching to this point, so the money has already been transferred to the credit card; hence can not be back to the bank account. In these kinds of situations, it is better to put these tasks into off-line search block where they are checked firstly and if any successful path exists then the system will follow that particular path. The IndiGolog uses regression formulas to determine off-line searching.

TaMoGolog also uses ConGolog interrupt construct $< \phi \rightarrow \Gamma >$ for defining interrupt related conditions and tasks. It uses the same concept as provided by the ConGolog for interrupts; that is, if the interrupt gets control from higher priority processes when the condition is true then the task starts its execution; otherwise, if the condition never becomes true then the task is never executed. Most of the task modeling languages in HCI ignore this important factor. Through this approach, it is feasible to model them appropriately in the resulting task model; e.g., $< (temp > 20) \longrightarrow \mathit{startAlarm} >$ says that when the temperature is above 20 then the *startAlarm* task will start its execution.

The *failure-handling* category provides constructs deal with alternative path(s) in the case if the current path fails to finish for achieving the desired goal(s). In normal online execution, if the path (here path means a task or set of tasks for achieving a particular goal) fails to finish then it is considered the failure of the targeted goal, but Golog-BDI adoption approach [107] provides a construct to deal with it. TaMoGolog uses online alternative execution con-

struct from Golog-BDI adoption approach and also provides another construct for off-line failure-handling.

- The construct $[\Gamma_1 \rhd \Gamma_2]$, taken from Golog-BDI adoption, is for online failure-handling and defines that the task Γ_1 starts its execution and if it finishes successfully then the task Γ_2 will not be executed at all; otherwise, if Γ_1 fails anywhere during its execution then Γ_2 will start its execution from that point. This is useful for providing alternative solution(s) if the current solution fails for achieving the desired goal. TaMoGolog assumes online execution in this form, so there is no back tracking. The system leaves the failed task as it is and starts the alternative solution. For example; [*payByCard* $\rhd$ *payByBank*] tells that if payment transaction by card fails to finish, may be due to insufficient funds, then the system will complete the payment transaction through bank account.

- TaMoGolog provides *off-line alternative execution* construct $[\Gamma_1 \rhd_\Sigma \Gamma_2]$. This construct checks off-line while using IndiGolog search construct (Σ) before performing task Γ_1 and if Γ_1 is able to finish successfully then Γ_1 will start execution; otherwise, Γ_1 will be dropped and the alternative solution will be executed. TaMoGolog uses two variants of off-line support: *partially off-line* in which only the task Γ_1 is checked through search operator, and fully off-line in which task Γ_2 is also checked through search operator. This solution is applicable only at those places where there is not any online block in that part of the task model. If we take the previous example with this construct then [*payByCard* $\rhd_{\Sigma p}$ *payByBank*] with partial online means that before going forward to start *payByCard* task, the system first checks whether it is possible to complete payment transaction by this task, and if so, then it starts executing this task; otherwise, it drops *payByCard* and completes the payment transaction through *payByBank*. Whereas, [*payByCard* $\rhd_{\Sigma f}$ *payByBank*] with fully online means that *payByBank* will also be checked from IndiGolog search operator before execution.

4.5.3 TaMoGolog Syntax and Semantics

This section provides TaMoGolog syntax framework and its semantics for constructing task models. TaMoGolog uses situation calculus [102] to provide predicates structure for defining different properties of the task model. It also uses Golog procedure definition and the set of constructs, provided in Table 4.4, to define tasks structure and their relationships in a task model. Using predicate framework of situation calculus, it is possible to customize or to

extend TaMoGolog given syntax structure to accommodate it to any targeted environment.

On the semantics side, at the top level for tasks execution, TaMoGolog uses *transition semantics* for single-step execution and *evaluation semantics* for complete execution [51, 85, 18, 102], same as in the Golog-family, and assumes *high-level program execution* [19] where program (in our case, a task model) evolves after executing an action (in our case, a *unit* task, *waiting* task, condition checking, or any task performed in atomic way) in a step.
Following are the details of the TaMoGolog syntax and its semantics:

Task Model Framework Structure

TaMoGolog provides a set of predicates to denote the various objects of interests for the task modeling framework. These domain independent predicates are:

ViewModel(n): n is a view model
ViewType(p): p is a view type
TaskModel(m): m is a task model
ModelContainer(n,p): the view model n contains view type p
ViewModelTask(p,m): the view type p is realized by a task model m

The following definitions of above last two predicates define relations between framework concepts, as described early in Section 4.4.

$$ModelContainer(n,p) \stackrel{\text{def}}{=} ViewModel(n) \wedge ViewType(p)$$
$$ViewModelTask(p,m) \stackrel{\text{def}}{=} ViewType(p) \wedge TaskModel(m)$$

The first defines the relation between *view model* and *view type*, while the second one defines relation between *view type* and *task model*.

Continue Example: For the example described in Section 4.4.4, we provide:

```
1.  ViewModel(book-purchasing).
2.  ViewType(user-interactive).
3.  ViewType(complete).
4.  TaskModel(book-purchasing-user).
5.  TaskModel(book-purchasing-complete).
6.  ModelContainer(book-purchasing, user-interactive).
7.  ModelContainer(book-purchasing, complete).
8.  ViewModelTask(user-interactive, book-purchasing-user).
9.  ViewModelTask(complete, book-purchasing-complete).
```

Task and Task Types

TaMoGolog provides a set of domain independent predicates for defining tasks and categorizing their types. TaMoGolog categorizes task, as defined by our framework, into three basic types; i.e., *unit*, *waiting*, and *composite*; and provides predicates for each of these.

UnitTask(υ): υ is a *unit* task
WaitingTask(ϖ): ϖ is a *waiting* task
CompositeTask(Γ): Γ is a *composite* task

For specifying any other task in the task model, whose exact basic category is not yet known, TaMoGolog provides predicate:

Task(α): α is a task

It is still possible to associate this task to some behavioral type. This is useful where there is not full knowledge of basic type or where someone wants to extend the basic categories through defining a new category of tasks.

TaMoGolog provides two predicates for defining task types, other than the above basic types, and to associate tasks to those defined types:

Type(t): t is a task type, normally represents *behavioral* category
TaskType(α, t): α is a task of type t

A task that is associated with some type firstly must be defined as a task of either some *basic* type or from the predicate *Task*. Formally:

$$\begin{aligned}&\mathit{TaskType}(\alpha,\, t) \stackrel{\text{def}}{=}\\ &\quad \exists\alpha'.\{(\mathit{Task}(\alpha') \vee \mathit{UnitTask}(\alpha') \vee \mathit{WaitingTask}(\alpha') \vee \mathit{CompositeTask}(\alpha'))\wedge\\ &\quad \alpha = \alpha'\} \wedge \exists t'.\{\mathit{Type}(t') \wedge t = t'\}\end{aligned}$$

Continue Example: Few tasks from our continuing example (Section 4.4.4):

```
1.  UnitTask(Adding-Book-To-Catalog).
2.  UnitTask(Payment-Transaction).
3.  WaitingTask(Transaction-Completed).
4.  CompositeTask(Payment-Method).
5.  Type(services).
6.  Type(system).
7.  Type(user).
8.  Type(user-interaction).
9.  TaskType(Adding-Book-To-Catalog, user).
10. TaskType(Adding-Book-To-Catalog, user-interaction).
11. TaskType(Payment-Transaction, services).
12. TaskType(Transaction-Completed, system).
```

Composite Task Definition

TaMoGolog uses Golog procedure definition for providing composite task definition. The following is procedure definition taken from Golog [73] where P is the name of procedure, $\vec{x}$ is parameter list, and δ is a program providing the structure of procedure:

proc $P(\vec{x})\delta$ **end**

At parent-level, the composite task is called through Golog procedure call definition; that is, by the name of the procedure $\Gamma(\vec{\Theta})$ and the parameter passing is through *call-by-value* [18, 73].
TaMoGolog assumes that for each composite task, there will be a procedure definition against it to define its structure.

$$\mathit{CompositeTask}(\Gamma(\vec{x})) \Leftrightarrow \exists P(\vec{y}), \delta, n.\{\textbf{proc}\ P(\vec{y})\delta\ \textbf{end} \wedge \Gamma = P \wedge |_{i=1}^{n}\ (x_i = y_i)\}$$

Where $|_{i=1}^{n}\ (x_i = y_i)$ stands for the conjunction of $(x_1 = y_1) \wedge (x_2 = y_2) \wedge ... \wedge (x_n = y_n)$ and provides substitution in the form $P_{\vec{x}}^{\vec{y}}$. Note that we use this same definition following in other places.

TaMoGolog restricts that each composite task can have only one procedure definition against it in a task model.

$$\mathit{CompositeTask}(\Gamma(\vec{x})) \wedge (\textbf{proc}\ P(\vec{y})\delta\ \textbf{end}) \wedge (\textbf{proc}\ Q(\vec{z})\delta\ \textbf{end}) \wedge (\Gamma = P) \wedge (\Gamma = Q) \wedge \exists n.\ |_{i=1}^{n}\ (x_i = y_i \wedge x_i = z_i) \Rightarrow P(\vec{y}) = Q(\vec{z})$$

Which specifies that only one procedure definition is possible for each composite task with the same parameter list. It is possible to have two composite tasks in a task model with same name but with different parameter list. In that case, these are treated as two separate composite tasks.

Continue Example: Book-Purchasing and Payment-Method composite tasks definition from the Book-Purchasing complete-view task model (Section 4.4.4):

```
1.  proc Book-Puchasing
2.     [Human-User  (Book-Selection;
3.            Adding-Book-To-Catalog; Making-Catalog)]*;
4.     [Human-User (Checking-Out | Cancel-Order)]
5.  end
6.
7.  proc PaymentMethod
8.     [Human-User (Pay-By-Card|Pay-By-Bank|Pay-By-Voucher)]
9.  end
```

Top Hierarchy

The predicate *TaskModel*(m) with name m provides the top hierarchy of the tasks structure in a task model. The realization is defined by a composite task with the same name as the task model name. This composite task resides at the top level in the task hierarchy of the task model. Formally:

$$\mathit{TaskModel}(m(\vec{x})) \Rightarrow \exists n, \Gamma.\{\mathit{CompositeTask}(\Gamma(\vec{y})) \wedge m = \Gamma \wedge |_{i=1}^{n} (x_i = y_i)\}$$

$$\mathit{TaskModel}(m(\vec{x})) \wedge \mathit{CompositeTask}(\Gamma(\vec{y})) \wedge \mathit{CompositeTask}(\Gamma'(\vec{z})) \wedge m = \Gamma \wedge m = \Gamma' \wedge \exists n.\ |_{i=1}^{n} (x_i = y_i \wedge x_i = z_i) \Rightarrow \Gamma(\vec{y}) = \Gamma'(\vec{z})$$

The above statements defines that there exists only one composite task with the same name and parameter list as the name and parameter list of the task model.

Waiting Task Definition

TaMoGolog treats *waiting* task as one from the basic task category. The Golog-family provides a waiting/test action as a situation calculus formula where the system checks whether it meets the condition or not and as the condition becomes true the system moves to the new state, which in fact remains the same. The *Trans* definition of waiting/test action from [73] is given in Section 4.2.2.

TaMoGolog uses the same semantics for the waiting task, but provides a way to model them as a basic type category in the task model through *WaitingTask* predicate. This is useful when there is need to model this explicitly as a separate task in a task model. In this case, this is modeled through *WaitingTask* predicate; otherwise, this can also be modeled as part of the composite task structure rather than defining it separately.

Each waiting task, defined through *WaitingTask* predicate, will have a Golog procedure definition that will contain the related Golog waiting/test action through a situation calculus formula. Formally:

$$\mathit{WaitingTask}(\varpi(\vec{x})) \Leftrightarrow \exists P(\vec{y}), \phi_{\varpi}, n.\{\textbf{proc}\ P(\vec{y})\phi_{\varpi}?\ \textbf{end} \wedge \varpi = P \wedge |_{i=1}^{n} (x_i = y_i)\}$$

$$\mathit{WaitingTask}(\varpi(\vec{x})) \wedge (\textbf{proc}\ P(\vec{y})\phi_{\varpi}?\ \textbf{end}) \wedge (\textbf{proc}\ P(\vec{z})\phi_{\varpi}?\ \textbf{end}) \wedge (\varpi = P) \wedge (\varpi = Q) \wedge \exists n.\ |_{i=1}^{n} (x_i = y_i \wedge x_i = z_i) \Rightarrow P(\vec{y}) = Q(\vec{z})$$

External Entities

TaMoGolog defines external entities as those applications, systems, or human users outside of the system boundaries, which interact with the system in order to execute tasks requested by the system, generate exogenous actions, or

participate in making nondeterministic decisions. TaMoGolog provides following predicates to denote the various objects of interests related to external entities:

Agent(*agt*): *agt* is an external entity (called as agent) that interacts with the system
AgtEnv(*agt, s*): *agt* has control in situation s for making decision in external nondeterministic constructs
Responsible(*agt,* α): external entity *agt* is responsible for executing task α

Here, the predicate *Responsible* defines the relation between the external entity and the task in a task model for which external entity is responsible. Formally:

$Responsible(agt,\ \alpha) \stackrel{\text{def}}{=} \exists agt'.\{Agent(agt') \wedge agt = agt'\} \wedge \exists \alpha'.\{(Task(\alpha') \vee UnitTask(\alpha') \vee WaitingTask(\alpha') \vee CompositeTask(\alpha')) \wedge \alpha = \alpha'\}$

The frameworks for managing and implementing execution of actions/tasks by external entities are normally domain dependent and vary from one domain to another. An example of such a framework is provided by de Leoni et al. [22, 21]. Their framework provides and implements an approach for adaptive process management systems where the tasks are executed by external services. The framework for how the external entities make decision for external nondeterministic constructs will be discussed in the forthcoming Section 4.6. Here, we will not divulge into the details of how to define these at low-level; but, the frameworks can be defined easily by using situation calculus to describe low-level details targeting some particular domain and environment.

Continue Example: Agents and some of their responsible tasks from the Book-Purchasing complete-view task model (Section 4.4.4):

```
1. Agent(Human-User).
2. Agent(CreditCardSystem).
3. Agent(Bank).
4. Agent(VoucherSystem).
5. Responsible(Human-User, Search-Criteria).
6. Responsible(Human-User, Shipping-Details).
7. Responsible(CreditCardSystem, Payment-Transaction).
8. Responsible(CreditCardSystem, Transaction-Notification).
9. Responsible(Bank, Transaction-Notification).
```

Exogenous Tasks

These are the actions that are directly not part of the target system. These are generated by external entities (e.g., some external application or system) for

notifying particular inputs or for providing few services while running behind the scene and the system in question has no direct control over their generation. Golog-family calls these as exogenous actions [18] and TaMoGolog also uses the same notation. These can be defined in a task model while using the predicates *Type* (*exog*) and *TaskType* (α, *exog*), where *exog* type is reserved for exogenous actions/tasks.

Continue Example: Transaction-Notification is an exogenous task in the Book-Purchasing complete-view task model (Section 4.4.4) that is generated by credit card system, bank, or voucher system when the transaction has been completed successfully.

```
1. Type(exog).
2. TaskType(Transaction-Notification, exog).
```

Fluents

TaMoGolog uses Golog-family approach for defining both kinds of fluents: relational and functional. It provides these three predicates for defining fluents using situation calculus.

Fluent(f): f is a fluent either functional or relational
FluentDef(f, d): fluent f is defined by definition d
FluentInit(f, x): initially, fluent f has value x

In above predicates, *FluentDef* is used when a fluent has a definition formula to evaluate its value. For example, $FluentDef(fireDanger, temp > 40)$ provides definition of fluent *fireDanger* and the value of this fluent depends on provided definition; i.e., becomes true if $temp > 40$, otherwise false. The predicate *FluentInit* defines the values of fluents at the initial time of the task model. TaMoGolog also provides a predicate *InitialState* that defines the state of the task model at initial state when no task from the task model has been executed so far.

$InitialState(m) \equiv \Omega_m$

Where m stands for the name of the task model, while situation calculus formula Ω_m is a conjunction of all fluents' initial states and may have other axioms that provide some knowledge about the initial state of the task model. This predicate is equivalent to Golog initial situation predicate S_0.

Precondition Axioms and Postcondition Effects

TaMoGolog uses situation calculus approach [102] for defining the set of precondition axioms for each unit task and the postcondition effects on fluents

after execution of a unit task. For defining the precondition axioms, it provides predicate:

Precondition $(\upsilon) \equiv \Pi_\upsilon$

Which is equivalent to situation calculus predicate $Poss(\alpha, s) \equiv \Pi_\alpha(s)$, where formula Π_υ is a conjunction of all conditions under which the unit task is possible to execute and is equivalent to situation calculus formula Π_α by restoring situations s.
For defining the postcondition effects on fluents after execution of a unit task, TaMoGolog uses predicate:

Postcondition $(\upsilon, F(\vec{x}), \Omega_F(\vec{x})) \equiv \Phi_F(\vec{x})$

Which defines that executing task υ has an effect on fluent F under any condition Ω_F and the new value of fluent F becomes according to the situation calculus formula Φ_F, and $\vec{x}$ provides free variables in the predicate. This predicate is equivalent to the successor state axioms in situation calculus defined as $F(\vec{x}, do(\alpha, s)) \equiv \Theta_F(\vec{x}, do(\alpha, s), s)$ where Θ_F is a combination of Φ_F and Ω_F. TaMoGolog emphasizes on simple syntax form (more near to *Prolog* style) for writing postcondition effects on fluents as per nature required for our task modeling framework.

Continue Example: Few examples of fluents, the precondition axioms, and the postcondition effects to fluents from the Book-Purchasing complete-view task model (Section 4.4.4):

```
1. Fluent(OrderPrice).          FluentInit(OrderPrice, 0).
2. Fluent(ShoppingCartList).    FluentInit(ShoppingCartList, {}).
3. Fluent(ShippingPrice).       FluentInit(ShippingPrice, 0).
4. Fluent(BooksPrice).          FluentInit(BooksPrice, 0).
5. Fluent(CardSelected).        FluentInit(CardSelected, false).
6. Fluent(BankSelected).        FluentInit(BankSelected, false).
7. Fluent(VoucherSelected).     FluentInit(VoucherSelected, false).
8. Fluent(SelectedBook).        FluentInit(SelectedSelected, {}).
9.
10.Precondition(Confirm-Payment) ≡ CardSelected ∨ BankSelected ∨
11.                                          VoucherSelected.
12. Precondition(Price-Generation) ≡ ¬(ShoppingCartList = {}).
13.
14.Postcondition(Adding-Book-To-ShippingCart,ShoppingCartList,true)≡
15.                     ShoppingCartList ∪ {SelectedBook}.
16.Postcondition(Price-Generation,OrderPrice,true)≡
17.                     BooksPrice + TotalVAT + ShippingPrice.
```

Failure Handling: Off-line Alternative Execution Construct

TaMoGolog uses Golog-BDI adoption approach [107] for providing failure handling constructs where in the case of the failure of a task path to achieve some goal, the system adopts the alternative path. The alternative path may suggest a new path to achieve the goal or may ask to try the same path, it entirely depends on what is in the alternative path. TaMoGolog provides two constructs: the first one is *online alternative execution* construct $[\Gamma_1 \rhd \Gamma_2]$ directly taken from Golog-BDI adoption [107]; while the second one *off-line alternative execution* construct $[\Gamma_1 \rhd_\Sigma \Gamma_2]$ is defined by TaMoGolog based on the Golog-BDI adoption approach by using the IndiGolog *off-line search* construct. TaMoGolog provides two kinds of *Trans* definition for the *off-line alternative execution* construct $[\Gamma_1 \rhd_\Sigma \Gamma_2]$. They are:

This first *Trans* definition supports partially off-line in the sense that off-line search block is used explicitly just on the first path. The alternative path may have its own search block, but not explicitly provided by the *Trans* definition. So:

$$\begin{gathered} \mathit{Trans}([\Gamma_1 \rhd_{\Sigma_p} \Gamma_2], s, \Gamma', s') \equiv \\ (\exists \Gamma_1'.Trans(\Gamma_1, s, \Gamma_1', s') \wedge \Gamma' = \Gamma_1' \rhd_{\Sigma_p} \Gamma_2 \wedge \exists s^*.Do(\Gamma_1', s', s^*)) \vee \\ (\neg \exists s^*.Do(\Gamma_1, s, s^*) \wedge \mathit{Trans}(\Gamma_2, s, \Gamma', s')). \end{gathered}$$

Which implies that the transaction will be from Γ_1 if and only if there exists some path that leads this selected task towards the finishing state; otherwise, Γ_1 will be discarded and the transaction will be from Γ_2. The subscript Σ_p in the construct stands for partial-searching.

This second *Trans* definition supports fully off-line in the sense that off-line search block is applied explicitly on both paths, the first one and the alternative path. So:

$$\begin{gathered} \mathit{Trans}([\Gamma_1 \rhd_{\Sigma_f} \Gamma_2], s, \Gamma', s') \equiv \\ (\exists \Gamma_1'.Trans(\Gamma_1, s, \Gamma_1', s') \wedge \Gamma' = \Gamma_1' \rhd_{\Sigma_f} \Gamma_2 \wedge \exists s^*.Do(\Gamma_1', s', s^*)) \vee \\ (\neg \exists s^*.Do(\Gamma_1, s, s^*) \wedge \mathit{Trans}(\Gamma_2, s, \Gamma', s') \wedge \exists s^{**}.Do(\Gamma', s', s^{**})). \end{gathered}$$

This implies that the transaction will be from Γ_1 if and only if there exists some path that leads this selected task(s) towards the finishing state; otherwise, Γ_1 will be discarded. Then Γ_2 will also be checked for the existence of some path in it that leads it to the finishing state. If such a path exists then the transaction will be from it; otherwise, it will be assumed as the failure of the whole path. The subscript Σ_f in construct stands for full-searching.

The *Final* definition is same for both cases, where it is assumed to be in final if Γ_1 is in final form or in the second case when there is no success path in Γ_1 and Γ_2 is in final state.

$$Final([\Gamma_1 \rhd_\Sigma \Gamma_2], s) \equiv Final(\Gamma_1, s) \vee \neg\exists s_*.Do(\Gamma_1, s, s_*) \wedge Final(\Gamma_2, s).$$

Goals in a Task Model

A task-model structures system activities (tasks) and forms relations between them to define how to achieve some targeted goal (or a set of goals) through performing these set of activities. In Golog-family, a high-level program proceeds till it finishes its execution successfully and provides the end state. Whereas, a goal is a state of the system that is described by a set of fluents with particular values that we want to achieve. If we reach to a state where the task model finishes successfully after executing a series of tasks, says $[\alpha_1, \alpha_2, ..., \alpha_n]$ where α_1 is executed before α_2 and so on, and the set of fluents representing the goal is subset of the set of fluents of the current state of the system, and the goal formula satisfies the conjunction of the goal related fluents' current states; then we say that our task model finished successfully and also achieved the targeted goal.

Keeping in this view, TaMoGolog provides the following predicate to write goal(s) in a task model.

$$Goal(g) \equiv \Delta_g$$

Where g stands for goal name and Δ_g is a situation calculus formula for representing state of set of fluents that must be achieved at the finishing state of the task model to achieve the desired goal. It is possible to reach more than one ending states due to the nondeterministic branches of the task model; so, there may be more than one path to achieve the same goal or there may be more than one goal to be achievable through one task model. In TaMoGolog, a task model is considered to be successful if any one of these goals have been achieved along the finishing state of the task model; otherwise, it is considered to be the failure of task model.

Overall, the task model must satisfy the Do predicate provided in ConGolog specification [18], which represents the successful execution of the task model m starting from situation s and finishing in situation s' . In TaMoGolog, at finishing situation s' some goal must also be satisfied. Formally:

$$Do(m, s, s') \equiv \exists m'.Trans^*(m, s, m', s') \wedge Final(m', s') \wedge \exists g.\Delta_g(s').$$

Domain of a Task Model

The above provided predicates are used to define properties of a task model. Additional predicates can be added to define low-level details and to provide any domain knowledge in task models. In summary, D_{TM} is a domain of a task model in TaMoGolog that is specified by a theory in the following form:

- Axioms describing task modeling framework structure
- Set of tasks and axioms describing their types
- Set of fluents and axioms for defining fluents' definition and initial values
- For each *unit* task, set of precondition axioms
- For each *unit* task, set of postcondition effects (successor state axioms) that describe how the execution of the task changes the values of effected fluents
- Definition of each *composite* and *waiting* task through Golog procedure definition
- Axioms describing external entities (agents) and their responsible tasks (e.g., exogenous tasks)
- Axioms describing predicates for any domain knowledge
- Set of interrupts and their definitions
- Axioms describing the initial state of the task model
- Axioms describing the set of goals of the task model

4.6 A Framework for External Nondeterministic Constructs

GameGolog [20], a recent extension to the Golog-family, provides a way to model external entities, called agents, participating in making decisions for nondeterministic constructs. TaMoGolog uses constructs from GameGolog alongside its own three constructs based on the approach defined by GameGolog. At the higher level, TaMoGolog uses the same semantics as defined by GameGolog, but it differs slightly when defining the domain theory using situation calculus at lower level.

First, we provide some common assumptions. Following, when we talk about *agent*, we mean it as some external entity that can be another system, some external application, or some human user (which, in fact, interacts with the system through some interface provided by the system itself or by some external application/system, and this interface acts just as a communication channel between the human user and the system; e.g., when a user decides a payment method on a web page, the web page provides a communication channel between the user and the system). Another assumption, when we say the system; we mean it as *the system in question* that is reflected by the task

model or the *main user program* in the case of Golog-family based high-level program. Following is TaMoGolog approach for defining the framework for external nondeterministic constructs.

Nondeterministic Decision Environment and Agent Environment

TaMoGolog defines a way to give the control to the agent for making nondeterministic decision. We say, *nondeterministic decision environment* is a point in a task model or in a high-level program where there comes a nondeterministic choice branch. TaMoGolog assumes that in normal circumstances, the system (normally the user program in Golog-family) controls over the nondeterministic decision environment, unless otherwise stated. In the cases of internal nondeterministic constructs, as the system controls over environment so, ideally, it makes the decision of the branch nondeterministically. Sometimes, this nondeterministic decision depends on the implementation environment or based on some conditions.

In the case of external nondeterministic constructs, TaMoGolog assumes that the system gives the control to the interested agent for making the nondeterministic decision. Then, the system performs as instructed by the agent. TaMoGolog calls it as *agent environment* where the agent has been given the control to make decision in nondeterministic decision environment. The predicate *AgtEnv* is used to define an agent environment. An agent can make the decision only if it has control over the agent environment and then it makes the system aware of such decision. At this point the system takes the control back from the agent and performs as instructed by the agent. It is also noted that at any given moment, only one agent can have the control of agent environment and the system manages this control in order to avoid any collision.

Actions Type

For this framework, TaMoGolog differentiates actions into three categories: *controlling* actions that are used to manage the control of agent environment, *choice* actions based on GameGolog approach that indicate the agent's decisions; and *normal* actions/tasks that, in fact, are all other actions and are executed as part of the program to perform different things. GameGolog provides the foundation for these using a specialization of situation calculus where each action has an agent parameter along other parameters. TaMoGolog assumes that only the controlling and choice actions have an agent parameter explicitly in order to record the decision made by an agent.

Agent Environment Framework

TaMoGolog provides actions, fluents, and predicates using situation calculus for managing the agent environment to give the control to the agent for making nondeterministic decision. For this, TaMoGolog provides two controlling actions; one that gives control to a particular agent for making the nondeterministic decision, and the second is to take the control back from the agent. TaMoGolog assumes that an agent can only make the nondeterministic decision after attaining the control of agent environment and during this time when an agent has control over the environment; no other agent can take the control of the environment or make any nondeterministic decision. The two controlling actions are:

startControl(*agt*): the agent *agt* has been given the control of the agent environment
endControl(*agt*): the control of the environment has been taken from agent *agt*

We assume that these actions do not affect fluents in any other way than the ones' defined below. TaMoGolog provides a domain independent fluent $Free(s)$ that indicates whether the agent environment is free to take control by any agent for making nondeterministic decision. Initially, the *Free* is true and as an agent gets the control, it becomes false. The system gives control to an agent only if the Free is true; that means, currently no other agent has control over environment; otherwise, if the *Free* is false then the requesting agent waits till it becomes true. When an agent leaves the control, the *Free* becomes true so any other agent is eligible to get the control. The effect axioms for *Free* fluent are:

$$Free(do(\exists agt.endControl(agt), s))$$
$$\neg Free(do(\exists agt.startControl(agt), s))$$

These describe that the fluent *Free* becomes false when the agent has been given the control through *startControl* action, while becomes true again after the control has been taken from the agent through *endControl* action. The success state axioms for the fluent *Free* are:

$$Free(do(a, s)) \equiv$$
$$\exists agt.a = endControl(agt) \vee (Free(s) \wedge \neg(\exists agt.a = startControl(agt)))$$

TaMoGolog provides a functional fluent $controlAgt(s)$ that provides the name of the agent who currently has control over the agent environment. TaMoGolog also provides a special functional defined fluent $sysAgt(s)$ that gives the name of the system, as when the control is taken from any agent, it is given back to the system and at this stage the *controlAgt* (s) contains the system name. The *sysAgt* (s) is defined as:

$$sysAgt(s) = sys \stackrel{\text{def}}{=} SysAgent(sys)$$

In a task model, there can be only one system agent defined through *sysAgt* predicate. Formally:

$$sysAgt(sys) \wedge sysAgt(sys') \Rightarrow sys = sys'$$

The successor state axioms for *controlAgt* (s) are:

$$controlAgt(do(a, s)) = agt \equiv$$
$$a = startControl(agt) \vee (\exists agt'.a = endControl(agt') \wedge agt = sysAgt(s)) \vee$$
$$(controlAgt(s) = agt \wedge (a \neq startControl(agt) \vee a \neq endControl(agt)))$$

TaMoGolog provides a special predicate *AgtEnv*(*agt, s*) that takes an agent name and tells whether currently the agent has control over the agent environment. In a situation s, only one agent can have control over the agent environment. The effect axioms for *AgtEnv* are:

$$AgtEnv(agt, do(startControl(agt), s))$$
$$\neg AgtEnv(agt, do(endControl(agt), s))$$

Which say that an agent gets control of agent environment after the system gives the control through *startControl* action and the agent does not stay any longer in it after the system takes back the control through *endControl* action. The successor state axioms for the predicate *AgtEnv* are:

$$AgtEnv(agt, do(a, s)) \equiv$$
$$a = startControl(agt) \vee (AgtEnv(agt, s) \wedge a \neq endControl(agt))$$

The axiomatization of *AgtEnv* predicate contains the following properties:

1. An agent keeps control of agent environment between *startControl* and *endControl* actions, and it is possible that the system performs some choice actions between these controlling actions.

$$AgtEnv(agt, s) \Leftrightarrow \exists \alpha, s', s'', s'''.\{$$
$$s = do(startControl(agt), s') \wedge$$
$$(s'' = do(endControl(agt), s) \wedge \neg AgtEnv(agt, s'')) \vee$$
$$(ChoiceAct(\alpha) \wedge Poss(\alpha, s) \wedge s'' = do(\alpha, s) \wedge AgtEnv(agt, s'') \wedge s''' =$$
$$do(endControl(agt), s'') \wedge \neg AgtEnv(agt, s'''))$$
$$\}$$

2. If an agent gets control over agent environment and any action other than *endControl* happens, the agent still controls the agent environment in new state.

$$AgtEnv(agt, s) \Leftrightarrow$$
$$\exists s', \forall a\{s' = do(a, s) \wedge a \neq endControl(agt) \wedge AgtEnv(agt, s')\}$$

3. In a situation s, only one agent has the control over agent environment to make nondeterministic decision.

$$AgtEnv(agt, s) \wedge AgtEnv(agt', s) \Rightarrow agt = agt'$$

The precondition axioms for controlling actions are:

$$Poss(startControl(agt), s) \equiv Free(s) \wedge \neg AgtEnv(agt, s)$$
$$Poss(endControl(agt), s) \equiv \neg Free(s) \wedge AgtEnv(agt, s)$$

The *startControl* action is possible if the agent environment is free to given control to any agent. The *endControl* action is possible if the agent environment is not free and currently the same agent controls it.

Choice Actions

TaMoGolog uses the approach for choice actions same as described by GameGolog [20]. These are special actions to model the decisions of agents. These actions have an agent parameter in order to save agent's decision in the history and to make sure that only the controlling agent is making the decision. TaMoGolog uses the following set of choice actions from GameGolog:

$left(agt)$: the agent decides to select the left choice
$right(agt)$: the agent decides to select the right choice
$continue(agt)$: the agent decides to continue the iteration
$stop(agt)$: the agent decides to stop the iteration
$pick(agt, x)$: the agent picks a binding for variable x

TaMoGolog provides this one extra choice action:

$number(agt, num)$: the agent gives the number num for telling that how many instances will participate in the concurrent iteration

TaMoGolog provides a special predicate $ChoiceAct(c)$ where c is a choice action. Only the agent that has control over agent environment can perform it. Formally:

$$do(c(agt), s) \wedge ChoiceAct(c(agt)) \Rightarrow AgtEnv(agt, s)$$

The precondition for a choice action is that the same agent has control over the agent environment. So:

$$Poss(c(agt), s) \equiv ChoiceAct(c(agt)) \wedge AgtEnv(agt, s)$$

External Nondeterministic Decision Framework

The above sets of predicates and actions provide the foundation for the framework of external nondeterministic constructs. In order to make a nondeterministic decision, TaMoGolog assumes the execution of three actions, the first one is a controlling action for giving the control to the agent, the second one is a choice action performed by the agent to make the nondeterministic decision, while the last one is again a controlling action to take back the control from the agent. So:

$$do(endControl(agt), do(c(agt), do(startControl(agt), s)))$$

Where c (agt) is a choice action performed by agent. It is interesting to note that in the case of concurrency, if a process is in the agent environment then any other process cannot enter into the agent environment as we are using a single predicate *Free* for all processes for controlling the agent environment. If any other process wants to get control over the agent environment then it needs to wait till the first one leaves the control.

The *Trans* function of a choice action in GameGolog is equivalent to:

$$\textbf{1.}\ Trans(c_{GG}, s, c'_{GG}, s') \equiv Poss(c_{GG}, s) \wedge s' = do(c_{GG}, s) \wedge c'_{GG} = nil$$

Where c_{GG} is a choice action in GameGolog. On the other side; in TaMoGolog, a choice action is executed between the controlling actions. Formally:

$$\begin{gathered}\textbf{2.}\ Do(\delta_{TGend}, s, s') \equiv \exists agt, s'', s'''.\{ \\ Poss(startControl(agt), s) \wedge s'' = do(startControl(agt), s) \wedge \\ Poss(c_{TG}, s'') \wedge s''' = do(c_{TG}, s'') \wedge Poss(endControl(agt), s'') \wedge \\ s' = do(endControl(agt), s''')\}\end{gathered}$$

Where δ_{TGend} is a program in TaMoGolog that contains a sequence of three actions; i.e., $startControl(agt), c_{TG}(agt), endControl(agt)$; and c_{TG} is a choice action in TaMoGolog. So, we can define $Do(\delta_{TGend}, s, s')$ as:

$$\begin{gathered}\textbf{3.}\ Do(\delta_{TGend}, s, s') \equiv \\ s' = \exists agt.do(endControl(agt), do(c_{TG}, do(startControl(agt), s)))\end{gathered}$$

In other words, $Do(\delta_{TGend}, s, s')$ in terms of three *Trans* functions:

$$\begin{gathered}\textbf{4.}\ Do(\delta_{TGend}, s, s') \equiv \exists s'', s'''.\{ \\ Trans(t_{TG}, s, t'_{TG}, s'') \wedge Trans(c_{TG}, s'', c'_{TG}, s''') \wedge Trans(t_{TG}, s''', t'_{TG}, s')\}\end{gathered}$$

Where t_{TG} and c_{TG} are controlling action and choice action respectively, and their *Trans* definitions are:

5. $Trans(t_{TG}, s, t'_{TG}, s') \equiv Poss(t_{TG}, s) \wedge s' = do(t_{TG}, s) \wedge t'_{TG} = nil$
6. $Trans(c_{TG}, s, c'_{TG}, s') \equiv Poss(c_{TG}, s) \wedge s' = do(c_{TG}, s) \wedge t'_{TG} = nil$

If we compare definition 1 with 2 and 4, the difference is the controlling actions before and after the choice action. As controlling actions do not affect fluents other than ways described above; so, we can define a transformation function that can transfer a GameGolog choice action into TaMoGolog program (δ_{TGend}) for external nondeterministic decision:

7. $\Upsilon(c_{GG}) = \delta_{TGend} = startControl(agt); c_{TG}; endControl(agt)$

Performing $\Upsilon(c_{GG})$ is:

8. $do(\Upsilon(c_{GG}), s) \equiv do(endControl(agt), do(c_{TG}, do(startControl(agt), s)))$

Therefore:

For every choice action in GameGolog, we have a transformation function that can transfer GameGolog choice action into TaMoGolog program for external nondeterministic decision; where the Do and Final are:

$$Do(\Upsilon(c_{GG}), s, s') \equiv Do(\delta_{TGend}, s, s')$$
$$Final(\Upsilon(c_{GG}), s) \equiv Final(\delta_{TGend}, s)$$

Here, c_{GG} stands for a choice action in GameGolog and δ_{TGend} stands for a program for external nondeterministic decision in TaMoGolog as defined in definition 7.

Trans and *Final* Predicates for External Nondeterministic Constructs

TaMoGolog uses GameGolog choice actions with transformation function where these come in the definition of *Trans* and *Final*. For example, the GameGolog choice action $left(agt)$ is replaced with $\Upsilon(left(agt))$ that transfers it into $(startControl(agt); left(agt); endControl(agt))$.

So, the *Trans* and *Final* definition of GameGolog external nondeterministic branch construct $[agt\,\Gamma_1 \mid \Gamma_2]$ in TaMoGolog is:

$$Trans([agt\,\Gamma_1 \mid \Gamma_2], s, \Gamma', s') \equiv$$
$$s' = do(\Upsilon(left(agt)), s) \wedge \Gamma' = \Gamma_1 \vee s' = do(\Upsilon(right(agt)), s) \wedge \Gamma' = \Gamma_2$$

$$Final([agt\,\Gamma_1 \mid \Gamma_2], s) \equiv Flse$$

The same rule applies to other external nondeterministic constructs that are taken from GameGolog. The *Trans* and *Final* definitions of them are already defined in Section 4.2.2. The difference is that in the places of choice actions, TaMoGolog uses the transformation function definition.

Here, we provide *Trans* and *Final* definitions of TaMoGolog own defined external nondeterministic constructs.

1. *External selected priority concurrency* $[agt\,\Gamma_1\langle\rangle\Gamma_2]$:

$$\begin{gathered}Trans([agt\,\Gamma_1\langle\rangle\Gamma_2], s, \Gamma', s') \equiv \\ s' = do(\Upsilon(left(agt)), s) \wedge \Gamma' = (\Gamma_1\rangle\rangle\Gamma_2) \vee \\ s' = do(\Upsilon(right(agt)), s) \wedge \Gamma' = (\Gamma_2\rangle\rangle\Gamma_1)\end{gathered}$$

$$Final([agt\,\Gamma_1\langle\rangle\Gamma_2], s) \equiv False$$

   The *Trans* definition says that the agent chooses to go to the left or to the right, and then the remaining task structure is the normal concurrent priority with the chosen task has higher priority.

2. *External first-step decision concurrency* $[agt\,\Gamma_1\langle|\rangle\Gamma_2]$:

$$\begin{gathered}Trans([agt\,\Gamma_1\langle|\rangle\Gamma_2], s, \Gamma', s') \equiv \\ s' = do(\Upsilon(left(agt)), s) \wedge \Gamma' = (\Gamma_1\langle\langle|\ \Gamma_2) \vee \\ s' = do(\Upsilon(right(agt)), s) \wedge \Gamma' = (\Gamma_2\ |\rangle\rangle\Gamma_1)\end{gathered}$$

$$\begin{gathered}Trans(\Gamma_1\langle\langle|\ \Gamma_2, s, \Gamma', s') \equiv \exists\Gamma_1'.\Gamma' = (\Gamma_1'\ \|\ \Gamma_2) \wedge Trnas(\Gamma_1, s, \Gamma_1', s') \\ Trans(\Gamma_1\ |\rangle\rangle\Gamma_2, s, \Gamma', s') \equiv \exists\Gamma_2'.\Gamma' = (\Gamma_1\ \|\ \Gamma_2') \wedge Trnas(\Gamma_2, s, \Gamma_2', s')\end{gathered}$$

$$\begin{gathered}Final([agt\,\Gamma_1\langle|\rangle\Gamma_2], s) \equiv False \\ Final(\Gamma_1\langle\langle|\ \Gamma_2, s) \equiv Final(\Gamma_1, s) \wedge Final(\Gamma_2, s) \\ Final(\Gamma_1\ |\rangle\rangle\Gamma_2, s) \equiv Final(\Gamma_1, s) \wedge Final(\Gamma_2, s)\end{gathered}$$

   These above definitions say that whichever choice the agent chooses, the next transition will be from the chosen task, and then the remaining part is a concurrent tasks structure between the remaining part of the chosen task and the unselected task. To handle this, TaMoGolog introduce new "auxiliary" constructs $(\Gamma_1\langle\langle|\ \Gamma_2)$ and $(\Gamma_1\ |\rangle\rangle\Gamma_2)$ that model this state of computation after the agent chooses to go left or right respectively. It is interesting to note that these two new "auxiliary" constructs are in final state only if both participated tasks are in final states.

3. *External selected concurrent iteration* $[agt\,\Gamma]^{\|}$:

$$Trans([agt\,\Gamma]^{\|}, s, \Gamma', s') \equiv$$
$$s' = do(\Upsilon(number(agt)), s) \wedge$$
$$((num = 0 \wedge \Gamma' = True?) \vee (num > 0 \wedge \Gamma' = [\Gamma]^{\|^{num}}))$$

$$Final([agt\,\Gamma]^{\|}, s) \equiv False$$

The *Trans* definition describes that the external entity gives the number it wants the task instances in the concurrent iteration, and then the remaining tasks structure is the concurrent iteration of the task with that given number of instances.

Mapping to ConGolog Program

As defined in [20], GameGolog programs that do not involve the external every-step concurrency construct can be expressed directly into the ConGolog programs. For this, it defines a translation function ∂ by induction on the structure of GameGolog program. We apply a translation function ∂_{TG}, after amending the original translation function ∂ defined in [20], by induction on TaMoGolog programs. Interestingly, TaMoGolog own defined concurrency related constructs except the external first-step decision concurrency construct; i.e. $[agt\,\Gamma_1\langle|\rangle\Gamma_2]$, can also be expressed directly in ConGolog programs. Here we define these except those two constructs (external normal concurrency and external first-step decision concurrency):

$$\partial_{TG}(\alpha) = \alpha$$
$$\partial_{TG}(\phi?) = \phi?$$
$$\partial_{TG}(\Gamma_1; \Gamma_2) = \partial_{TG}(\Gamma_1); \partial_{TG}(\Gamma_2)$$

$$\partial_{TG}([agt\ \Gamma_1 \mid \Gamma_2]) =$$
$$startControl(agt); [left(agt); endControl(agt); \partial_{TG}(\Gamma_1) \mid$$
$$rigt(agt); endControl(agt); \partial_{TG}(\Gamma_2)]$$

$$\partial_{TG}([agt\ \pi x.\Gamma]) =$$
$$startControl(agt); \pi x.pick(agt, x); endControl(agt)\partial_{TG}(\Gamma)$$

$$\partial_{TG}([agt\ \Gamma]^*) =$$
$$startControl(agt); (continue(agt); endControl(agt); \partial_{TG}(\Gamma); startControl(agt))^*;$$
$$stop(agt); endControl(agt)$$

$$\partial_{TG}([agt\ \Gamma_1\langle\rangle\Gamma_2]) =$$
$$startControl(agt); [left(agt); endControl(agt); \partial_{TG}(\Gamma_1\rangle\rangle\Gamma_2) \mid$$
$$rigt(agt); endControl(agt); \partial_{TG}(\Gamma_2\rangle\rangle\Gamma_2)]$$

$$\partial_{TG}([agt\ \Gamma]^{\|}) = \\ startControl(agt); number(agt, num); endControl(agt); [(num = 0)?; True \mid \\ (num > 0)?; \partial_{TG}([\Gamma]^{\|^{num}})]$$

Implementation of the Framework for External Nondeterministic Constructs

We provide the low-level implementation details of the framework for external nondeterministic constructs, defined above, in two forms. **Appendix A** provides Golog-family based high-level program syntax and **Appendix B** provides Prolog-based syntax targeting IndiGolog interpreter implementation platform `P-INDIGOLOG` [104, 105]. The Prolog-based coding of the framework can be used directly in `P-INDIGOLOG` platform for utilizing external nondeterministic constructs in Golog-family based user programs. The programmer needs to define in the *device manager* of the agent platform (external application or system) the interpretation of framework actions (e.g., *choice* actions) in order to receive requests from the system for making nondeterministic decisions and to feed back responses. As discussed earlier in this section, all other external nondeterministic constructs except two are defined directly in ConGolog-based high-level programs. For TaMoGolog own defined two "auxiliary" constructs ($\Gamma_1\langle\langle|\ \Gamma_2$) and ($\Gamma_1\ |\rangle\rangle\Gamma_2$), we provide *Trans* and *Final* definitions at the `P-INDIGOLOG` platform level in Appendix B. Furthermore, **Appendix C** provides a way to implement external nondeterministic constructs through our defined labeling framework. This is critical considering the practical and implementation constraints of the platform.

4.7 Summary

This chapter has presented a framework for task modeling. The framework provides a set of concepts for establishing a conceptual foundation to structure task models from different abstractions. There was need of an expressive, dynamic, and well-defined (syntactically and semantically) language to construct task models as suggested by the proposed framework. We provided the definition of a formal task modeling language, called TaMoGolog, on the top of the foundations of Golog-family for constructing dynamic and rich task models.

In the first part of the chapter, after the preliminary background we provided the details of the proposed framework concepts with an example. The second part of the chapter provided the formalization details of the TaMoGolog. First, it provided the TaMoGolog set of constructs mostly obtained from Golog-family with addition to few of its own defined constructs. Secondly, it provided the formal syntax and semantics framework of the TaMoGolog. Thirdly, it defined the formalization of framework for external nondeterministic constructs

at the language level using the GameGolog approach. This enables to model the external entities' participation in making nondeterministic decisions at run time. The low level implementation of the framework for external nondeterministic constructs are provided in Appendix A, Appendix B, and Appendix C.

Overall, this chapter provides a foundation that is used in the subsequent chapters for our work of task model-based usability evaluation at the development environment level.

Chapter 5

Task Model-based Usability Evaluation in Development Environment

5.1 Motivation

This chapter is devoted to describe **TaMU** (**Ta**sk **M**odel-based **U**sability Evaluation) framework, which defines how to manage and automate experiments at the Integrated Development Environment (IDE) level for conducting usability evaluation based on task models created in TaMoGolog.

Usability evaluation aims at involving users, especially product end-users, and experts (e.g., UI experts, system analyst, etc) in the evaluation process of a specific product to find usability flaws and errors and refine the product in accordance with the feedback. Usability evaluation is performed using existing rigorous approaches and techniques that enable the process of defining and running experiments, collecting and analyzing results, and making decisions regarding which feedback to adopt and to what extent [25]. Unfortunately, in many cases these usability evaluation techniques are performed manually [63], and due to the budget and schedule concerns sometimes they are neglected or poorly defined. In addition, the evaluation is performed, in many cases, at the end when it is difficult to make changes in the design [25]. Automating evaluation methods and techniques, and applying them throughout the development process, provides several benefits, e.g.; reduced development costs and time, improved error tracing, better feedback, and increased coverage of evaluated features [63].

> *It is important to note that we use the term usability evaluation for the evaluation of both product usability and functionality. We use experiments to find usability issues and serve as a kind of acceptance test for the developed features.*

The motivation behind integrating and automating the evaluation process into the software development environment, i.e., into the IDE, is clear. Defining evaluation experiments and running them from within the IDE equips the software development team with the mechanisms to monitor and control a continuous evaluation process tightly coupled with the development process, thus receiving on-going user feedback while continuing development. This was already suggested in our three-fold integrated framework in Chapter 3 and the UEMan tool presented there includes, among others, traceability between conclusions of a specific evaluation experiment to the appropriate parts of code that implement the conclusions, and vice versa. Even though UEMan provides an effective framework to perform and automate the evaluation process, it lacks the ability to model formally user and system tasks and behavior. Thus, the experiment itself is not formalized in a way that can enable automatic analysis. Here, we focus on how we can formally model user and system tasks and their behavior thus providing automatic analysis of user and system recorded data.

To achieve the above goal, we propose a way to define task model-based usability evaluation from within the IDE thus providing development teams with the ability to receive users' on-going feedback during development, and enable them to automatically collect and analyze users and system behavior to recognize usability flaws and errors in an efficient and effective way.

This chapter is organized as follows:

Section 5.2 introduces TaMU framework that defines our approach towards task model-based usability evaluation and describes an end-to-end life-cycle to manage and automate this evaluation process at the IDE level.

Section 5.3 describes the role of TaMoGolog as task modeling language in TaMU framework. It explains the effects of tagging tasks and variables at the code level, benefits of TaMoGolog for usability evaluation, the reflection of scenarios in evaluation experiments through TaMoGolog-based task models, the role of TaMoGolog-based task models in recording users and system activities and behavior during execution of experiments, and the process of performing automated analysis with the help of task models.

Section 5.4 presents TaMUlator, a java-based tool, that works at the IDE level to provide the realization of TaMU process life-cycle through providing a set of APIs and interfaces. It also presents a case study in which six development teams used TaMUlator to evaluate a software project they developed.

Section 5.5 highlights related work, especially those automated tools that also provide task model-based usability evaluation.

Section 5.6 concludes and explains the differences of our approach from the previous ones.

The TaMUlator tool and the evaluation case study presented in Section 5.4 is author's collaborative work with *Dr. Yael Dubinsky*, who taught one year course "Annual Project in Software Engineering"in Computer Science Department at Technion, IIT from 2008 to 2010. The TaMUlator tool presented here was developed by one of the six teams during the course session 2009/10.

5.2 TaMU Framework

High-level usability is acknowledged as a significant feature of software products. For this, we provide a framework, called **TaMU** (**Ta**sk **M**odel-based **U**sability Evaluation), for managing and automating evaluation experiments at the IDE level based on formal task models and then analyzing the recorded data to highlight usability issues. The TaMU framework handles it at three levels: *process-level*, *model-level*, and *tool-support-level*. Figure 5.1 shows TaMU framework and its three pillars.

- *At the process-level*, the framework describes an end-to-end evaluation life cycle, called *TaMU life-cycle*, for describing and managing evaluation experiments using TaMoGolog-based task models at the IDE level.

- *At the model-level*, the framework uses TaMoGolog-based task models to implement model-based usability evaluation approach. The framework uses TaMoGolog to model the interested set of user and system tasks and their complex behavior for evaluating the targeted application. The created task models provide a way to record users and application data while evaluating the targeted application, and a mean for the analysis of the recorded data to highlight usability issues.

- *At the tool-support-level*, the framework provides an automated tool support, called **TaMUlator** (**Ta**sk **M**odel-based **U**sability Eva**lua-tor**), for the realization of framework's evaluation life-cycle at the IDE level. This tool enables defining TaMoGolog-based task models, running evaluation experiments, recording user and system behavior as per the defined mode, and analyzing automatically the task models and recorded data for highlighting usability issues.

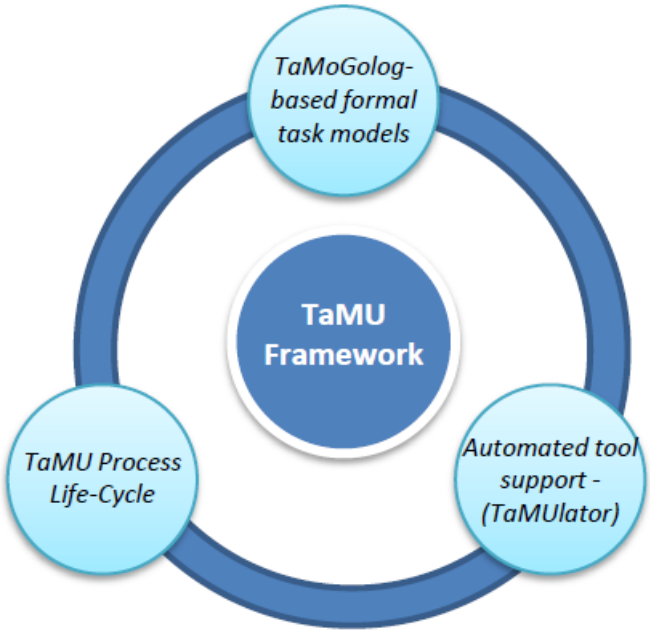


Figure 5.1: TaMU framwork

5.2.1 TaMU Process Life-cycle

The TaMU framework provides the definition of an end-to-end life-cycle, called *TaMU life-cycle*, for managing the process of usability evaluation using TaMoGolog-based formal task models in the development environment. The TaMU life cycle consists of five phases: *Tag, Task Model Creation, Evaluation-Experiment Creation, Run & Record,* and *Analysis* as shown in Figure 5.2. The TaMU process life cycle approach is best suited for iterative and incremental types of development (especially for agile development approach), thus emphasizing that the inspection of results (either automated or manual) should forward input to the development process and enable improvements in the product during the next iteration. For this reason, the figure shows that the outputs of the last two phases become the input to the developing application. The details of each phase follow.

Phase 1 - Tag: The software development team tags the program at the code level with the set of relevant tasks and those variables (fluents) that can be part of precondition axioms or postcondition effects. Through this, the software team can define the abstraction of task at multi-levels. For example, in the case of printing functionality, at upper-level the whole function can be defined as a task *"printPage"*, while at lower-level this can be a series of three

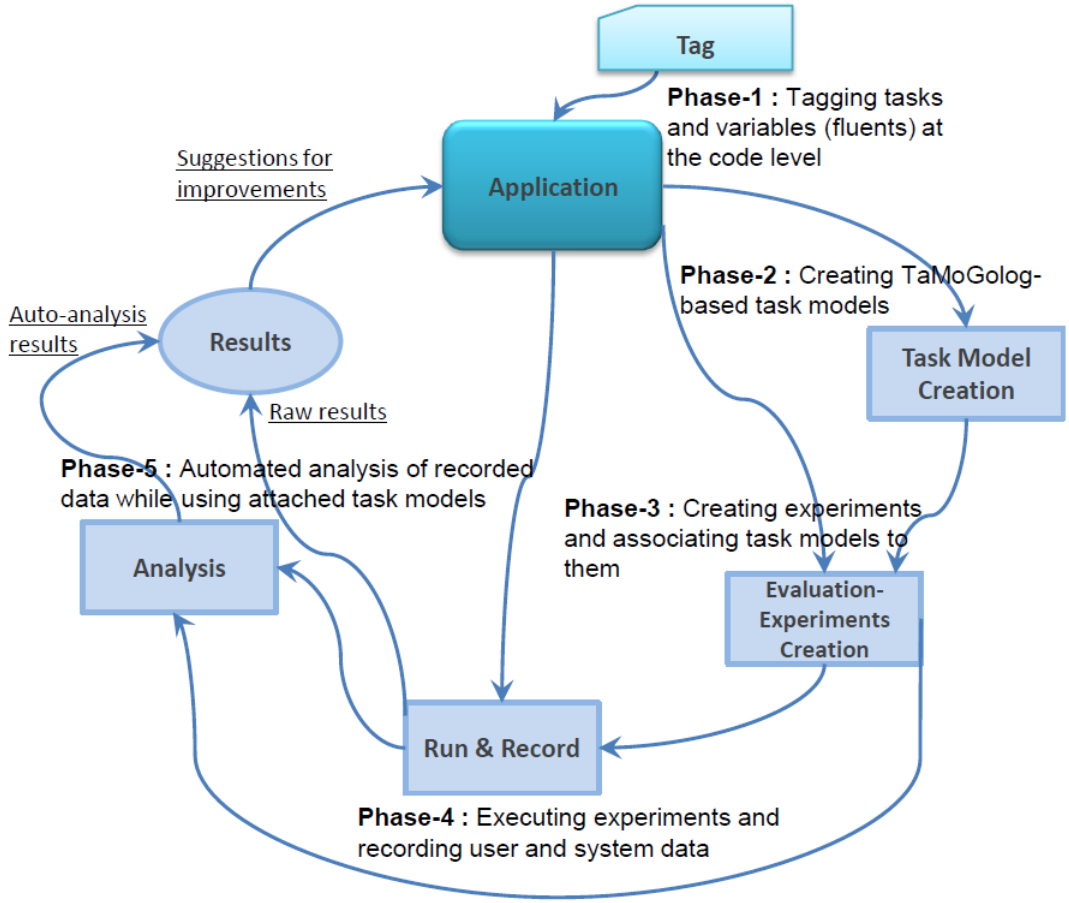


Figure 5.2: TaMU process life-cycle

tasks named as *"checkPrinter"*, *"sendRequest"*, and *"confirmPrint"*.

Tagging tasks and variables in the code gives the opportunity to define task models at different abstraction levels, which help to evaluate the product from these abstraction levels. The developed tool TaMUlator provides the functionality of tagging tasks and variables at the code level. These tagged sets of tasks and variables are then available for using in task models.

Phase 2 - Task Model Creation: The evaluator creates TaMoGolog-based task models for each planned usability scenario, where each task model constructs the structure of corresponding scenario at a certain abstraction, in order to highlight usability issues related to that scenario. It is up to the evaluator to decide how many task models at different abstraction levels are required for a particular usability scenario. TaMU framework assumes that only those tasks and variables that were tagged in the previous phase can be used in constructing task models. The task models reflect the structure of the scenarios that are supposed to be followed from the point of view of the evaluator, irrespective of the system implementation. This helps analyzing users' behavior and finding flaws in the implemented application.

Phase 3 - Evaluation-Experiment Creation: The evaluator creates evaluation experiments and links the related TaMoGolog-based task models, constructed in previous phase, to the created experiments. In each experiment, evaluating users are supposed to achieve a list of goals through performing tasks (where each goal reflects a scenario-path, e.g., achieving a goal of depositing money in the bank account) while using the developing/developed product to evaluate. This explains why the Figure 5.2 shows the developing product as input to this phase. We have already described details of such evaluation experiments in Chapter 3 using the developed plug-in tool UEMan.

Phase 4 - Run & Record: The evaluating users execute the created experiments and perform tasks on the evaluating application. During executing experiments, the user actions and the evaluated-application data is recorded as suggested in the previous section.

Phase 5 - Analysis: During this phase, the logged data of users' activities and the evaluated-application is analyzed (manually or with the help of some automated tool, e.g., TaMUlator), through some analysis criteria. The analysis results; for example errors and flaws (e.g., preconditions not fulfilled, skipped tasks, etc), usability problems (e.g., any other path selection by user that was not mentioned in the task model for achieving the targeted goal), and the user and the system behavior (e.g., user inputs, user trend for a particular path, variables values after executing tasks), etc; are created through applying

appropriate statistical techniques based on some analysis criteria.

The evaluation team inspects the results and then suggests improvements and development tasks for the next phase of development. Figure 5.2 shows this link from experiments' results to the application (for the coming iteration's development tasks), which we already have explained in details in Chapter 3 using the UEMan tool.

5.3 The Role of TaMoGolog in conducting Usability Evaluation

TaMoGolog task modeling language, described in Chapter 4, provides a solid foundation for defining complex system behaviors and scenarios in an appropriate, accurate, and unambiguous form. It provides the facility for constructing dynamic and rich task models from different views perspectives at different abstraction levels that can be used for variety of purposes such as system analysis, system design, model-based usability evaluation. It fills the gap in existing task modeling languages by providing a well-defined formal syntax and semantics, enabling precondition axioms to tasks, postcondition effects to fluents (variables), defining the way to include domain knowledge in task models, and providing a rich set of operators for handling complex system structure. These are the reasons amongst others, e.g., explicit external entities participation in making nondeterministic choices, for the section of TaMoGolog as our framework's task modeling language for conducting model-based usability evaluation.

The TaMU framework uses TaMoGolog-based task models for two purposes: to model user and system tasks and behavior for usability scenarios in order to provide a way to record interested data, and to analyze the recorded data while comparing with these created task models to highlight usability issues and to draw conclusions. In the following subsections, we describe our approach for tagging interested tasks and variables (fluents) at the application code level, the benefits of using TaMoGolog in usability evaluation, the construction of TaMoGolog-based task models for usability scenarios, the process of recoding user and system data during executing the evaluation experiment, and finally the analysis criteria for highlighting usability issues and for drawing conclusions.

5.3.1 Defining and Tagging Tasks and Variables at the Code level

Our three-fold integration approach, described in Chapter 3, and the evaluation framework both insist the integration and automation of the evaluation

process into the software development environment, i.e., into the IDE. We have earlier described the rationales behind this. Keeping the idea, TaMU framework suggests defining and tagging at the code level in the development environment the interested tasks and variables to be used in task models. Through this, it is possible to evaluate a usability scenario in an evaluation experiment from different views perspectives where each view describes the usability scenario in the task model at some certain abstraction level.

We describe a *task* as an abstract entity that hides its internal functionality, performs one or more operations/actions, and provides an overall behavior for achieving some specific goal. When tagging tasks at the code level, we can define tasks at multiple abstraction levels. For example, we can define pressing a button, selecting an option from two inputs, checking the card validity as atomic tasks. While on the other level, we can also define the complete payment transaction functionality as an atomic task. In each case, we also tag the interested set of those variables (fluents) that are either part of precondition axioms or postcondition effects of executing these tasks. This gives us the flexibility to create task models from different abstraction levels for the same usability scenario, thus enables us to evaluate the targeted application from several abstractions.

The tagging at the code level enables us to define user tasks and system internal functionalities separately in the resulting task models. This helps us to record users' actions and system internal behavior during evaluation experiments. It is useful in two aspects: the first one is related to usability issues, as sometimes the problem arises due to the system internal actions not working properly and through this process we can check it easily; the second one is testing the system internal actions and functionalities, as recording the relevant data enables us to find out testing issues such as performance, accuracy, etc. For example, if the system is using an algorithm to manage something internally and the execution of another task depends on the implementation of this algorithm and it may be possible that the task is executing slowly due to the slow response of the algorithm. Therefore, this approach, in fact, provides a way to test product functionalities during evaluation experiments. Hence, through this it also reduces the time and cost needed to spend later during the testing phase. This approach of testing system functionalities through usability experiments fits perfectly to agile development, where there is due to short-nature of iteration life-cycle these experiments can also acts as an acceptance tests for the product features.

5.3.2 Benefits of TaMoGolog in Usability Evaluation

This subsection describes the reasons and the benefits for selecting and using TaMoGolog in the TaMU evaluation framework.

Defining task models at different abstraction levels for a usability scenario: TaMoGolog and defining tasks at the code level provides a way to construct task models from multi-view perspectives at different abstraction levels, for achieving the same set of goals. So, more than one task model for a usability scenario, where each task model lays at a certain abstraction, can be attached to an evaluation experiment. This gives us the chance to record users' and system behavior and to highlight usability issues from several aspects.

Precondition axioms for tasks: TaMoGolog, as based on Golog-family [18, 19, 20, 73, 107], provides a way of writing explicitly the precondition axioms for atomic tasks. These precondition axioms attached to a task define constraints that should be met before execution of the task. This defining of precondition axioms is very important from usability evaluation perspective, as it enables to record whether the user or the system fulfilled these constraints before executing the attached task. If these constraints are not fulfilled then there are higher chances of the failure of the attached task or there is a usability issue that the interface was unable to prevent the user to fulfill all constraints before executing that task. For example, a precondition constraint for the task $withdrawMoney(amount)$ can be that the amount should be greater than zero. In this case, ideally the interface should prevent the user to give an amount zero or less than zero, and if the user is able to do it then the task will not work correctly. Through precondition axioms, we can find out where the user or the system violated these constraints while executing the attached task.

Postcondition effects on variables (fluents): The postcondition effects on variables (fluents) in TaMoGolog-based task model provide a way to check whether the executed tasks have been performed in the way we expected. These postcondition effects tell what should be the new value of these variables if we execute the associated task. Through providing this in the attached task model, we are able to record their values before the execution of task and after the execution. This enables us to check whether the task performed correctly as it supposed to be or it violated these postcondition rules. Interestingly, this provides a testing of the system functionalities while performing evaluation experiments.

External entities participation in nondeterministic decisions: TaMoGolog, using GameGolog approach [20], provides a way to define and model explicitly the external entities (external applications/systems or human users) participation in making nondeterministic decisions. From evaluation perspective, this enables to record participated external entities behavior during making these decisions and the resulting effects. For example, $(a; [agt$

$b|c]$; d) is a task structure representing that the goal is achieved either executing tasks in the order *a;b;d* or in the order *a;c;d* where external entity *agt* decides to choose between task b or task c. If the results indicate that most of the participated users chose task b but they spent more time to finish the task b compare to those participated users who chose task c, we can conclude that the users prefer to choose task b as initially it looks to them more suitable but then they find it more difficult to finish, compared to the choice of task c. These behavioral data is important from usability perspective as it enables us to find out the right solutions for the right users.

Modeling exogenous tasks: Exogenous tasks are executed by external applications/systems to either let the targeted system know their input or for providing services behind the scene. TaMoGolog provides the facility to write them and their effects on variables explicitly in task models, which enables us to record their effects on variables after their execution during evaluation experiments. This is useful from evaluation perspective, as it tells us their effects on variables and other tasks, e.g., it is possible that due to some exogenous task execution some precondition axiom for a task is no longer true and that was the reason behind the failure of that particular task.

Defining goal(s) in task models: We can define the target set of goals in TaMoGolog-based task models in which the task model is considered successful if it is finished successfully and any of the goal in the attached set is achieved. In usability evaluation, the attached set of goals is in fact the evaluation-scenario's set of goals. This enables us to know whether the user was able to achieve any goal from the targeted set of goals while performing these tasks. It is also possible that a user performed the tasks as modeled by the task model but was unable to achieve any of the goals, or on the other side the user was able to achieve any of the the targeted goals but following some other path, which was not mentioned in the task model.

Representing domain knowledge in task models: This enables us to model any domain specific knowledge in the attached task models that can be helpful during analysis of the experiment's recorded data. This can be used to get some specific knowledge about the users and the system behavior in order to understand usability issues properly.

Customizable and extensible: The customizable and extensible nature of TaMoGolog is useful in the regard that it can be fitted into different environments for usability evaluation. It is possible to define new predicates using situation calculus [102] to extend it or to redefine the previous predicates.

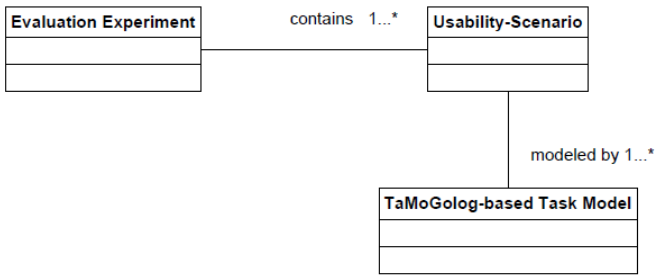


Figure 5.3: Relationship between evaluation experiment and the attached task models

Rich and powerful set of constructs: TaMoGolog powerful and rich set of constructs, mostly obtained from the Golog-family as described in Chapter 4, enables the construction of task models for complex usability scenario unambiguously and accurately. From evaluation perspective, it is important to have accurate and unambiguous task models because the analysis is carried on the basis of recorded data and the attached task models.

5.3.3 Modeling Usability Scenarios through TaMoGolog-based Task Models

A *usability-scenario* in evaluation experiments provides a scenario where the user much achieves some targeted set of goals by performing a series of tasks. The purpose of the experiment is to check usability aspects such as ease, efficiency, ability to learn and memorize, performance, and error handling of the targeted application. This is checked by applying appropriate usability metrics such as time to complete a task and time spent dealing with errors. Users perform tasks on the evaluating application to achieve targeted set of goals, while during this the evaluators or the automated tools log users' actions and system responses and then the recorded data is analyzed to check usability aspects against the targeted metrics. Model-based usability evaluation techniques use task-models [91] that describe how the set of activities are supposed to be executed in order to achieve the targeted set of goals. Then the recorded data is analyzed against the attached task models for finding out usability issues using some predefined usability metrics. The TaMU framework uses TaMoGolog, as described early, to construct task models for each usability-scenario. Through TaMoGolog, it is possible to construct different task models for the same usability scenario, where each task model reflects the usability scenario at some particular abstraction level and shows how to achieve the targeted set of goals through performing series of tasks. This helps to record data from several asbtraction levels during execution of the evalu-

ation experiment and enables us to highlight usability issues from multiple abstractions. Figure 5.3 shows that an evaluation experiment is used to analyze one or more usability scenarios where each *usability-scenario* is modeled by one or more task model at different abstraction levels that are used for analyzing the experiment results. An example of a task model for a scenario of managing bank account is given in Section 4.5.3.

Example: Managing-Bank-Account Task Model

Here, we provide a brief example of a task model of a scenario for managing bank account in which a user can view account details, or can deposit money in the account, or can withdraw money from the account. The system allows the user to perform any one of these tasks, in any particular order, after the user login to the system. The user can perform these tasks as many time as he/she wants till log out from the system. The TaMoGolog-based task model below (in brief form) provides the realization of this scenario. It defines three goals where each goal is in fact corresponding to each of the main task (viewing, depositing, withdrawing). We can divide this scenario into three sub-scenarios in which each scenario will deal with a particular goal, e.g., to view account, or to deposit money in the account, or to withdraw money from the account.

```
1  TaskModel(managing_bank_account).
2  Fluent(userAge).
3  Fluent(balance).
4  Fluent(amount).
5  UnitTask(login).
6  UnitTask(logout).
7  UnitTask(view_account).
8  UnitTask(deposit(amount)).
9  UnitTask(withdraw(amount)).
10 CompositeTask(managing_bank_account).
11 CompositeTask(manage_account).
12 Agent(customer).
13 Precondition(login) ≡ userAge > 18.
14 Precondition(view_account) ≡ TRUE.
15 Precondition(deposit) ≡ amount > 0.
16 Precondition(withdraw) ≡ (balance - amount) > 0.
17 Postcondition(deposit, balance, true) ≡ balance + amount.
18 Postcondition(withdraw, balance, true) ≡ balance - amount.
19 Goal(viewingAccount, balance) ≡ balance.
20 Goal(depositMoney, balance) ≡ balance + amount.
21 Goal(withdrawMoney, balance) ≡ balance - amount.
```

```
22 proc managing_bank_account
23   login;
24   [customer manage_account]*;
25   logout
26 end
27 proc manage_account
28   [customer (view_account|deposit(amount)|withdraw(amount))]
29 end
```

5.3.4 The Data Recording Process during Evaluation Experiments

The TaMU framework uses evaluation experiments to highlight usability issues that also serve as a kind of acceptance test for the developed features. The TaMU framework suggests usage of task models attached to the evaluation experiment and the tagged tasks and variables at the code level for recording desirable data during execution of the evaluation experiment. This recorded data describe users' and system activities and behavior, and are used for analysis purposes in order to highlight usability issues through applying some usability metrics. TaMU framework suggests to record data through following steps:

- When the user or the system starts any task (either it is modeled in any of task models attached to that evaluation experiment or it is a tagged task at the code level), the task is checked in the attached task models and if it exists there then the values of precondition axioms are recorded.

- As any task (modeled in any of the attached task models or tagged at the code level) starts execution, information about the task; such as starting time, previous task, etc; are recorded and when it successfully finishes execution then again relevant information, such as ending time, are also recorded. The TaMU framework approach of handling the tagging of tasks and variables at the code level gives the flexibility to record different kinds of information that help in the phase of automatic analysis of the logged data.

- When a task completes execution, firstly, it is checked in the attached task models and if it exists there, then the values of postcondition related variables are recorded.

- When an external entity (some external application/system or human user) gives some input or participates in any nondeterministic decision, the input or the decision is also recorded.

- At the beginning and the ending of the evaluation experiment, values for all of those variables that come in any of the attached task models are recorded, which represent task model initial state values and ending state values.

- During the experiment, if the user quits a task due to inability to finish it properly, the values of the variables related to precondition axioms and postcondition effects are also recorded.

5.3.5 The Data Analysis Criteria using TaMoGolog-based Task Models

Usability metrics are used for measuring quantitative usability aspects of the targeted system [25], such as Whiteside, Bennett, and Holtzblatt [125] provide a list of measurement criteria; e.g., time to complete a task, time spent in errors, ratio of successes to failures, etc; that can be used to determine the quantitative level of usability a system provides to its users. ISO standard 9241[1] also provides usability metrics and categorizes them through their contributions towards three aspects: *effectiveness*, *efficiency*, and *satisfaction*. Normally model-based usability evaluation approaches, such as [71] and [91], use different quantitative attributes on the recorded data as the analysis criteria to highlight usability issues.

TaMU framework focuses on those quantitative measurements in the analysis criteria that can be obtained accurately through some automated tool support using the attached TaMoGolog-based task models. Here, we briefly describe how the TaMoGolog-based task models attached to an evaluation experiment help in analyzing the recorded data of the experiment.

- Many of measurements suggested by Whiteside, Bennett, Holtzblatt [125]; such as *time to complete a task, per cent of task completed, per cent of task completed per unit time, ratio of success to failure, per cent of number of errors, frequency of help and documentation use, per cent of favorable/unfavorable user comments*; can be obtained directly from the experiment's recorded data or computed through applying the appropriate statistical techniques on the recorded data.

- We categorized tasks in the experiment's recorded data into four categories for the analysis purpose:

  - *`Completed-and-successful:`* A task is considered *completed-and-successful* if it finished execution properly and the recorded at-

[1]ISO 9241-11: Ergonomic requirements for office work with visual display terminals (VDTs)– Part 11: Guidance on usability

tached postcondition variables' values satisfy the attached postcondition effect axioms to this task in the task model. For example, if the task *deposit*(*amount*), the example defined previously in *Managing-Bank-Account* task model, finishes successfully and the recorded value of the attached variable *balance* to this task satisfies the attached postcondition effect to this task in the task model, i.e., the new *balance* is the previous *balance* plus the current depositing *amount*, then this task is considered *completed-and-successful* in the analysis result. We also verify that the precondition axioms to this task in the task model are also satisfied in order to know that the specific task performance was not affected due to the violation of certain precondition axiom.

- `Completed-and-unsuccessful:` A task is considered *completed-and-unsuccessful* when it finished execution properly but the recorded attached postcondition variables' values do not completely satisfy the attached postcondition effect axioms to this task in the task model. Here, we are interested in those variables that violated attached postcondition effect axioms to this task. For example, in the above described example if the new value of the attached variable *amount* does not satisfy the attached postcondition effect axiom to the task *deposit*(*amount*) in the task model, then we consider this task as *completed-and-unsuccessful*. We also check the precondition axioms as it is possible that the violation is due to some unsatisfying precondition axiom.
- `Failed:` A task is considered *failed* when it failed to finish successfully or the user abandoned it in the midway. First, we check precondition axioms attached to this task in the task model and are interested in any violated axioms, as these can be the main reason behind the failure of the task. For example, in the above described example it is possible that the task *deposit*(*amount*) failed because the attached precondition variable *amount* violated the attached precondition axiom to this task in the task model, i.e., the value of depositing *amount* should be greater than zero. Here, we separate the satisfying attached postcondition variables to unsatisfying attached postcondition variables to this task in order to know how much the progress had been done in the task execution when the user decided to leave this task. By analyzing both satisfying and unsatisfying the attached pre- and post-condition axioms we can conclude the reasons for the failure of the task.
- `Avoided:` A task is considered *avoided* when it was in the *execution-path*, but the user did not try to perform it. There can be two cases

for this: in the first case, the user chose an alternative path and the task was not in that chosen alternative execution-path while in the second case, the user chose the defined execution-path but decided not to perform it. In the second case, we can get hint from the system state at that point the reason behind the user decision not to perform this task.

- The postcondition effect axioms attached to system tasks in the task model are also used to check the correctness of system functionalities. Each system task in the task model is a representation of some system function/action. The attached postcondition variables' values are used to tell after comparing them with the attached postcondition effect axioms to this task in the task model whether this system function/action has performed as it was supposed to be. For example, the task *sumValue* in a task model represents a system function "$sum(a, b) = c$"that sums the value of two variables into the third variable. But, if the value of variable c after the execution of system function *sum* violates the postcondition effect axiom of task *sumValue* to this variable in the task model then its mean that the system function is not working properly.

- The users' selection of tasks and the task structure in the task model are used for the analysis of users' execution-path selection behavior; such as the users who selected execution-path as mentioned in the task model and were able to successful finish, the users who selected execution-path as mentioned in the task model but were unable to successful finish, the users who selected some alternative execution-path but were able to successful finish, the users who selected wrong execution-paths, etc.

- The goals in the attached task model help in finding out whether users were able to achieve the targeted goal through some predefined execution-path in the task model or through some alternative execution-paths.

- The task structure in the attached task model also helps finding users' trend, behavior, and pattern for some particular execution-path selection and their performance in each execution-path.

- The analysis of exogenous actions' recorded data is useful to check their effect on system states and on other tasks. We can check whether execution of a particular exogenous action made precondition axioms of some other task in satisfied mode or in unsatisfied mode. This is useful in order to know if the reason behind failure of a task is the execution of some exogenous action.

In addition to the above, the TaMU framework also uses other information such as user groups, their particular preferences, and so forth; which are provided in the attached task models for different data analysis purposes.

5.3.6 The Role of TaMoGolog Formalism in TaMU Framework

The above subsections have described TaMoGolog critical role in TaMU framework for modeling user and system tasks, for recording data during evaluation experiment as per defined mode, and the criteria that uses TaMoGolog-based task models for the automatic analysis of the experiment recorded data. In the forthcoming sections, we provide its realization through our developed TaMUlator tool and its usage in a case study. In Chapter 4, we provided the rationales behind providing task modeling framework and the definition of TaMoGolog task modeling language. Here, we describe briefly the role of TaMoGolog formalization, provided in Chapter 4, in relation to our usability evaluation approach.

The provided TaMoGolog formalism is useful for different purposes from communication amongst members of development team and evaluation team to performing automatic analysis of the recorded data. Firstly, the formalization provides a standard syntax for writing task models and semantics and also to elucidate its meaning within appropriate context. This is also important from communication perspective as it enables the team members (both from development and evaluation) to create the accurate task models of user and system activities and behavior, and it helps in unambiguous understanding of the created task models which reduces the understanding gap.

The provided formalization is especially useful and critical from our usability evaluation perspective. First, the semantics formalism guides the compiler in automated tool for the compilation of the task models written in TaMoGolog. Through this, it is possible to use the same task models in different automated tools if each of these follows the provided semantics specification. After the compiler in an automated tool compiles the task model within the provided specification, the tool records the user and system activities according to the stipulations of the evaluator. Without a formal specification of language syntax and semantics, it can be possible that the evaluator thinks differently from how the tool actually handles it. However, if followed appropriately, a well defined formal specification, as we provided in Chapter 4, the understanding of task models during evaluation experiments will be the same both of for the evaluator and the tool.

As the TaMoGolog semantics is based on transition semantics for single-step execution, so the TaMoGolog-based task models are evaluated at each unit task execution. In our TaMU framework, we define tasks at the code level that can be at any abstraction, e.g., pressing a button in a function can be a

task or the whole function itself can be defined as a task. It helps to record the system states at each execution step as defined by the attached task model during the execution of the evaluation experiment. This enables us to analyze at each step the system state in verisimilitude and for the attached task model. The task model reflects what the system state should be ideally at each point; hence, it enables to find the differences between the ideal state and the actual state during the execution state of experiment. These differences provide a mean to look up the usability problems at those points.

Finally, the external nondeterministic constructs' formal specification helps us to understand users and external systems/applications participation in tasks execution and in making nondeterministic decisions. Through this, it is possible to merge system and user tasks in the same task model so to record the user and the system activities and behavior at the same time. This also provides a mean to record and analyze users' behavior more appropriately, e.g. how many users preferred a particular execution path or how many users picked a particular value for some task.

In the forthcoming sections, first we describe that our developed tool TaMUlator uses the formal specification to compile the TaMoGolog-based task models and then uses these compiled task models to record users and system activities and finally during automatic analysis of the results. Secondly in the case study, we also provide few examples of task models that we used to record user behavior and to highlight usability issues.

5.4 TaMUlator: A Tool for Managing TaMU Process Life-Cycle

We present a tool, called **TaMUlator** (**Ta**sk **M**odel-based **U**sability Eva**luato r**), for managing and automating TaMU process life cycle at the IDE level, i.e., defining the experiments using TaMoGolog-based task models, and running the created experiments. This enables us to record user and system behavior as per the defined mode. TaMUlator is a Java-based tool that provides a set of APIs and interfaces to work at the IDE level, thus fitting in different development environments.

TaMUlator allows the development team to tag *unit* tasks and variables (possible candidates to be used in precondition axioms and postcondition effects) of interest at the code level. It provides an API, called **TTag-API** (**T**ask **Tag**ging **API**). Developers can use it or any other facility, such as AspectJ[2], to tag unit tasks and variables in the code. This tagging facility can be used during any stage of the development process, and does not require any

[2]http://www.eclipse.org/aspectj/

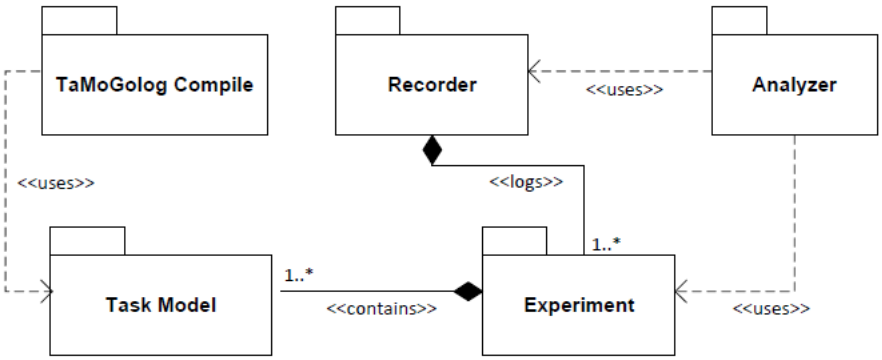


Figure 5.4: TaMUlator high-level modules overview

internal intervention in the program code. TaMUlator provides an easy and dynamic way to define different usability scenarios for the evaluation. This is achieved by compiling TaMoGolog-based task models that can be aggregated into evaluation experiments, which can be evaluated at any time by the built-in *Analyzer* using the recorded data of these experiments, or can be manually evaluated by exporting the recorded data into a CSV (comma-separated values) format for analysis. The analysis results (either automated or manually) help the software development team in drawing conclusions to derive relevant development tasks for further improvements in the developing product.

5.4.1 High-Level Modules Overview

TaMUlator comprises five main modules. Figure 5.4 presents a high-level modules view showing five modules and the relationships between them. The five modules are: *Task-Model* for tagging tasks and variables at the IDE level and keeping task models for each usability scenario; *TaMoGolog-Compiler* for compiling TaMoGolog-based script of task models; *Experiment* for creating and managing evaluation experiments to be performed by evaluating users; *Recorder* for recoding interested data during the execution of evaluation experiment; and *Analyzer* for automatic analysis of the experiment data. Following is description of each module.

TaMoGolog-Compiler: This module is responsible for compiling the TaMoGolog-based script to a *Task-Model* interpretation that can be understood by other modules. The current version supports only a subset of TaMoGolog constructs (i.e., waiting, sequence, nondeterministic internal choice, and nondeterministic external choice) and provides limited support for writing domain knowledge in task models.

Task-Model: This module contains specific usability scenarios that were specified in TaMoGolog-based task models and were created by compiling the scripts in *TaMoGolog-Compiler* module. It provides developers with the opportunity to associate task models with the created evaluation experiments.

Experiment: This module manages evaluation experiments so that evaluating users can perform different tasks on the target application to achieve the desired goals. Each experiment is associated with one or more task models in the *Task-Model* module, and is independently responsible for managing its own task models. This is a self-manageable module, and can be independently enabled and disabled.

Recorder: This module is responsible to record all the activities that were reported to the TaMUlator. While the evaluation experiment executes, each tagged task that was enabled by an evaluating user or the system and the values of variables related to precondition axioms (before executing the task) and postcondition effects (after completing the task) are reported and recorded. The recorded data related to each evaluation experiment can be retrieved and analyzed.

Analyzer: This module takes the evaluation experiment from the *Experiment* module and the related recorded data from the *Recorder* module, and gives feedback after analyzing the data together with the original task models associated with the evaluation experiment.

5.4.2 TaMUlator APIs

TaMUlator provides APIs and interfaces of a Java library to work at the IDE level. Following is brief description of each API:

TTag-API (Task Tagging API): This API provides a set of interfaces, classes, and methods that are used for tagging tasks and variables in the application code. Tagging facility through this can be used during any stage of the development process, and does not require any internal intervention in the program code.

TM-API (Task Model API): This API is provided by the *Task-Model* module. Developers use it to change task model parameters (e.g., task name) and export these task models structures as a tree using the Java JTree[3] class.

[3]http://download.oracle.com/javase/1.4.2/docs/api/javax/swing/JTree.html

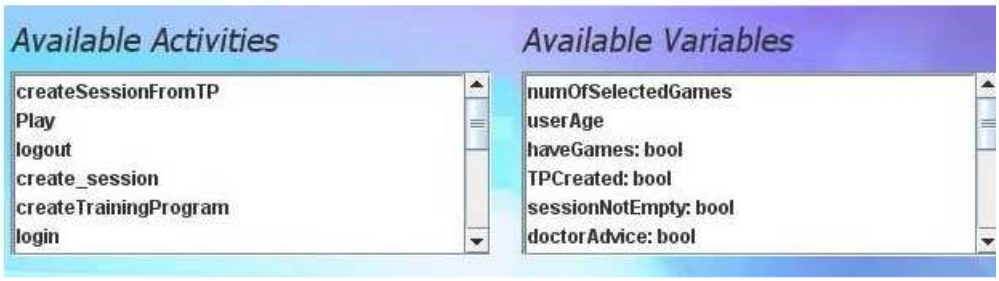


Figure 5.5: Tagged set of tasks (activities) and variables

EExp-API (Evaluation-Experiment API): This API is provided by the *Experiment* module. Developers use it to manage the associated task models inside the *Experiment* module.

5.4.3 How TaMUlator Works

TaMUlator allows the development team to tag tasks (after tagging, the tasks are treated as unit tasks) and variables (possible candidates to be used in precondition axioms and postcondition effects) of interest at the program code level. Only the tagged set of tasks and variables are later available for using in task models. Figure 5.5 shows a TaMUlator screen-shot of available tagged set of tasks (called activities in the TaMUlator environment) and variables for an evaluating application.

To perform tagging, developers need to "wire" their programs in "interesting" locations (for tasks and variables) throughout their code. The TaMUlator tool leaves it to the software team to define those points of interest (e.g., method invocations, object state changes, or certain events). For this, Developers can use the TTag-API, provided by the TaMUlator, or through the AspectJ facility. AspectJ is a Java extension and provides one of the simplest ways in Java-based platforms to enable support for this type of events. It supports the aspect constructs separating cross-cutting concerns from an object-oriented system and provides a mechanism to merge these aspects into the underlying system. Through this, developers can define multiple *pointcuts* and *advices* that inject small snippets to let TaMUlator know that an event has occurred. Figure 5.6 shows a screen-shot of the TaMUlator demonstrating a typical AspectJ hook. In the case of using TTag-API, the TaMUlator receives data as any tagged event occurs or if any tagged variable changes its value.

After developers have "wired" their code to TaMUlator, it is time to define task models for usability scenarios (currently, TaMUlator supports only one task model per scenario). The process of defining task models in TaMUla-

```
pointcut login(LogirManager framework) :
    call(public UserProfile LogirManager.login(..))
        && target(framework);

after(LogirManager framework): login(framework){
    try {
        if (taskModuleLoaded)
            taskModule.activityOccured("login");
    } catch (TaskModuleException e) {
        System.err.println(e.getMessage());
    }
}
```

Figure 5.6: Screenshot of a typical AspectJ hook

Figure 5.7: A task model written in TaMUlator

tor is very easy; simply write a TaMoGolog-based script, using the available set of tagged tasks and variables, that defines the tasks structure with temporal relations in tasks, precondition axioms, and postcondition effects and let TaMUlator compile it through the *Compiler* module. Figure 5.7 shows a TaMUlator screen-shot of a simple task model written in TaMoGolog (Note that the current TaMUlator environment uses `Label name(parameter-list)body` for defining *composite* task or *waiting* task that is in fact equivalent to Golog [73] procedure definition `proc name(parameter-list)body end`).

After compilation, TaMUlator keeps these task models in *Task-Model* module from where they are used in evaluation experiments and for automatic analysis of results. Using TM-API, developers can change compiled task model parameters (e.g., task's name) and export these task models structures as a

tree using the Java JTree[4] class.

During the third phase of TaMU life-cycle, the evaluation team creates evaluation experiments, using TaMUlator, so that evaluating users can perform different tasks on the target application. Each of these experiments contains one or more usability scenarios the evaluation team wishes to track, where each scenario is reflected by already created task model in the previous step. Each experiment is associated with one or more task models in the *Task-Model* module, and is independently responsible for managing its own task models.

During the next phase, while the evaluation experiment is executed by the evaluating user, each tagged task (a *task* is also called an ***activity*** in TaMUlator environment) that was enabled by the user, and the values of variables related to precondition axioms and postcondition effects (before executing the task and after completing the task) are reported and recorded by TaMUlator. In the current environment of TaMUlator, any task record comprises the following 4-tuple: (name, time-stamp, precondition axioms' status, task-model set[5]), while variable records for checking postconditions comprise a 3-tuple: (name, new value, time-stamp).

TaMUlator provides a built-in automatic *Analyzer* to evaluate the experiment results after comparing with associated task models, or can export the recorded data into a CSV format for manual analysis. At any time, the development team/evaluator can issue an analysis of the recorded data against any task model or experiment. Figure 5.8 shows a TaMUlator screen-shot of *Analyzer* output.

5.4.4 The *Analyzer* Module

TaMUlator provides the *Analyzer* module to analyze the recorded data it has collected when users performed evaluation experiments. The development team/evaluator can directly view the information related to a specific task model or experiment after retrieving the recorded data in a CSV format from the *Recorder* module, or can ask the *Analyzer* to provide results after analyzing the data automatically according to the analysis criteria described in Section 5.3.5.

The *Analyzer* compares and analyzes the structure of the attached task models to the recorded data. Thus, the *Analyzer* checks whether the recorded data is consistent with the task structure, by checking that the appearance of the recorded tasks are in the same order as in the task structure (for checking *execution-path* selection), making sure that precondition axioms were met and postcondition variables possess the desired values. If any of these conditions are not met, the scenario for the task model is considered as not properly

[4] http://download.oracle.com/javase/6/docs/api/javax/swing/JTree.html

[5] All task models that have the mentioned activity.

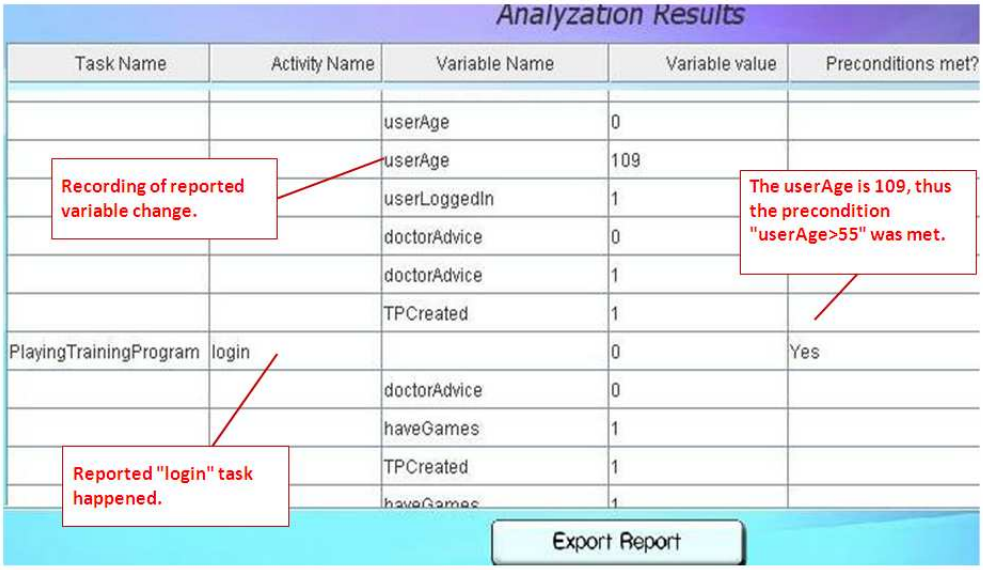


Figure 5.8: Analyzer output of an evaluation experiment

fulfilled, compared to what the evaluator wanted. This helps the development team/evaluator to find usability issues in the targeted application, and is especially useful for locating the points where users normally make mistakes, so the development team can take care of those places in next development iteration.

5.4.5 TaMUlator Evaluation Case Study

We present a case study in which six development teams (composed of 6-7 students each) used TaMUlator to evaluate the software project they developed[6]. We gathered the data using the course material, e.g., product requirements and exercises; project artifacts, e.g., task models and experiments' results; and written material by team members, e.g., communication in web forums, feedback on roles, and the final course retrospective. We analyzed the data using a qualitative comparison of triangulated data sources.

The project, named "Brain Fitness-Room", aims to develop a system that supports maintaining and strengthening memory and brain capabilities as well as identifying any decline in these capabilities. The motivation for such a system includes retaining and improving brain capabilities, detecting brain illness, specifically dementia syndrome [103], slowing down the progress of

[6]The team members were 4th year CS-major students participating in the "annual project in software engineering" course of the Computer Science Department at Technion, IIT.

known and unknown afflictions, and preventing brain illnesses. The system includes a pool of games that fits in with its goals, thus providing, among other goals, fun for the users and enabling the collection of data for future studies.
The main components were defined as follows:

- The application provides three types of user interfaces for three types of users: players, doctors, and administrators. Players can log in, play a session or a specific game, view history, or get advice.
- A session builder can be used to create a session for the player in which he/she plays and is advised.
- A game pool is provided. The games can be run on the system and can be added separately. A standard is required so all games can run on all systems (belonging to the six teams). The two game types:
  - Left hemisphere medium/long-term memory game; e.g., study a list of random words for a few minutes, then, after half an hour, write down as many as you remember.
  - Right hemisphere short-term memory game; e.g., study a random shape for one minute, then draw it from memory.
- An automated built-in system advisor can issue warnings and suggestions by analyzing the collected data.

History and statistics are stored in a database and can be viewed upon request.

In addition to the above-mentioned requirements, the subject of usability evaluation using formal task models was presented to the teams. In the first development iteration (see forthcoming subsections), all teams had to develop a tool to enable writing basic task models in TaMoGolog. An end-to-end scenario was defined, to develop evaluation experiments based on TaMoGolog-based task models using an editor, execute them while recording user and system activities and behavior, and analyze the results based on a comparison between the attached task model and the recorded data while performing the experiment. The following task model shows a scenario to evaluate session execution that was provided to the development teams as their first TaMoGolog-based task model.

```
1  TaskModel(PlayingSession).
2  // unit tasks:
3  UnitTask(Activate-Session).
4  UnitTask(Game-Playing).
5  UnitTask(Game-Stopping).
```

```
6  // composite tasks:
7  CompositeTask(PlayingSession).
8  // fluents (Variables):
9  Fluent(setOfGameForCurrentUser).
10 Fluent(currentGame).
11 // precondition axioms
12 Precondition-axioms(Activate-Session) ≡
13                   setOfGameForCurrentUser ≠ empty.
14 Precondition-axioms(Game-Playing) ≡
15                   currentGame ∈ setOfGameForCurrentUser.
16 Precondition-axioms(Game-Stopping) ≡ TRUE.
17 // task model main structure:
18 Label PlayingSession ( ){
19    Activate-Session;
20    Game-Playing;
21    Game-Stopping
22 }
```

Based on the teams' work, one tool was selected and all teams used this selected tool (TaMUlator) during two iterations of evaluation.

Development Method

Here, we describe the development method we used while working with the teams. The approach is based on agile development [2, 14] and is presented using three main perspectives: human/social (H), organizational (O), and technical (T). More information about the HOT framework that provides case analysis using these three perspectives can be found in [49]. As part of the human/social perspective, the main ideas we foster are teamwork, collaboration, and reflection. Teams meet every week for a four-hour compulsory meeting in which they communicate regarding the product development and the process management. Periodically, team members reflect on their activities.

As part of the organizational perspective, the project was defined for two releases. Each was composed of two iterations of three weeks; i.e., four development iterations. Roles were ascribed to each of the team members as part of his/her team management; e.g., in charge of unit testing, tracking, designing (see more details regarding the role scheme at [27]). Each of the role holders presented measure(s) for the relevant responsibilities.

As part of the technical perspective, the following practices were used: automated testing, continuous integration, and refactoring to ensure simple design. The role scheme supported these practices by emphasizing the appropriate practices for specific iterations and changing the role scheme accordingly

in other iterations. For instance, the person in charge of continuous integration worked mainly at the first iteration to provide the infrastructure and work procedure; refactoring activities were the responsibility of the designer at the third iteration, and so on.

The usability evaluation of the product that is being developed is yet another practice that was implemented as part of this project. Based on the experience with guiding the implementation of the agile approach [27, 49, 114], and the integration of User Center Design (UCD) techniques in the last four years in agile projects in the industry and academia [30, 49], following are the main practices we used (also part of our three-fold integration framework described in Chapter 3):

1. Iterative design activities that include cycles of development, which contain development tasks that were derived from usability evaluation.
2. Role holders in the subject of usability evaluation and using TaMUlator.
3. Measurements that were taken by the role holders as part of fulfilling their responsibilities.

5.4.6 Evaluating TaMUlator

TaMUlator was used in the third and fourth iterations to evaluate the system that was developed. Here, we present some of the evaluating scenarios and results from the fourth iteration when teams were more experienced with using TaMUlator and with usability evaluation in general.

Example Task Models

The following TaMoGolog-based task models are examples of the use of TaMUlator for usability evaluation and for functionality evaluation of the Brain Fitness Room product:

(i). *Evaluate the way advice is viewed by the player.*

```
1 Label Main ( ){
2   PatientLogin;
3   Selection;
    PatientLogout
4 }
5 Label Selection ( ){
6   [ (AdvisroStateIsNormal)? ; ViewNormalAdvice |
7     (AdvisroStateIsWarning)? ; ViewWarningAdvice |
8     (AdvisroStateIsDanger)? ; ViewDangerAdvice ]
9 }
```

(ii). *Evaluate time spent in different brain-games.*

```
1  Precondition-axioms(Login) ≡ numberOfSession = 5.
2  Label Session ( ){
3   Login;
4    ActivateSession;
5    Game1; Game2; Game3;
6    SessionEnded;
7    Logout
8  }
9  Label Game1 ( ){
10   SessionGamePalying; GameStopping
11 }
12 Label Game2 ( ){
13   SessionGamePalying; GameStopping
14 }
15 Label Game3 ( ){
16   SessionGamePalying; GameStopping
17 }
```

(iii). *Evaluate user behavior using two task models.*

```
1  Precondition-axioms(Login) ≡ userAge > 55.
2  Label Behavior-Scenario ( ){
3    [user (Bad-Scenario | Good-Scenario)]
4  }
5  Label Bad-Scenario ( ){
6   Login;
7    CreateTrainingProgram;
8    [(haveGame)? ; FreeSession];
9    Play
10 }
11 Label Good-Scenario ( ){
12   Login;
13   CreateTrainingProgram;
14   [(haveGame)? ; CreatSessionFromTP];
15   Play
16 }
```

Activity	**Average Time (sec)**
Login until logout for regular user	412.55
Login until logout for doctor	56.88
Login until start of game session	63.77
Average game time for all games	74.38
Bird game average play time	43.66
Piano Kombat average play time	121.22
Pirate Memory average play time	107.83
Silly Color game average play time	24.8

Table 5.1: Example task model no. 2 - time measures

(iv). *Evaluate the correct flow of session creation.*

```
1 Precondition-axioms(Login) ≡ userAge > 55.
2 Precondition-axioms(AddGameToSession) ≡ TRUE.
3 Precondition-axioms(Play) ≡ gameInSession > 0.
4 Precondition-axioms(Logout) ≡ TRUE.
5 Postcondition-axioms(AddGameToSession, gameInSession, TRUE) ≡
6                                              gameInSession + 1.
7 Label Main ( ){
8   Login;
9    AddGameToSession; Play;
10   Logout
11 }
```

Using Task Models to Analyze User Experience

Given the task model definition, the evaluator can define and run evaluation experiments. As long as the experiments are activated, user and system activities are recorded and stored. The user and system behavior are analyzed by comparing the recorded behavior to the task model for a specific scenario.

Here, we present the analysis of an example task model, which is an example of usability evaluation through time spent in different situations when the precondition of five active sessions is met (see above example task model no. 2). Four measures were derived from the user experience: *a*) Time elapsed between login and first play activation (as one indicator of the usability of the main menu), *b*) Time spent on each game and whether the user completes the game, *c*) Average session time, and *d*) Average time in the system. The summary of results is shown in Table 5.1, Figure 5.9, and Figure 5.10.

Based on these results, the development team reached several conclusions and suggested several development tasks accordingly. Following are two ex-

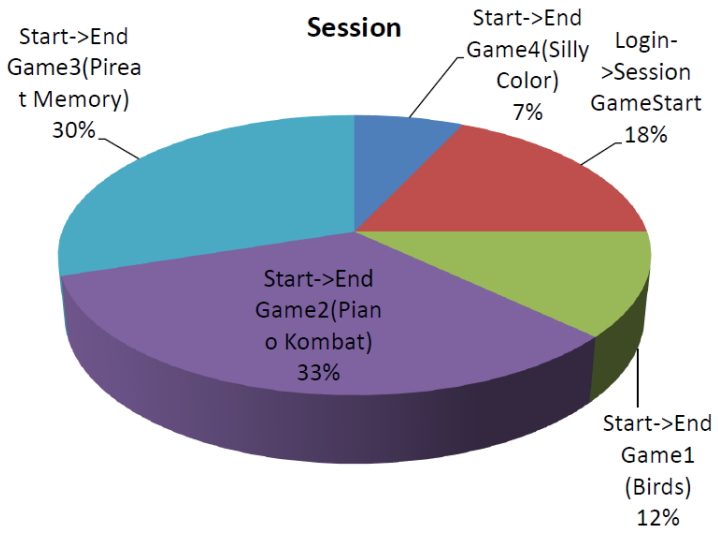


Figure 5.9: Time spent game playing

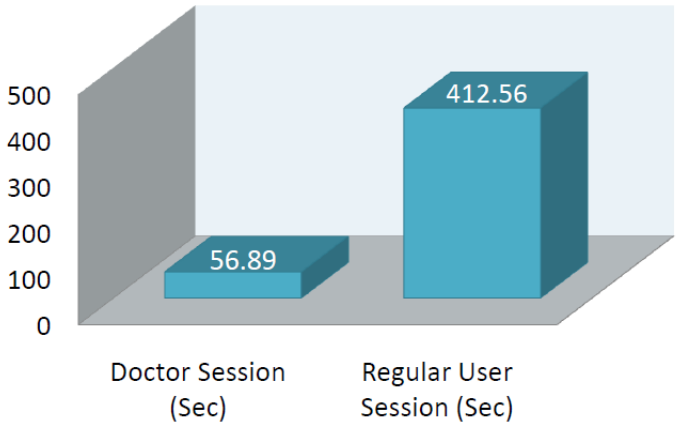


Figure 5.10: Time spent on users' sessions

amples: 1) The average time from login until start of game session (63.77 seconds) is relatively high since all that was required from a new user (player) is to select the default session and start playing. The team suggested changing the user interface in a way that it will be clear that you cannot press the 'play' button before selecting a session. This included a new arrangement of the 'session'and 'play' buttons and a change in the usage of the 'session' check box in cases the user needs the default session. 2) The average play time of the Silly Color game (24.8 seconds) is too short. The team suggested adding a higher difficulty level to increase game effectiveness.

We presented the automated analysis when a precondition is met and time is measured. Automated analysis in the case of the other example task models yields different types of results, e.g., in example task model no. 1) we receive a measure for how many times a user viewed advices in general and specific advices (normal, warning, danger) in particular. This is automatically compared with the behavior of other users thus enables reaching a comprehensive conclusion.

In the final retrospective on the course, team members were asked to grade their satisfaction between 1 to 5 (very satisfied) with respect to the project topic, course methodology (agile, time management and early detection of problems, emphasis on testing and usability), tools that were used (Trac and Moodle), and the services in the physical lab they worked in. 32 team members answered and the average grade for the methodology was high (4.09) (for project topic 4.36, tools 3.98/3.28 respectively, and for lab services 2.66). Specifically, regarding the roles that concerned with usability, team members referred to the importance of learning and dealing with usability while developing. Following are some of their comments on this matter: *"It is important to get feedback from the users..."*, *"It does not matter how good the product is, [people] will use it only if it is simple and user friendly. A lot of things that seem clear to developers are not clear to the end users"*, and *"The role of being in charge of the evaluation experiment was an important role with which we specified the usage of our system by the user"*.

The limitations of this case study is that it was done as part of a project in which the TaMUlator was developed hence students were aware of the benefits and wanted to succeed in using it.

5.5 Related Work

Automating usability evaluation is not a new phenomenon. Plenty of methodologies have evolved and suggestions have been given to automate the evaluation process in software development life cycles to reduce time and cost and to get feedback more effectively and efficiently. Ivory and Hearst [63]

conducted a very detailed survey on the state-of-the-art in the automation of usability evaluation techniques at different levels. They analyzed 132 evaluation techniques, both for Web and WIMP (Windows, Icons, Pointer, and Mouse) interfaces, and found that only 33% of those techniques are supported by automated tools using the taxonomy (*none*, *capture*, *analysis*, and *critique*) suggested by Balbo [6] to distinguish the level of automation, supported by these tools. The survey concluded that there is a great under-exploration of usability evaluation methods automation and suggested focusing the research on automation techniques.

The task model-based usability evaluation area is still underexplored as there are only a handful of tools available that support and automate the process of usability evaluation through task models at different abstraction levels.

ConcurTaskTrees (CCT) [92] is the most widely used technique for writing task models in model-based usability evaluation. It provides graphical representation for different abstraction of tasks through a hierarchical-based task tree, and specifies temporal relationships between tasks and sub-tasks using operators based on LOTOS [118] formal notations. It supports four kinds of tasks abstraction, called *user* tasks, *abstract* tasks, *interaction* tasks, and *application* tasks. This representation technique is supported by CCTE (ConcurTaskTrees Environment), a tool for creating task trees and building the relationship between different sub-tasks in the task-tree according to the semantics of task model [80]. The created task model can be saved in a different format, e.g., as a JPEG image or in XML format. The tool also provides a simulator environment for better analyzing the dynamic behavior of created task models. Sinnig et al. [111] enhanced the set of temporal operators of CCT (by adding *stop*, *nondeterministic choice*, *deterministic choice*, and *instance iteration* operators), and also provided a concept for expressing a special kind of cooperative task model that distinguishes the different roles and the actors who perform those roles.

The first automated usability evaluation tool based on CCT is USINE (USability INterface Evaluator), developed at CNUCE-NCE (Pisa) [71]. For input, USINE takes the task models (generated in XML format in CCT) and the logs generated by users (through the log recording tool "Replay"), and then the designer creates a log-task table in USINE for mapping physical actions performed by users to the basic tasks of the task model. USINE then creates the precondition table automatically, emphasizing the possibility of doing one task before others. When users perform tasks, the tool takes the users' log files and gives the result after analyzing the user data with the task model, log-task table, and precondition table. This whole process is divided into three phases: the *preparation* phase, the *automatic analysis* phase, and the *evaluation* phase [71].

RemUSINE (Remote USINE) [91] is an extension of USINE and provides the support of remote usability evaluation, an enhanced methodology, and the facility to analyze a large set of usability data. A recent enhancement is the MultiDevice RemUSINE tool [93] for mobile applications that includes the possibility of detecting those environment conditions that could affect users' interaction with mobile applications such as battery level consumption, data exchange rate, disconnection problems, and the surrounding environment.

AWUSA (Automated Website USability Analyzer) [10] focuses on remote task-based usability evaluation and targets websites rather than system applications. It works on already built websites, so it is the evaluator/architect's duty to discover task models of the target website. It takes three inputs: the website, the discovered task structure (which defines the business model of the website), and the captured logging information of users (generated by the website server).

ReModEl (REmote MODel-based EvaLuation) [10] has a client-server architecture for remote usability evaluation where the server contains task models (using the concept of CCT) and the targeted task models are delivered to the client via a corresponding graphical user interface. The designer creates a dialog graph (for the specific device, e.g., PDA or notebook) based on the server-side task models, to provide a corresponding concrete user interface. On the client side, the system captures users' interactions with the proposed user interface and returns them to the server, which analyzes them with the created task models. So the AWUSA applies reverse engineering for usability evaluation of the web site while ReModEl uses forward engineering by transforming task models to multi-model interfaces for usability evaluation of the target application.

DiaTask [101] is used to develop a dialog graph to represent the navigation structure of the application, based on the already specified task model. This tool creates the first prototype of the application and helps build the requirements properly as it analyzes the results when users interact with these prototypes. The approach has been further enhanced in [97] for usability evaluations of smart environments.

5.6 Summary and Conclusions

In this chapter, we presented a framework, called TaMU, for managing and automating task model-based usability evaluation in software development environments. The framework uses TaMoGolog-based task models to record users and application data while evaluating the target application, and a mean for the automatic analysis of the recorded data to highlight usability issues.

We also presented TaMUlator tool that manages and automates end-to-end TaMU process life cycle at the IDE level, and provides the automatic analysis of the users and the application behavior. We also presented a case study where six development teams used our framework to evaluate their developing product.

Our TaMU framework approach for usability evaluation differs from previous approaches in many aspects.

Firstly, TaMU framework considers *usability evaluation* for the evaluation of both product *usability* and *functionality*, and uses *experiments* to find usability issues and system problems that serve as a kind of acceptance test for the developed features. The other approaches simply focus on usability aspects during performing evaluation experiments. As TaMU framework focuses both on product usability and functionality during evaluation experiments thus also works as acceptance testing to the product functionalities along finding usability issues, hence it is not required to perform a complete testing separately, which saves time and cost in the long run. Due to these reasons, TaMU framework fits very well in agile nature of development processes where there are normally short-time natures of development iterations.

Secondly, TaMU framework emphasizes the automation and management of usability evaluation at the IDE level to integrate it fully with the development process as described previously in Chapter 3. This integration at the IDE level enables the software development team with the mechanisms to monitor and control a continuous evaluation process tightly coupled with the development process, thus receiving ongoing user feedback and product functionality testing while continuing development. This also enables us to derive relevant development tasks for the forthcoming iterations for further improvements in the developing product. Due to this, TaMU framework can be worked very effectively in iterative and incremental type of development approaches.

Thirdly, TaMU framework approach suggests tagging tasks and variables at the program code level in the development environment. Through this, it is possible to model both the user and the system tasks in the task models attached to evaluation experiments. This enables us to capture both the user and the system activities and behavior during the evaluation experiment for finding usability issues and testing product functionalities. Also, tagging tasks at the code level enables us to abstract our *task* notion at different abstraction levels, e.g., we can define one computation step as a task or can enhance its abstraction by defining a complete functionality as a task. This task tagging at different abstractions is used to evaluate product usability and functionality at different abstraction levels; hence, helps in finding usability issues from several aspects.

Fourthly, the definition of a well-defined (syntactically and semantically) formal language, TaMoGolog, for modeling usability scenarios in evaluation

experiments and then using it for the automated analysis of the recorded data. We have already described in details the benefits of using TaMoGolog for usability evaluation in Section 5.3.2. In brief, the benefits of TaMoGolog for usability evaluation that were lacking in previous approaches are: the powerful set of operators for constructing task structures for complex system behavior unambiguously and accurately, the extendable and customizable nature of the language through the predicate system of situation calculus, provision of user and system tasks in the same task model or in separate task models that helps to model and capture their activities and behavior together or separately, the precondition axioms for task include all those conditions that must be true in order to execute the related task are extremely useful in finding out the reasons behind the failure of the task and to highlight the related usability issues, the task postcondition effect axioms to variables enable to test product functionalities, and the facility to write domain knowledge in the task model can also help in highlighting usability issues more effectively and accurately.

Chapter 6

Conclusions and Future Directions

6.1 Conclusions

The theme of this thesis is incorporating user experience as part of the development process. This is achieved through involving end users in the evaluation process of a specific product so as to collect their feedback and then to manage the ensuring development accordingly for enabling high-level usability in the end product. We broke down this theme into three levels, tackled one by one from high level to low level. Firstly, to overcome the gap between software development practice and user experience, we provided a three-fold integration framework that incorporates user-centered design (UCD) philosophy into agile software development at three levels: (*i*) the process life-cycle level, (*ii*) the iteration level, and (*iii*) the development-environment level. Secondly, we targeted our focus towards the development-environment level integration to tackle the challenges of *UCD management* from within the integrated development environment (IDE). This helps the software development team to automatically collect users and system behavior and feedback to recognize usability flaws and errors in an efficient way. Thirdly, we focused towards automating task model-based usability evaluation through recording the user and system activities and behavior as per the defined mode with the help of TaMoGolog-based formal task models. This enables the process to analyze automatically the results by comparing the task models and the recorded data to highlight usability issues in an effective manner, and to draw conclusions to derive relevant design and development tasks for further improvements in the developing product. Figure 6.1 shows the hierarchy of our targeted three levels.

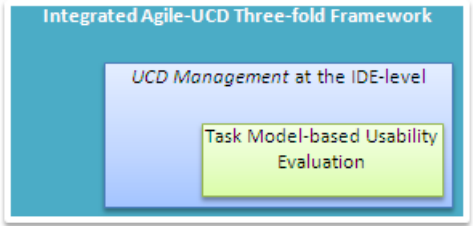


Figure 6.1: Hierarchy of thesis targeted areas

Firstly, we provided the preliminary background knowledge about the four areas related to the thesis. These four areas are the agile development, the user-centered design (UCD) philosophy, the usability evaluation approach, and the task analysis.

We presented our three-fold framework for utilizing the benefits of user-centered design philosophy while developing software projects with agile approach. We suggested a number of attributes for selecting appropriate UCD methods during different phases of development. We suggested a life-cycle for performing UCD activities alongside agile development activities. We also provided suggestions for aligning UCD concepts, roles, and activities within the development iteration activities for maximum benefits. We further discussed other approaches and techniques that integrate user-centered design in agile development at different granularities.

We provided the concept of *UCD management* for managing and automating UCD activities at the IDE level. This mechanism enables the software development team to monitor and control a continuous evaluation process, thus receiving ongoing user feedback while continuing development.

We presented a tool UEMan that enables the software development team to manage and automate the UCD activities at the IDE level alongside the development activities. We also presented two case studies where in the first case study the UEMan was evaluated by using itself, while in the second case study six software development teams used it to evaluate the software projects they developed.

We presented a framework for task modeling. This framework provides a set of concepts; i.e., *task*, *task type*, *view-type*, *task-model*, and *view-model*; for laying out a conceptual foundation to structure task models from different abstractions. We categorized task types into two categories: the *basic* category and the *behavioral* category. We provided definition of sets of task types and view types. We also presented an example to further elucidate our task modeling framework concepts.

We provided the definition of a formal task modeling language, called TaMoGolog, on the top of the foundations of Golog-family [18, 19, 20, 73, 107] of

high-level programming language for constructing dynamic and rich task models. We provided the formalization details of the TaMoGolog. For this, firstly we presented the TaMoGolog set of constructs. Secondly, we provided the formal syntax and semantics framework of TaMoGolog. Thirdly, we defined the formalization of the framewrok for external nondeterministic constructs at the language level using GameGolog [20] approach. This enables to model the external entities' participation in making nondeterministic decisions at run time. Fourthly, we provided the low level implementation of the framework for external nondeterministic constructs in the Golog-family Prolog-based interpreter `P-INDIGOLOG` [105].

For the automatic analysis of the evaluation, task model-based usability evaluation approach uses formal task models to model the user and the system tasks and behavior. These task models are then used as a mean for producing automatic analysis of the recorded users' and system data. We presented a framework, called TaMU framework, for managing and automating task model-based usability evaluation in software development environments. We described the role of TaMoGolog as task modeling language in TaMU framework. We explained the effects of tagging tasks and variables at the code level, benefits of TaMoGolog for usability evaluation, the reflection of TaMoGolog-based task models through scenarios in evaluation experiments, the role of TaMoGolog-based task models in recording users and system activities and behavior during execution of experiments, and the process of performing automated analysis with the help of TaMoGolog-based task models.

Finally, we presented TaMUlator tool that manages and automates an end-to-end TaMU evaluation life-cycle at the IDE level, and provides the automatic analysis of the user and the system behavior. Also, we presented a case study where six development teams used our TaMU framework and TaMUlator tool to evaluate their developing product. Moreover, we discussed other techniques and tools that perform task model-based usability evaluation and highlighted the differences of our approach from these.

6.2 Future Directions

The thesis provides foundations on which a number of future research directions arise. In this thesis, we conducted our research work towards three dimensions aims at achieving the targeted theme. These three dimensions are: incorporating UCD philosophy into agile software development approach, managing and automating UCD activities and usability evaluation at the IDE level, and defining a task modeling framework and a formal task modeling language to support the automated usability evaluation analysis process. Here, we are summarizing the possible future directions:

Towards the integrated framework approach:

- The application of proposed framework in small- to large-scale software projects within the industry to evaluate its effectiveness. One of the challenges in this regard is to perform empirical evaluation studies of the framework for projects of different scales.
- The enhancement in the framework to accommodate changes that can make it applicable for other iterative and incremental natured software development processes.

Towards management and automation of UCD activities and usability evaluation:

- An evaluation study of the TaMU framework for estimating the efforts required for utilizing it (e.g., time and efforts required for tagging in the code, creating task models, conducting experiments, etc) and for assessing the effectiveness of it for capturing usability issues and different human behaviors.
- Detailed empirical comparative study of TaMU framework with traditional usability evaluation methods for checking its efficiency and effectiveness.
- The support for a wide range of UCD activities and usability evaluation methods at the automated tool level.
- The complete support of TaMoGolog language in TaMUlator tool as currently it only implements a subset of TaMoGolog functionalities. One future direction would be integrating these two tools (TaMUlatore and UEMan) to provide all the relevant functionalities under one umbrella.
- The current TaMUlator automated analysis covers parts of the usability issues. To cover a wide range of usability issues, there is need to enhance the TaMUlator *Analyzer* module to cover fully the analysis criteria suggested in Section 5.3.5. An important objective is to provide the results with help of robust statistical techniques in order to highlight usability issues more accurately and to recognize/analyze different users' and system behavioral patterns.
- The support of automatic *critique* [6] in the usability evaluation framework (and in the resulting automated tool); which provides the facility to automatically suggest the possible solutions, recommendations, and improvements after analyzing the usability issues. This would be very useful for working with agile development to save time and cost.

- Providing a simulation environment where Golog-family based autonomous agents would act like end users of the developing/developed product for participating in evaluation experiments. This kind of simulation environment can give a large amount of evaluation data in short-time, and also saves cost as conducting usability with real end users incurs cost and time.

Towards task modeling framework and TaMoGolog:

- Task modeling techniques are used in other areas such as in software engineering to help designers analyze and develop their system designs. For example, Reichart et al. [101] suggest transformational (pattern-guided) model-based development for interactive systems, using task models amongst other models. There is need to analyze the usage of proposed task modeling framework and TaMoGolog language in other areas, e.g. for collection of requirements or for model-driven development purposes.
- Providing the support in TaMoGolog towards constructing task models for collaborative environments.
- The graphical representation of TaMoGolog language and a task modeling environment. This task modeling environment would allow writing task models in textual or graphical form and exporting the created task models in different formats.

Most of the above possible future research directions target only one of the three dimensions, while few of these target a combination of two or all of the three dimensions.

Appendix A

Golog-family based High-level Program Syntax

This appendix provides implementation of the external nondeterministic constructs framework in Golog-family based high-level program syntax. We are not going to domain specific low-level details. We provide here the set of axioms for predicates, fluents, and actions; and provide the procedure definition for each of those constructs that can be directly transferred into Golog-family based programs as discussed in Section 4.6.

Predicates for Agents

First, we provide predicates for defining agents (e.g., external application/ systems).

Agent (*agt*)
SysAgent (*sys*)

The *Agent* predicate is for defining external agents (applications/systems) that participate in making nondeterministic decisions, while the predicate *SysAgent* is to define the system (or the user program) name that controls the agent environment when no external agent has control over it.

Primary Actions

The following primary actions are used by the system for managing the control of agent environment and for sending the requests to external agents.

startControl(*agt*): gives control to agent *agt*
endControl(*agt*): takes control from agent *agt*
requestExtCh(*agt*, $\vec{list}$): requests to agent *agt* to make a choice and the $\vec{list}$ contains left and right program parts

requestItr(agt): requests to agent agt to make decision on iteration
requestPick(agt): requests to agent agt to pick a binding for the variable x
requestNum(agt): requests to agent agt for making decision that how many instances of the program the agent wants in the concurrent iteration

The first two actions manage agent environment, while the purpose of remaining actions is to give required signals to the agent for making the nondeterministic choice decision. It is also possible to send other information to agents during request by adding parameters in the requesting actions in order to understand the request properly.

Exogenous Action

The following choice actions are generated by agents, through which they let the system know of their decisions. In ConGolog, actions generated by external applications/systems are regarded as exogenous actions.

$agtLeft(agt)$: agent agt chooses left-sided choice
$agtRight(agt)$: agent agt chooses right-sided choice
$agtContinue(agt)$: agent agt chooses to continue the iteration
$agtStop(agt)$: agent agt chooses to stop the iteration
$agtPick(agt, x)$: agent agt gives a value for the variable x
$agtNumber(agt, num)$: agent agt gives the number for the concurrent iteration

Defined Fluents

$sysAgt(s) = sys \stackrel{\text{def}}{=} SysAgent(sys)$
The *sysAgt* fluent provides the name of the system (or user program) that controls the agent environment when no external agent controls it.

Functional and Relational Fluents

We divide fluents into three categories: the first category manages the control of the agent environment, the second category checks whether the agent has made the decision, while the third category deals with the decisions of agents.

- *Fluents related to controlling the agent environment are:*

  $Free(s)$: provides whether the agent environment is occupied or not
  $controlAgt(s) = agt$: gives the name of agent who currently controls the agent environment
  $AgtEnv(agt, s)$: given the agent name, it tells whether this agent currently controls the agent environment or not

- *The second category contains:*

  $ExtChSelected(s)$: becomes true when the agent selects the choice
  $ItrSelected(s)$: becomes true when the agent makes the iteration decision
  $PickSelected(s)$: becomes true when the agent picks a binding for the variable x
  $NumSelected(s)$: becomes true when the agent selects the number for the concurrent iteration

  These fluents becomes true when the agent makes some decision based on request. The agent generates an exogenous action in response to the request and the related fluent becomes true, which is used to make further decisions.

- *The third category contains:*

  $Left(s)$: becomes true if the agent selects left-sided program
  $Right(s)$: becomes true if the agent selects right-sided program
  $Continue(s)$: becomes true if the agent decides to continue the iteration
  $Stop(s)$: becomes true if the agent decides to stop the iteration
  $pickMade(agt, s) = x$: contains the binding for the variable x provided by agent
  $numMade(agt, s) = num$: contains the number provided by agent for the concurrent iteration

  These fluents becomes true or contain values according to the decisions taken by agents. These change their values when agents generate exogenous actions in order to let the system know their decisions.

Initial Values of Fluents

The following are values of fluents at the initial history when no action has been executed so far. This state is represented by S_0 in Golog-family. Initially, *Free* and *Stop* have value true, while others relational fluents have false value. On the other side, initially the controlling agent name is the system (or user program) name. The last axiom says that initially all agents have no control over the agent environment.

$Free(s)$
$controlAgt(s) = sysAgent$
$\neg ExtChSelected(s)$

$\neg PickSelected(s)$
$\neg NumSelected(s)$
$\neg Left(s)$
$\neg Right(s)$
$\neg Continue(s)$
$Stop(s)$
$\forall agt.Agent(agt) \wedge pickMade(agt, s) = nil$
$\forall agt.Agent(agt) \wedge numMade(agt, s) = nil$
$\forall agt.Agent(agt) \wedge \neg AgtEnv(agt, s)$

Precondition Axioms for Actions

The following are precondition axioms for each primary and exogenous action.

$Poss(startControl(agt), s) \equiv Free(s) \wedge \neg AgtEnv(agt, s) \wedge Agent(agt)$
$Poss(endControl(agt), s) \equiv \neg Free(s) \wedge AgtEnv(agt, s) \wedge Agent(agt)$
$Poss(requestExtCh(agt, \overrightarrow{list}), s) \equiv AgtEnv(agt, s) \wedge \neg(\overrightarrow{list} = \jmath)$
$Poss(requestPick(agt), s) \equiv AgtEnv(agt, s)$
$Poss(requestItr(agt), s) \equiv AgtEnv(agt, s)$
$Poss(requestNum(agt), s) \equiv AgtEnv(agt, s)$
$Poss(agtLeft(agt), s) \equiv AgtEnv(agt, s)$
$Poss(agtRight(agt), s) \equiv AgtEnv(agt, s)$
$Poss(agtContinue(agt), s) \equiv AgtEnv(agt, s)$
$Poss(agtStop(agt), s) \equiv AgtEnv(agt, s)$
$Poss(agtPick(agt, x), s) \equiv AgtEnv(agt, s)$
$Poss(agtNumber(agt, num), s) \equiv AgtEnv(agt, s) \wedge num > 0$

As stated, the *startControl* is possible if the agent environment is free and currently the requested agent does not control the environment, while in order to take the control back from the agent the reverse is the precondition. For all other actions, the precondition is that the agent, receiving the request or that making the decision, must have control over the agent environment. The precondition for the *agtNumber* choice action also checks whether the given number is zero or greater than zero as iteration cannot be of a negative number.

Effect Axioms for Relational Fluents

Here are the effect axioms for each relational fluent, as stated in [102], in order to describe under which actions' execution these fluents change their value to true or false through situation calculus formulas $\Upsilon_F^+(\vec{x}, \alpha, s)$ and $\neg\Upsilon_F^-(\vec{x}, \alpha, s)$.

$Free(do(\exists agt.endControl(agt), s))$

$\neg Free(do(\exists agt.startControl(agt), s))$
$AgtEnv(agt, do(startControl(agt), s))$
$\neg AgtEnv(agt, do(endControl(agt), s))$
$ExtChSelected(do(\exists agt.agtLeft(agt), s))$
$ExtChSelected(do(\exists agt.agtRight(agt), s))$
$\neg ExtChSelect(do(\exists agt, \overrightarrow{list}.requestExtCh(agt, \overrightarrow{list}), s))$
$ItrSelected(do(\exists agt.agtContinue(agt), s))$
$ItrSelected(do(\exists agt.agtStop(agt), s))$
$\neg ItrSelect(do(\exists agt.requestItr(agt), s))$
$PickSelected(do(\exists agt, x.agtPick(agt, x), s))$
$\neg PickSelect(do(\exists agt.requestPick(agt), s))$
$NumSelected(do(\exists agt, num.agtNumber(agt, num), s))$
$\neg NumSelected(do(\exists agt.requestNum(agt), s))$
$Left(do(\exists agt.agtLeft(agt), s))$
$\neg Left(do(\exists agt.agtRight(agt), s))$
$Right(do(\exists agt.agtRight(agt), s))$
$\neg Right(do(\exists agt.agtLeft(agt), s))$
$Continue(do(\exists agt.agtContinue(agt), s))$
$\neg Continue(do(\exists agt.agtStop(agt), s))$
$Stop(do(\exists agt.agtStop(agt), s))$
$\neg Stop(do(agt.agtContinue(agt), s))$

Successor State Axioms for Fluents

The following are successor state axioms for each fluent (relational and functional) while using situation calculus formula $F(\vec{x}, do(\alpha, s)) \Leftrightarrow \Phi_F(\vec{x}, do(\alpha, s), s)$ as described in [102].

$Free(do(a, s)) \equiv$
$\exists agt.(a = endControl(agt)) \vee (Free(s) \neg(\exists agt.a = startControl(agt)))$

$AgtEnv(agt, do(a, s)) \equiv$
$a = startControl(agt) \vee (AgtEv(agt, s) \wedge a \neq endControl(agt))$

$controlAgt(do(a, s)) = agt \equiv$
$a = startControl(agt) \vee (\exists agt', a = endControl(agt') \wedge agt = sysAgt(s)) \vee$
$(controlAgt(s) = agt \wedge (a \neq startControl(agt) \vee a \neq endControl(agt)))$

$ExtChSelected(do(a, s)) \equiv$
$agt.(a = agtLeft(agt) \vee a = agtRight(agt)) \vee$
$(ExtChSelected(s) \wedge \neg(\exists agt, \overrightarrow{list}.a = requestExtCh(agt, \overrightarrow{list})))$

$ItrSelected(do(a, s)) \equiv$
$agt.(a = agtContinue(agt) \vee a = agtStop(agt))$
$(ItrSelected(s) \wedge \neg(\exists agt.a = requestItr(agt)))$

$PickSelected(do(a, s)) \equiv$
$\exists agt, x.a = agtPick(agt, x) \vee$
$(PickSelected(s) \wedge \neg(\exists agt.a = requestPick(agt)))$

$NumSelected(do(a, s)) \equiv$
$\exists agt, num.a = agtNumber(agt, num) \vee$
$(numSelected(s) \wedge \neg(\exists agt.a = requestNum(agt)))$

$Left(do(a, s)) \equiv$
$\exists agt.a = agtLeft(agt) \vee (Left(s) \wedge \neg(\exists agt.a = agtRight(agt)))$

$Right(do(a, s)) \equiv$
$existsagt.a = agtRight(agt) \vee (Right(s) \wedge \neg(\exists agt.a = agtLeft(agt)))$

$Continue(do(a, s)) \equiv$
$\exists agt.a = agtCotinue(agt) \vee (Continue(s) \wedge \neg(\exists agt.a = agtStop(agt)))$

$Stop(do(a, s)) \equiv$
$\exists agt.a = agtStop(agt) \vee (Stop(s) \wedge \neg(\exists agt.a = agtContinue(agt)))$

$pickMade(agt, do(a, s)) = x \equiv$
$a = agtPick(agt, x) \vee (pickMade(agt, s) = x \wedge a \neq agtPick(agt, x))$

$NumMade(agt, do(a, s)) = num \equiv$
$a = agtNumber(agt, num) \vee (numMade(agt, s) = num \wedge a \neq$
$agtNumber(agt, num))$

Procedure Definitions for External Nondeterministic Constructs

Following are proper Golog-family based procedure definitions for each of external nondeterministic constructs using the above defined sets of predicates, actions, fluents, and axioms.

1 - External Nondeterministic Choice $[agt\,\Gamma_1 \mid \Gamma_2]$:

proc $ExtNDChoice(agt, \Gamma_1, \Gamma_2)$
  $startControl(agt); requestExtCh(agt, [\Gamma_1, \Gamma_2]); ExtChSelected?;$
  $[Left?; endControl(agt); \Gamma_1 \mid Right?; endControl(agt); \Gamma_2]$
end

The *ExtNDChoice* procedure takes three parameters: the agent name and two programs (tasks). After giving the control to the agent, the system (or user program) sends a request to the agent for selecting a program between two given programs. The system waits until the agent generates exogenous action to tell the decision, either *agtLeft* or *agtRight*, which makes the fluent *ExtCh-Selected* true. The system then takes back the control and starts execution of the selected program. Note that the process of taking the control back from the agent is carried, after deciding left or right side in order to avoid any wrong value due to concurrent processes.

2 - External Nondeterministic Choice of Argument $[agt\, \pi x.\Gamma(x)]$:

proc $ExtNDChArg(agt, \Gamma)$
$startControl(agt); requestPick(agt); PickSelected?;$
$\pi x.\{pickMade(agt) = x?; endControl(agt); \Gamma(x)\}$
end

The *ExtNDChArg* procedure takes two parameters: the agent name and the program. After giving the control to the agent, the system (or user program) asks the agent for picking a binding for the variable. The fluent *pickMade* contains the picking binding and the system binds this with the program and then executes the program according to this binding.

3 - External Nondeterministic Iteration $[agt\, \Gamma]^*$:

proc $ExtNDItr(agt, \Gamma)$
$startControl(agt); requestItr(agt); ItrSelected?;$
while $(Continue \wedge \neg Stop)$
do $(endControl(agt); \Gamma; startControl(agt); requestItr(agt); ItrSelected?)$
$endControl(agt);$
end

The *ExtNDIrt* procedure uses *Continue* and *Stop* fluents in order to keep the loop of executing program. If the agent generates *agtContinue* action, then the while-condition becomes true and the do-part starts execution; otherwise, when the agent generates *agtStop* action then the while-condition becomes false and the system takes control back from the agent.

4 - External Nondeterministic Selected Priority $[agt\,\Gamma_1\langle\rangle\Gamma_2]$:

proc $ExtNDSelPrt(agt, \Gamma_1, \Gamma_2)$
$startControl(agt); requestExtCh(agt, [\Gamma_1, \Gamma_2]); ExtChSelected?;$
$[Left?; endControl(agt); (\Gamma_1\rangle\rangle\Gamma_2) \mid Right?; endControl(agt); (\Gamma_2\rangle\rangle\Gamma_1)]$
end

The *ExtNDSelPrt* procedure deals with priority concurrency. If the agent chooses left side then the left-sided process gets higher priority and vice versa.

5 - External Nondeterministic Selected Concurrent Iteration $[agt\,\Gamma]^{\|}$:

proc $ExtNDConItr(agt, \Gamma)$
$startControl(agt); requestNum(agt); NumSelected?;$
$[(NumMade(agt) = 0)?; endControl(agt) \mid (NumMade(agt) > 0)?;$
$\pi n.\{NumMade(agt) = n?; endControl(agt); [\Gamma]^{\|^n}\}]$
end

The *ExtNDSelPrt* procedure checks if the agent has selected zero number then it takes the control back from the agent and does nothing. Otherwise, if the given number is greater than zero, then it creates instances of the program for that given number of times and then executes these in the concurrent iteration form.

6 - External Nondeterministic First-Step Decision Concurrency $[agt\,\Gamma_1\langle|\rangle\Gamma_2]$:

proc $ExtNDFirstStepCon(agt, \Gamma_1, \Gamma_2)$
$startControl(agt); requestExtCh(agt, [\Gamma_1, \Gamma_2]); ExtChSelected?;$
$[Left?; endControl(agt); (\Gamma_1\langle\langle|\ \Gamma_2) \mid Right?; endControl(agt); (\Gamma_1\ |\rangle\rangle\Gamma_2)]$
end

The external first-step decision concurrency construct also uses two new "auxiliary" constructs to model the selected type of concurrency. The *Trans* and *Final* definition, at the interpreter level, of these two "auxiliary" constructs is provided in the Appendix B. In *ExtNDFirstStepCon* procedure, if the agent decides the left side then it calls the *left-side-first* auxiliary construct $(\Gamma_1\langle\langle|\ \Gamma_2)$; in which, the first executed action will be from left side and the remaining will be the normal concurrency between the remaining part of left side program and the right side program. The *right-side-first* auxiliary construct $(\Gamma_1\ |\rangle\rangle\Gamma_2)$ performs the first action from right side program and then the normal concurrency between both.

Appendix B

Prolog-based Code for IndiGolog Interpreter

This appendix provides the Prolog-based corresponding code of the Golog-family based program parts provided in the Appendix A. The targeted platform is IndiGolog interpreter implementation `P-INDIGOLOG` [104, 105]. This code implementation can be used directly in user programs created for `P-INDIGOLOG` platform provided the handling of actions at agent's side. In `P-INDIGOLOG` platform, external entities (applications/systems/devices) communicate with the system through *Environment Manager* module and each of these external entities is responsible to define internally how to handle the action execution. The *Environment Manager* sends action execution commands and also receives exogenous action generation signals from external entities and let the system know as it receives. Here, we are not going into the domain dependent low-level details related to the execution of requests by external entities, it receives from the system, for making nondeterministic choice decision and the relaying back such decision. We assume that the user program and the related external entity will provide these domain-dependent low level details.

Predicate for Agents

Each agent will be define by the agent predicate, but there will be only one definition of the system agent through the sysAgent predicate.

```
agent(agt).
sysAgent(sys).
```

Relational and Functional Fluents

The predicate rel_fluent stands for relational fluent, while predicate fun_fluent stands for functional fluent.

```
fun_fluent(sysAgt).
rel_fluent(free).
rel_fluent(extChSelected).
rel_fluent(itrSelected).
rel_fluent(pickSelected).
rel_fluent(numSelected).
rel_fluent(left).
rel_fluent(right).
rel_fluent(continue).
rel_fluent(stop).
fun_fluent(controlAgt).
rel_fluent(pickMade(Agt)) :- agent(Agt).
rel_fluent(numMade(Agt)) :- agent(Agt).
rel_fluent(agtEnv(Agt)) :- agent(Agt).
```

Primary Actions

To define each primary action.

```
prim_action(startControl(Agt)) :- agent(Agt).
prim_action(endControl(Agt)) :- agent(Agt).
prim_action(requestExtCh(Agt, List)) :- agent(Agt).
prim_action(requestItr(Agt)) :- agent(Agt).
prim_action(requestItr(Agt)) :- agent(Agt).
prim_action(requestPick(Agt)) :- agent(Agt).
prim_action(requestNum(Agt)) :- agent(Agt).
```

Exogenous Action

The choice actions are treated as exogenous actions.

```
exog_action(agtLeft(Agt)) :- agent(Agt).
exog_action(agtRight(Agt)) :- agent(Agt).
exog_action(agtContinue(Agt)) :- agent(Agt).
exog_action(agtStop(Agt)) :- agent(Agt).
exog_action(agtPick(Agt, X)) :- agent(Agt).
exog_action(agtNumber(Agt, Num)) :- agent(Agt).
```

Preconditin Axioms for Actions

Precondition axioms for each primary and exogenous action, where in predicate poss(Action, Cond), Cond provides precondition axioms.

```
poss(startControl(Agt), and(free, neg(agtEnv(Agt)))).
poss(endControl(Agt), and(neg(free), agtEnv(Agt))).
poss(requestExtCh(Agt, List), agtEnv(Agt)) :-  noEmpty(List).
poss(requestPick(Agt), agtEnv(Agt)).
poss(requestItr(Agt), agtEnv(Agt)).
poss(requestNum(Agt), agtEnv(Agt)).
poss(agtLeft(Agt), agtEnv(Agt)).
poss(agtRight(Agt), agtEnv(Agt)).
poss(agtContinue(Agt), agtEnv(Agt)).
poss(agtStop(Agt), agtEnv(Agt)).
poss(agtPick(Agt), agtEnv(Agt)).
poss(agtNumber(Agt, Num), and(agtEnv(Agt), Num >= 0)).
```

Initial Values for Fluent

The predicate initially(Fluent, InitValue) provides initial value of each fluent.

```
initially(sysAgt, Sys) :- sysAgent(Sys).
initially(free, true).
initially(controlAgt, Sys) :- sysAgent(Sys).
initially(agtEnv(Agt), false) :- agent(Agt).
initially(extChSelected, false).
initially(pickSelected, false).
initially(numSelected, false).
initially(left, false).
initially(right, false).
initially(continue, false).
initially(stop, true).
initially(pickMade(Agt), []) :- agent(Agt).
initially(numMade(Agt), 0) :- agent(Agt).
```

Successor State Values of Fluents

P-INDIGOLOG *provides three predicates for defining successor state values for fluents.*
causes_true$(Action, Fluent, Cond)$ *provides information when a relational fluent becomes true.*
causes_false$(Action, Fluent, Cond)$ *provides information when a relational fluent becomes false.*
causes_value$(Action, Fluent, Value, Cond)$ *provides information when a functional fluent gets a new value.*

```
causes_true(endControl(Agt), free, true).
```

```
causes_false(startControl(Agt), free, true).
causes_true(startControl(Agt), agtEnv(Agt), true).
causes_false(endControl(Agt), agtEnv(Agt), true).
causes_true(agtLeft(Agt), extChSelected, true).
causes_true(agtRight(Agt), extChSelected, true).
causes_false(requestExhCh(Agt, List), extChSelected, true).
causes_true(agtContinue(Agt), itrSelcted, true).
causes_true(agtStop(Agt), itrSelcted, true).
causes_false(requestItr(Agt), itrSelcted, true).
causes_true(agtPick(Agt, X), pickSelcted, true).
causes_false(requestPick(Agt), pickSelcted, true).
causes_true(agtNumber(Agt, Num), numSelcted, true).
causes_false(requestNum(Agt), numSelcted, true).
causes_true(agtLeft(Agt), left, true).
causes_false(agtRight(Agt), left, true).
causes_true(agtRight(Agt), right, true).
causes_false(agtLeft(Agt), right, true).
causes_true(agtContinue(Agt), continue, true).
causes_false(agtStop(Agt), continue, true).
causes_true(agtStop(Agt), stop, true).
causes_false(agtContinue(Agt), stop, true).
causes_val(agtPick(Agt, X), pickMade(Agt), X, agtEnv(Agt)).
causes_val(agtNumber(Agt, Num), numMade(Agt), Num,
                                and(agtEnv(Agt), Num >= 0)).
causes_val(startControl(Agt), controlAgt, Agt, true).
causes_val(endControl(Agt), controlAgt, sysAgt, true).
```

Procedure Definition for External Nondeterministic Constructs

1. External Nondeterministic Choice $[agt\,\Gamma_1 \mid \Gamma_2]$:

```
proc(ExtNDChoice(Agt, Prog1, Prog2),
 [startControl(Agt), requestExtCh(Agt, [Prog1, Prog2]),
 while(not(extChSelected), wait),
 ndet([?(left), endControl(Agt), Prog1],
 [ ?(right), endControl(Agt), Prog2])]).
```

Here, P-INDIGOLOG predicate $?(\phi)$ stands for *waiting* action, while *ndet* stands for internal nondeterministic choice. As P-INDIGOLOG platform based on Prolog, so it starts execution from the left side and it first checks whether the first program is able to execute. If so, then it executes the first one, otherwise; it backtracks and tries the second program.

2. External nondeterministic Choice of Argument $[agt\,\pi x.\Gamma(x)]$:

```
proc(ExtNDChArg(Agt, Prog),
 [startControl(Agt), requestPick(Agt),
 while(not(pickChSelected), wait),
 endControl(Agt), Pi(p, pickMade, Prog)]).
```

The `P-INDIGOLOG` predicate $pi(V, D, P)$ is for internal nondeterministic choice of argument where V stands for variable for binding with program while D is domain from which the system selects the variable, and P is the program to which the system do binding.

3. External nondeterministic Iteration $[agt\,\Gamma]^*$:

```
proc(ExtNDIt(Agt, Prog),
 [startControl(Agt), requestItr(Agt),
 while(not(ItrChSelected), wait),
 While(and(continue, not(stop)),
 [endControl(Agt), Prog, startControl(Agt), requestItr(Agt),
 while(not(ItrChSelected), wait)]), endControl(Agt) ]).
```

4. External nondeterministic Selected Priority $[agt\,\Gamma_1\langle\rangle\Gamma_2]$:

```
proc(ExtNDSelPrt(Agt, Prog1, Prog2),
 [startControl(Agt), requestExtCh(Agt, [Prog1, Prog2]),
 while(not(extChSelected), wait),
 ndet([?(left), endControl(Agt), pconc(Prog1, Prog2)],
 [?(right), endControl(Agt), pconc(Prog2, Prog1)])]).
```

The predicate `pconc(P1, P2)` is for priority concurrency where `P1` gets higher priority than `P2`.

5. External nondeterministic Selected Concurrent Iteration $[agt\,\Gamma]^{\|}$:

```
proc(ExtNDConItr(Agt, Prog),
 [startControl(Agt), requestNum(Agt]),
 while(not(numSelected), wait),
 ndet([?(numMade(Agt)=0), endControl(Agt)],[ ?(numMade(Agt)>0),
 while((numMade(Agt)>0),
   [numMade(Agt) is numMade(Agt) - 1, append(Prog, List, List)]),
   endControl(Agt),rrobin(List)])]).
```

Here, when the agent returns number greater than zero, the program is added in the list that given number of times through Prolog function `append(Prog, List, List)`. The `P-INDIGOLOG` predicate `rrobin/1` takes a list of programs and performs concurrency in round robin fashion.

6. External nondeterministic First-Step Decision Concurrency $[agt\,\Gamma_1\langle|\rangle\Gamma_2]$:

```
proc(ExtNDFirstStepCon(Agt, Prog1, Prog2),
 [startControl(Agt), requestExtCh(Agt, [Prog1, Prog2]),
 while(not(extChSelected), wait),
 ndet([?(left), endControl(Agt), lsfconc(Prog1, Prog2)],
 [ ?(right), endControl(Agt), srfconc(Prog1, Prog2)])]).
```

The predicates `lsfconc` and `rsfconc` (define below) provide the *left-side-first* concurrency and *right-side-first* concurrency at interpreter level.

Trans and *Final* for lsfcon ($\Gamma_1\langle\langle|\ \Gamma_2$) and rsfcon ($\Gamma_1\ |\rangle\rangle\Gamma_2$)

The external *first-step decision concurrency* construct definition uses two new "auxiliary" constructs to model the selected type of concurrency. Following is their *Trans* and *Final* definition at `P-INDIGOLOG` platform level.

```
trans(lsfconc(E1, E2), H, E, H1) :-
      trans(E1, H, E3, H1), E = conc(E3,E2).
trans(rsfconc(E1, E2), H, E, H1) :-
      trans(E2, H, E3, H1), E = conc(E1,E3).
```

Here, `lsfconc` stands for *left-side-first* concurrency ($\Gamma_1\langle\langle|\ \Gamma_2$) while `rsfconc` stands for *right-side-first* concurrency ($\Gamma_1\ |\rangle\rangle\Gamma_2$). The *Trans* of `lsfconc` says that the program makes transaction of `E1` and the remaining part `E` is then a normal concurrency between `E3` and `E2`. The *Trans* of `rsfconc` makes transaction of `E2` and the remaining part `E` is then a normal concurrency between `E1` and `E3`. Following is *Final* definitions of both, where they are in final state if both `E1` and `E2` are in the final state.

```
final(lsfconc(E1, E2), H) :- final(E1, H), final(E2, H).
final(rsfconc(E1, E2), H) :- final(E2, H), final(E1, H).
```

Appendix C

Labeling Framework Implementation

In this appendix, we provide a way to implement external nondeterministic constructs through a labeling framework. In the implementation provided in Appendix A and Appendix B, when the system (or user program) requests the agent for making nondeterministic decision, the requesting action also contains program parts as parameters to let the agent know about choices for making the final decision. In practice, normally when it is asked to an external agent for making decision, it is not supposed to send the full program parts; rather than, only labels or names of choices are sent, the agent then makes decision and tells back to the system about the decision through the chosen label or choice name. Another important consideration is that it is possible to have more than two choices and sending program-parts as parameters does not look a better solution.

To tackle this problem, here briefly, we define a framework that provides a set of predicates using situation calculus to give labels to program parts, which can also be sent to the agent rather than program parts. The agent makes decision and sends this back by selecting the decided label. The following are predicates for defining labels to program parts.

$label(l)$: l is a label
$chProg(\Gamma_c)$: Γ_c is a program-part that is represented by a label.
$attaching(l, \Gamma_c)$: label l represents a program-part Γ_c.

The following functions are used to find different properties of above label predicates.

$attached(l, \Gamma_c)$: given a label l and program-part Γ_c, it tells whether the label is attached to a program-part or not.

$getChProg(l) = \Gamma_c$: given a label l, this function returns the program-part Γ_c attached to the label.
$getLabel(\Gamma_c) = l$: given a program-part Γ_c, this function returns back the label l attached to it.

We assume that a label can be attached to only one program-part and vice versa. Formally:

$$attached(l, \Gamma_c) \wedge attached(l', \Gamma_c) \Rightarrow l = l'$$
$$attached(l, \Gamma_c) \wedge attached(l, \Gamma'_c) \Rightarrow \Gamma_c = \Gamma'_c$$

We assume that previous defined fluents and their successor state axioms, actions, and their precondition axioms are valid in this framework too. Below, we provide only the additional set of fluents and their successor state axioms, actions and their precondition axioms, and procedure definition for the external nondeterministic choice construct. We are not going into detail of every external nondeterministic construct; however, this approach can be used for implementing other external nondeterministic constructs.

Actions

$requestExtChoice(agt, \overrightarrow{list})$: requests to agent agt to make a choice from the $\overrightarrow{list}$ that contains labels of all participant program parts.

Exogenous Actions

$selectionMade(agt, choice)$: agent agt chooses from the list of labels, where $choice$ represents a label attached to some program-part.

Fluents

$chList(s) = \overrightarrow{list}$: contains the current list of labels sent to agent for making a choice
$choiceMade(s) = choice$: contains the $choice$ made by agent from the list of labels
$ExtSelected(s)$: becomes true when the agent makes the decision

Initial values of fluents

$chList(s) = nil$
$choiceMade(s) = nil$
$\neg ExtSelected(s)$

Precondition axioms for Actions

$Poss(requestExtChoice(agt, \overrightarrow{list}), s) \equiv AgtEnv(agt, s) \wedge \neg(\overrightarrow{list} = \mathfrak{d})$
$Poss(selectionMade(agt, choice), s) \equiv AgtEnv(agt, s) \wedge choice \in chList(s)$

Effect axioms for relation fluents

$ExtSelected(do(\exists agt, choice.selectionMade(agt, choice), s))$
$\neg ExtSelect(do(\exists agt, \overrightarrow{list}.requestExtChoice(agt, \overrightarrow{list}), s))$

Successor State Axioms for Fluents

$ExtSelected(do(a, s)) \equiv$
$\exists agt, choice.a = selectionMade(agt, choice) \vee$
$(ExtSelected(s) \wedge \neg(\exists agt, \overrightarrow{list}.requestExtChoice(agt, \overrightarrow{list})))$
$chList(do(a, s)) = (list) \equiv$
$\exists agt.a = requestExtChoice(agt, \overrightarrow{list}) \vee$
$(chList(s) = \overrightarrow{list} \wedge \neg(\exists agt.a = requestExtChoice(agt, \overrightarrow{list})))$
$choiceMade(do(a, s)) = choice \equiv$
$\exists agt.a = selectionMade(agt, choice) \vee$
$(choiceMade(s) = choice \wedge \exists(\exists agt.a = selectionMade(agt, choice)))$

Procedure Definition

External Nondeterministic Choice $[agt\, \Gamma_1 \mid \Gamma_2]$:

proc $ExtNDChoiceLabels(agt, [l_1, l_2, ..., l_n])$
$startControl(agt); requestExtChoice(agt, [l_1, l_1, .., l_n]); ExtSelected?;$
$\pi\Gamma_c[(getProg(choiceMade) = \Gamma c)?; endControl(agt); \Gamma_c]$
end

The *ExtNDChoiceLabels* procedure takes two parameters: the agent name and a list of labels, where each label is attached to the corresponding program. After giving the control to the agent, the system (or user program) sends a request to the agent with a list of labels for making choice from that list. The system waits until the agent generates exogenous action $selectionMade(choice)$ in which parameter *choice* contains label chosen by agent. The system then takes the corresponding program through function *getProg* and executes the selected program. It is noteworthy that the control is taken back from the agent after taking the corresponding program from function *getProg*. This is important in order to avoid any collision in the case of concurrency. As the fluent *choiceMade* keeps the latest choice selected by an external agent; so, if we take control before taking the selected attached program then it may be possible that the process gets back control after some other concurrent process executes some external nondeterministic construct. In which case, the fluent *choiceMade* will have a new value. However, in the above-defined procedure, no other concurrent process can use nondeterministic construct until the process, who uses this procedure, takes the selected attached program.

Prolog-based Code

The following are labeling-functions definitions in Prolog-based syntax.

```
attached(L,P) :- label(L), chProg(P), attaching(L,P).
getLabel(P,L) :- chProg(P), attaching(L2,P), label(L2), L=L2.
getChProg(L,P) :- label(L), attaching(L,P2), chProg(P2), P=P2.
```

Here is Prolog-based code for the procedure *ExtNDChoiceLabels* for the `P-INDIGOLOG` platform.

```
proc(extNDChoiceLabels(Agt, List),
[startControl(Agt), requestExtChoice(Agt, List),
while(not(extSelected), wait),
pi(prog, [?(getChProg(choiceMade,prog)), endControl(Agt), prog])].
```

Bibliography

[1] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta. Agile software development methods - review and analysis. Technical Report 478, VTT PUBLICATIONS, 2002.

[2] Agile-Alliance. Manifesto for agile software development. Technical report, 2001. `http://www.agilealliance.org`.

[3] M. Alshamari and P. Mayhew. Task design: Its impact on usability testing. *Internet and Web Applications and Services, International Conference on*, 0:583–589, 2008.

[4] J. Anderson, F. Fleek, K. Garrity, and F. Drake. Integrating usability techniques into software development. *IEEE Software*, 18(1), 2001.

[5] J. Annett and K. Duncan. Task analysis and training design. *Occupational Psychology*, 41:211–221, 1967.

[6] S. balbo. Automatic evaluation of user interface usability: Dream or reality. In *In S. Balbo, Ed., Proceedings of the Queensland Computer- Human Interaction Symposium*, Bond University, Queensland, Australia, August 1995.

[7] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[8] H. Beyer and K. Holtzblatt. *Contextual design: defining customer-centered systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[9] S. Blomkvist. Towards a model for bridging agile development and user-centered design. In A. Seffah, J. Gulliksen, and M. C. Desmarais, editors, *Human-Centered Software Engineering Integrating Usability in the Software Development Lifecycle*, volume 8 of *Human-Computer Interaction Series*, pages 219–244. Springer Netherlands, 2005.

[10] G. Buchholz, J. Engel, C. Märtin, and S. Propp. Model-based usability evaluation - evaluation of tool support. In *HCI (1)*, pages 1043–1052, 2007.

[11] S. K. Card, A. Newell, and T. P. Moran. *The Psychology of Human-Computer Interaction*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1983.

[12] S. Chamberlain, H. Sharp, and N. Maiden. Towards a Framework for Integrating Agile Development and User-Centred Design. pages 143–153. 2006.

[13] A. Cockburn. *Crystal clear a human-powered methodology for small teams*. Addison-Wesley Professional, first edition, 2004.

[14] A. Cockburn. *Agile Software Development: The Cooperative Game (2nd Edition) (Agile Software Development Series)*. Addison-Wesley Professional, 2006.

[15] L. L. Constantine and L. A. D. Lockwood. *Software for use: a practical guide to the models and methods of usage-centered design*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[16] L. L. Constantine and L. A. D. Lockwood. Usage-centered software engineering: an agile approach to integrating users, user interfaces, and usability into software engineering practice. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 746–747, Washington, DC, USA, 2003. IEEE Computer Society.

[17] B. Crandall, G. Klein, and R. R. Hoffman. *Working Minds: A Practitioner's Guide to Cognitive Task Analysis (Bradford Books)*. The MIT Press, 1 edition, July 2006.

[18] G. de Giacomo, Y. Lespérance, and H. J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artif. Intell.*, 121:109–169, August 2000.

[19] G. de Giacomo, Y. Lespérance, H. J. Levesque, and S. Sardina. IndiGolog: A high-level programming language for embedded reasoning agents. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, chapter 2, pages 31–72. Springer, New York, USA, 2009. ISBN: 978-0-387-89298-6.

[20] G. de Giacomo, Y. Lespérance, and A. R. Pearce. Situation calculus based programs for representing and reasoning about game structures. In *KR*, 2010.

[21] M. de Leoni, G. D. Giacomo, Y. Lespérance, and M. Mecella. On-line adaptation of sequential mobile processes running concurrently. In *SAC*, pages 1345–1352, 2009.

[22] M. de Leoni, M. Mecella, and G. D. Giacomo. Highly dynamic adaptation in process management systems through execution monitoring. In *BPM*, pages 182–197, 2007.

[23] M. Detweiler. Managing ucd within agile projects. *Interactions*, 14(3):40–42, 2007.

[24] D. Diaper. Task analysis for knowledge description (takd): the method and an example. In D. Diaper, editor, *Task Analysis for Human-Computer Interaction*, chapter 4. Ellis Horwood.

[25] A. Dix, J. E. Finlay, G. D. Abowd, and R. Beale. *Human-Computer Interaction (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003.

[26] Y. Dubinsky, T. Catarci, S. R. Humayoun, and S. Kimani. Managing user-centred design in agile projects. In *In the workshop on "Optimizing Agile User-Centered Desig"during the 26th ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'2008)*, Florence, Italy, 2008.

[27] Y. Dubinsky and O. Hazzan. *A framework for teaching software development methods*, volume 15. 2005.

[28] Y. Dubinsky and O. Hazzan. Using a role scheme to derive software project metrics. *J. Syst. Archit.*, 52:693–699, November 2006.

[29] Y. Dubinsky, S. R. Humayoun, and T. Catarci. Eclipse plug-in to manage user centered design. In *I-USED*, 2008.

[30] Y. Dubinsky, S. R. Humayoun, T. Catarci, and S. kimani. Integrating user evaluation into software development environments. In *2nd DELOS Conference on Digital Libraries*, Pisa, Italy, 2007.

[31] T. Dybå and T. Dingsøyr. Empirical studies of agile software development: A systematic review. *Inf. Softw. Technol.*, 50:833–859, August 2008.

[32] Eclipse-Platform. http://www.eclipse.org/platform/. *The Eclipse Foundation.*

[33] K. Erol, J. Hendler, and D. S. Nau. Semantics for hierarchical task-network planning. Technical report, College Park, MD, USA, 1994.

[34] K. Erol, J. Hendler, and D. S. Nau. Htn planning: Complexity and expressivity. In *In Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94*, pages 1123–1128. AAAI Press, 94.

[35] X. Ferr. Integration of usability techniques into the software development process, 2003.

[36] X. Ferré, N. Juristo, H. Windl, and L. Constantine. Usability basics for software developers. *IEEE Softw.*, 18:22–29, January 2001.

[37] R. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In *IJCAI*, pages 608–620, 1971.

[38] D. Fox, J. Sillito, and F. Maurer. Agile methods and user-centered design: How these two methodologies are being successfully integrated in industry. In *Proceedings of the Agile 2008*, pages 63–72, Washington, DC, USA, 2008. IEEE Computer Society.

[39] G. D. Giacomo and H. Levesque. An incremental interpreter for high-level programs with sensing. In *Logical Foundations for Cognitive Agents*, pages 86–102. Springer, 1998.

[40] B. Göransson, J. Gulliksen, and I. Boivie. The usability design process - integrating user-centered systems design in the software development process. *Software Process: Improvement and Practice*, 8(2):111–131, 2003.

[41] J. D. Gould. Human-computer interaction. chapter How to design usable systems, pages 93–121. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995.

[42] J. D. Gould, S. J. Boies, and J. Ukelson. Human-computer interaction. chapter How to design usable systems. Amsterdam: Elsevier Science B.V, 1997.

[43] J. D. Gould and C. Lewis. Designing for usability: key principles and what designers think. *Commun. ACM*, 28:300–311, March 1985.

[44] J. Gulliksen and B. Göransson. Usability design: Integrating user centered system design in the software development process. In *INTERACT*, 2003.

[45] J. Gulliksen, B. Göransson, I. Boivie, S. Blomkvist, J. Persson, and Å. Cajander. Key principles for user-centred systems design. *Behaviour & IT*, 22(6):397–409, 2003.

[46] H. R. Hartson and P. D. Gray. Temporal aspects of tasks in the user action notation. *Hum.-Comput. Interact.*, 7:1–45, March 1992.

[47] H. R. Hartson, A. C. Siochi, and D. Hix. The uan: a user-oriented representation for direct manipulation interface designs. *ACM Trans. Inf. Syst.*, 8:181–203, July 1990.

[48] A. Hauser. Ucd collaboration with product management and development. *interactions*, 14:34–35, May 2007.

[49] O. Hazzan and Y. Dubinsky. *Agile Software Engineering*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[50] H. Hellman and K. Rnkk. Is user experience support effectively in existing software development processes?. In *5th COST294-MAUSE Open Workshop Meaningful Measures: Valid Userfil User Experience Measurment VUUM08*, June 2008.

[51] M. Hennessy. *The Semantics of Programming Languages*. John Wiley & Sons, 1990.

[52] J. A. Highsmith, III. *Adaptive software development: a collaborative approach to managing complex systems*. Dorset House Publishing Co., Inc., New York, NY, USA, 2000.

[53] K. Holtzblatt and S. Jones. Contextual Inquiry: A Participatory Technique for System Design. In D. Schuler and A. Namioka, editors, *Participatory Design. Principles and Practices.*, pages 177–210. Lawrence Erlbaum Associates., Hillsdale, New Jersey, 1993.

[54] S. R. Humayoun, T. Catarci, M. de Leoni, A. Marrella, M. Mecella, M. Bortenschlager, and R. Steinmann. The workpad user interface and methodology: Developing smart and effective mobile applications for emergency operators. In *HCI (7)*, pages 343–352, 2009.

[55] S. R. Humayoun, T. Catarci, M. de Leoni, A. Marrella, M. Mecella, M. Bortenschlager, and R. Steinmann. Designing mobile systems in highly dynamic scenarios: The workpad methodology. *Knowledge, Technology and Policy*, 22:25–43, 2009. 10.1007/s12130-009-9070-3.

[56] S. R. Humayoun, T. Catarci, and Y. Dubinsky. A dynamic framework for multi-view task modeling. In *Proceedings of the 9th ACM SIGCHI*

Italian Chapter International Conference on Computer-Human Interaction: Facing Complexity, CHItaly, pages 185–190, New York, NY, USA, 2011. ACM.

[57] S. R. Humayoun, T. Catarci, Y. Dubinsky, E. Nazarov, and A. Israel. Using a high level formal language for task model-based usability evaluation. In *M. De Marco, D. Teeni, V. Albano, S. Za (Ed.): "Information Systems: Crossroads for Organization,Management, Accounting and Engineering"*. Physica-Verlag Heidelberg - Springer, 2011.

[58] S. R. Humayoun, Y. Dubinsky, and T. Catarci. Ueman: A tool to manage user evaluation in development environments. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 551–554, Washington, DC, USA, 2009. IEEE Computer Society.

[59] S. R. Humayoun, Y. Dubinsky, and T. Catarci. A three-fold integration framework to incorporate user centred design into agile software development. In *Lecture Notes in Computer Science, LNCS Volume 6776, M. Kurosu (Ed.): Human Centered Design, HCII 2011*, pages 55–64, 2011.

[60] S. R. Humayoun, Y. Dubinsky, E. Nazarov, A. Israel, and T. Catarci. Tamulator: A tool to manage task model-based usability evaluation in development environments. In *Proceedins of IADIS Conference on Interfaces and Human Computer Interaction 2011*, IHCI 2011, Rome, Italy, July 2011.

[61] Z. Hussain, H. Milchrahm, S. Shahzad, W. Slany, M. Tscheligi, and P. Wolkerstorfer. Integration of extreme programming and user-centered design: Lessons learned. In W. Aalst, J. Mylopoulos, N. M. Sadeh, M. J. Shaw, C. Szyperski, P. Abrahamsson, M. Marchesi, and F. Maurer, editors, *Agile Processes in Software Engineering and Extreme Programming*, volume 31 of *Lecture Notes in Business Information Processing*, pages 174–179. Springer Berlin Heidelberg, 2009.

[62] IEEE-P1471. Recommended practice for architectural description. Technical standard, Institute of Electrical and Electronics Engineers. `http://www.iso-architecture.org/ieee-1471/introducing-p1471.pdf`.

[63] M. Y. Ivory and M. A. Hearst. The state of the art in automating usability evaluation of user interfaces. *ACM Comput. Surv.*, 33:470–516, December 2001.

[64] M. A. Jackson. *Principles of Program Design*. Academic Press, Inc., Orlando, FL, USA, 1975.

[65] N. Juristo, A. Moreno, and M.-I. Sanchez-Segura. Guidelines for eliciting usability functionalities. *IEEE Trans. Softw. Eng.*, 33:744–758, November 2007.

[66] N. J. Juzgado, H. Windl, and L. L. Constantine. Guest editors' introduction: Introducing usability. *IEEE Software*, 18(1):20–21, 2001.

[67] T. Kato. What "question-asking protocols" can say about the user interface. *International Journal of Man-Machine Studies*, 25(6):659 – 673, 1986.

[68] G. Kiczales. Aspect-oriented programming. In *ICSE*, page 730, 2005.

[69] T. K. Landauer. *The Trouble with Computers: Usefulness, Usability, and Productivity*. The MIT Press, 1996.

[70] E. L.-C. Law. A multi-perspective approach to tracking the effectiveness of user tests: A case study. In *Proceedings of the Workshop Improving the Interplay of Usability Evaluation and User Interface Design of NordiCHI 2004*, Oct. 2004.

[71] A. Lecerof and F. Paternò. Automatic support for usability evaluation. *IEEE Trans. Software Eng.*, 24(10):863–888, 1998.

[72] H. Levesque and M. Pagnucco. Legolog: Inexpensive experiments in cognitive robotics. In *Proceedings of the International Cognitive Robotics Workshop (COGROBO)*, pages 104–109, Berlin, Germany, 2000.

[73] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. Golog: A logic programming language for dynamic domains. *J. Log. Program.*, 31(1-3):59–83, 1997.

[74] C. Lewis, P. G. Polson, C. Wharton, and J. Rieman. Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Empowering people*, CHI '90, pages 235–242, New York, NY, USA, 1990. ACM.

[75] S. Lohmann and A. Rashid. Fostering remote user participation and integration of user feedback into software development. In *I-USED*, 2008.

[76] J.-Y. Mao, K. Vredenburg, P. W. Smith, and T. Carey. The state of user-centered design practice. *Commun. ACM*, 48:105–109, March 2005.

[77] D. J. Mayhew. *The usability engineering lifecycle: a practitioner's handbook for user interface design.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[78] R. Molich, A. D. Thomsen, B. Karyukina, L. Schmidt, M. Ede, W. van Oel, and M. Arcuri. Comparative evaluation of usability tests. In *CHI '99 extended abstracts on Human factors in computing systems*, CHI '99, pages 83–84, New York, NY, USA, 1999. ACM.

[79] A. Monk, P. Wright, J. Haber, and L. Davenport. *Improving Your Human-Computer Interface: A Practical Approach.* Prentice Hall International, Hemel Hempstead, 1993.

[80] G. Mori, F. Paternò, and C. Santoro. Ctte: Support for developing and analyzing task models for interactive system design. *IEEE Trans. Software Eng.*, 28(8):797–813, 2002.

[81] D. N. Morley and K. L. Myers. The spark agent framework. In *AAMAS*, pages 714–721, 2004.

[82] J. Nielsen. The usability engineering life cycle. *Computer*, 25:12–22, March 1992.

[83] J. Nielsen. *Usability Engineering.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[84] J. Nielsen and R. L. Mack. *Usability inspection methods.* Wiley, 1 edition, April 1994.

[85] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction.* John Wiley & Sons, 1992.

[86] D. Norman. Why doing user observations first is wrong. *interactions*, 13:50–ff, July 2006.

[87] D. A. Norman. *The Psychology Of Everyday Things.* Basic Books, June 1988.

[88] D. A. Norman and S. W. Draper. *User Centered System Design; New Perspectives on Human-Computer Interaction.* L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1986.

[89] S. R. Palmer and M. Felsing. *A Practical Guide to Feature-Driven Development.* Pearson Education, 1st edition, 2001.

[90] F. Paterno. *Model-Based Design and Evaluation of Interactive Applications.* Springer-Verlag, London, UK, 1st edition, 1999.

[91] F. Paternò and G. Ballardin. Remusine: a bridge between empirical and model-based evaluation when evaluators and users are distant. *Interacting with Computers*, 13(2):229–251, 2000.

[92] F. Paternò, C. Mancini, and S. Meniconi. Concurtasktrees: A diagrammatic notation for specifying task models. In *INTERACT*, pages 362–369, 1997.

[93] F. Paternò, A. Russino, and C. Santoro. Remote evaluation of mobile applications. In *TAMODIA*, pages 155–169, 2007.

[94] J. Patton. Hitting the target: adding interaction design to agile software development. In *OOPSLA 2002 Practitioners Reports*, OOPSLA '02, pages 1–ff, New York, NY, USA, 2002. ACM.

[95] M. E. Pollack. The uses of plans. *Artif. Intell.*, 57(1):43–68, 1992.

[96] P. G. Polson, C. Lewis, J. Rieman, and C. Wharton. Cognitive walkthroughs: a method for theory-based evaluation of user interfaces. *Int. J. Man-Mach. Stud.*, 36:741–773, May 1992.

[97] S. Propp, G. Buchholz, and P. Forbrig. Task model-based usability evaluation for smart environments. In *TAMODIA/HCSE*, pages 29–40, 2008.

[98] K. Radle and S. Young. Partnering usability with development: How three organizations succeeded. *IEEE Softw.*, 18:38–45, January 2001.

[99] T. Rafla, P. N. Robillard, and M. Desmarais. A method to elicit architecturally sensitive usability requirements: its integration into a software development process. *Software Quality Control*, 15:117–133, June 2007.

[100] A. S. Rao. Agentspeak(l): Bdi agents speak out in a logical computable language. In *MAAMAW*, pages 42–55, 1996.

[101] D. Reichart, P. Forbrig, and A. Dittmar. Task models as basis for requirements engineering and software execution. In *TAMODIA*, pages 51–58, 2004.

[102] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, September 2001.

[103] R. Restak. *The New Brain: How the Modern Age Is Rewiring Your Mind*. Rodale Books, 2003.

[104] S. Sardiña. Indigolog: An integrated agent arquitecture. Programmer and user manual, University of Toronto, 2004. `http://sourceforge.net/projects/indigolog/`.

[105] S. Sardiña. P-indigolog: An integrated agent arquitecture. Programmer and user manual, University of Toronto, 2005. `http://sourceforge.net/projects/indigolog/`.

[106] S. Sardina, G. De Giacomo, Y. Lespérance, and H. J. Levesque. On the semantics of deliberation in indigolog: from theory to implementation. *Annals of Mathematics and Artificial Intelligence*, 41:259–299, August 2004.

[107] S. Sardiña and Y. Lespérance. Golog speaks the bdi language. In *PROMAS*, pages 82–99, 2009.

[108] K. Schwaber. *Agile Project Management with Scrum*. Prentice Hall, 2004.

[109] K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[110] H. Sharp, Y. Rogers, and J. Preece. *Interaction Design: Beyond Human-Computer Interaction*. 2 edition, March 2007.

[111] D. Sinnig, M. Wurdel, P. Forbrig, P. Chalin, and F. Khendek. Practical extensions for task models. In *TAMODIA*, pages 42–55, 2007.

[112] J. Stapleton. *DSDM: Business Focused Development, 2/E: DSDM Consortium*. Addison-Wesley Professional, 2003.

[113] D. Sy. Adapting Usability Investigations for Agile User-Centered Design - International Journal of Usability Studies. *Journal of Usability Studies*, 2(3), May 2007.

[114] D. Talby, O. Hazzan, Y. Dubinsky, and A. Keren. Agile software testing in a large-scale project. *IEEE Softw.*, 23:30–37, July 2006.

[115] ISO. ISO 9241-11: Ergonomic requirements for office work with visual display terminals (vdts). The international organization for standardization, 1998.

[116] ISO. ISO/DIS 13407: Human centered design for interactive systems. The international organization for standardization, 1999.

[117] ISO. ISO TR 16982: Ergonomics of human-system interaction - human-cantered lifecycle process descriptions. The international organization for standardization, 2002.

[118] ISO (1998). Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on Temporal Ordering of Observation Behavior. ISO/IS 8807, ISO Central Secretariat, 1988.

[119] The-Eclipse-Plugin-Development-Environment. http://www.eclipse.org/pde/. *The Eclipse Foundation*.

[120] T. Uldall-Espersen and E. Frkjr. Usability and software development: Roles of the stakeholders. In J. Jacko, editor, *Human-Computer Interaction. Interaction Design and Usability*, volume 4550 of *Lecture Notes in Computer Science*, pages 642–651. Springer Berlin / Heidelberg, 2007.

[121] J. Ungar and J. White. Agile user centered design: enter the design studio - a case study. In *CHI '08 extended abstracts on Human factors in computing systems*, CHI '08, pages 2167–2178, New York, NY, USA, 2008. ACM.

[122] G. van der Veer and M. van Welie. Task based groupware design: putting theory into practice. In *Proceedings of the 3rd conference on Designing interactive systems: processes, practices, methods, and techniques*, DIS '00, pages 326–337, New York, NY, USA, 2000. ACM.

[123] K. Vredenburg, J.-Y. Mao, P. W. Smith, and T. Carey. A survey of user-centered design practice. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves*, CHI '02, pages 471–478, New York, NY, USA, 2002. ACM.

[124] C. Wharton, J. Rieman, C. Lewis, and P. Polson. *The cognitive walkthrough method: a practitioner's guide*, pages 105–140. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[125] J. Whiteside, J. Bennett, and K. Holtzblatt. Usability engineering: Our experience and evolution. In *Handbook of Human-Computer Interaction*, pages 791–817. M. Helander, ed., Elsevier Science Publishers, Amsterdam, 1988.

[126] J. Wielemaker. An overview of the swi-prolog programming environment. In *WLPE*, pages 1–16, 2003.

[127] E. Woodward, S. Surdek, and M. Ganis. *A Practical Guide to Distributed Scrum*. IBM Press, 1st edition, 2010.

[128] M. Wurdel, D. Sinnig, and P. Forbrig. CTML: Domain and task modeling for collaborative environments. *j-jucs*, 14(19):3188–3201, 2008. `http://www.jucs.org/jucs_14_19/ctml_domain_and_task`.