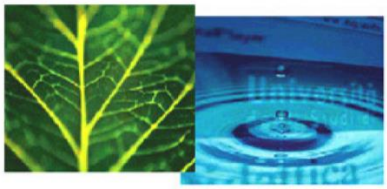
PhD Dissertation



International Doctorate School in Information and Communication Technologies

DISI – University of Trento

TOWARDS AN ONTOLOGY OF SOFTWARE

Xiaowei Wang

Advisor

Prof. John Mylopoulos

Universit à degli Studi di Trento

Co-Advisor

Dr. Nicola Guarino

Institute of Cognitive Sciences and Technologies,

Consiglio Nazionale delle Ricerche

April 2016

Abstract

Software is permeating every aspect of our personal and social life. And yet, the cluster of concepts around the notion of software, such as the notions of a software product, software requirements, software specifications, are still poorly understood with no consensus on the horizon. For many, software is just code, something intangible best defined in contrast with hardware, but it is not particularly illuminating. This erroneous notion, software is just code, presents both in the ontology of software literature and in the software maintenance tools. This notion is obviously wrong because it doesn't account for the fact that whenever someone fixes a bug, the code of a software system changes, but nobody believes that this is a different software system.

Several researchers have attempted to understand the core nature of software and programs in terms of concepts such as code, copy, medium and execution. More recently, a proposal was made by Irmak to consider software as an abstract artifact, distinct from code, just because code may change while the software remains the same. We share many of his intuitions, as well as the methodology he adopts to motivate his conclusions, based on an analysis of the condition under which software maintains its identity despite change. However, he leaves the question of 'what is the identity of software' open, and we answer this question here.

Trying to answer the question left open by Irmak, the main objective of this dissertation is to lay the foundations for an ontology of software, grounded on the foundational ontology DOLCE. This new ontology of software is intended to facilitate the communication within the community by reducing terminological ambiguities, and by resolving inconsistencies. If we had a better footing on answering the question 'What is software?', we'd be in a position to build better tools for maintaining and managing a software system throughout its lifetime. The research contents of the thesis consist of three results.

Firstly, we dive into the ontological nature of software, recognizing it as an abstract information artifact. To support this proposal the first main contribution of the dissertation is demonstrated from three dimensions: (1) We distinguish software (non-physical object) from hardware (physical object), and demonstrate the idea that the rapid changing speed of software is supported by the easy changeability of its medium hardware; (2) Furthermore, we discuss about the artifactual nature of software, addressing the erroneous notion, software is just code, presents both in the ontology of software literature and in the software maintenance tools; (3)At last, we recognize software as an information artifact, and this approach ensures that software inherits all the properties of an information artifact, and the study and research could be directly reused for software then.

Secondly, we propose an ontology founded on the concepts adopted from Requirements Engineering (RE), such as the notions of World and Machine phenomena. In this ontology, we make a sharp distinction between different kinds of software artifacts (software program, software system, and software product), and describe the ways they are inter-connected in the context of a software engineering process. Additionally, we study software from a Social Perspective, explaining the concepts of licensable software product and licensed software product. Also, we discuss about the possibility to adopt our ontology of software in software configuration management systems to provide a better understanding and control of software changes.

Thirdly, we note the important role played by assumptions in getting software to fulfill its requirements. The requirements for most software systems -- the intended states-of-affairs these systems are supposed to bring about -- concern their operational environment, usually a social world. But these systems don't have any direct means to change that environment in order to bring about the intended states-of-affairs. In what sense then can we say that such systems fulfill their requirements? One of the main contributions of this dissertation is to account for this paradox. We do so by proposing a preliminary ontology of assumptions that are implicitly used in software engineering practice to establish that a system specification S fulfills its requirements R given a set of assumptions A, and our proposal is illustrated with a meeting scheduling example.

Keywords
[software, assumptions, requirements engineering, ontology]

Acknowledgements

It has been more than four years since I started to pursue my Ph.D. degree. It was really a pleasure to do the study and research in Trento, where is a beautiful and quiet town in mountain. At the end of my Ph.D. study period, I would like to express my sincere thanks to the people who supported me all the time, without their kind help this thesis would not have been possible.

Firstly, I would like to say many thanks to my advisor John Mylopoulos. Thank you for bringing me into the field of research[1], for teaching me the way of thinking a researcher should possess, for showing me the working attitude a researcher should take, for shaping me from a student into a researcher in every possible aspect. I shall admit that this is the biggest challenge I have ever encountered in my life, and without you I should never have a chance to conquer it.

I also would like to say thanks to my co-advisor Nicola Guarino, and to my co-author Giancarlo Guizzardi in all my publications during the Ph.D. period. I enjoyed the group meetings we had with all of our four together, and frankly speaking it was not easy to follow the ideas of three senior researchers simultaneously, but it was really a good practice to examine a research subject from different perspectives as much as possible, resulting in the understanding of it as complete as possible. Besides the group discussions, you also showed me the way of term work, and kindly provided many detail comments in my writings, without your help I couldn't have completed my conference papers.

My thanks are also given to my dear colleagues and friends at the University of Trento. The regular seminar was always an excellent place to share ideas, and after the working hours it was your accompany that makes me feeling not alone here. Four years is not a short period, and it was really my pleasure to share the happy memories with you.

I always can't say too many thanks to my family, especially to my father who has been working hard for the whole life to bring my two brothers and me up. You are always the light in my life, you are the source of my energy, and I wish you are always happy and healthy.

Thanks all the people who provided kind help!

[1] Support for this work was provided by the ERC advanced grant 267856 for the project entitled 'Lucretius: Foundations for Software Evolution' (http://www.lucretius.eu).

Contents

1 Introduction 1

1.1 The Context and Motivations 1
1.2 Research Problems 2
1.2.1 What is Software? 2
1.2.2 How to Identify and Record Different Kinds of Software Changes? 2
1.2.3 How Does Software Change the World? 3
1.3 Contributions 5
1.3.1 Software as an Information Artifact 5
1.3.2 Identify and Record Changes in Different Kinds of Software Artifacts 6
1.3.3 Assumptions as a Bridge between the World and Machine 7
1.4 Structure of the Dissertation 8
1.5 Corresponding Publications 9

2 Related Work 11

2.1 Understanding Software 11
2.1.1 Software Interpreted in a General Sense 11
2.1.2 Software Interpreted in a Limited Sense as Computer Software 12
2.1.3 Software Interpreted as an Artifact 18
2.2 Understanding Software Change 20
2.2.1 Laws of Software Evolution 21
2.2.2 The Metaphor between Software Evolution and Biological Evolution 22
2.2.3 Taxonomies and Ontologies for Software Change 23
2.2.4 Identifying and Recording Software Changes 27
2.3 Understanding Assumptions 29
2.3.1 Interpretations from Linguistic and Cognitive Science Perspectives 29
2.3.2 van Lamsweerde's Interpretation of Assumptions in Requirements Engineering 30
2.3.3 Lewis's Assumption Management System 32

3 Baseline 35

3.1 DOLCE Adopted as the Foundational Ontology 35
3.2 Artifacts 38
3.3 World and Machine Framework 40
3.4 Situation Calculus 43

4 Towards an Ontological Analysis of Software 47

4.1 Software Changeability and Hardware Changeability 47
4.2 Software as an Artifact: from Code to Programs 49
4.3 Software as an Information Artifact 51

5 From Software Programs to Software Products 53

5.1 Identifying Different Kinds of Software Artifacts 53
5.1.1 From Software Programs to Software Systems 53
5.1.2 From Software Systems to Software Products 55
5.1.3 The Social Nature of Software 57
5.2 Recording Different Kinds of Software Artifacts 59
5.2.1 Representation of Different Kinds of Software Artifacts 59
5.2.2 Ontology-Driven Software Configuration Management 60

6 How Software Changes the World through Assumptions 63

6.1 A Preliminary Ontology of Assumptions 64
6.1.1 A Classification of Assumptions 65

6.1.2 The Causal Chain Enabled by these Assumptions 66
6.1.3 Interpretations of the Concept of Assumption 69
6.2 The Meeting Scheduler Case Study 71
7 Conclusions and Future Work 77
Bibliography 81

List of Figures

Figure 1. The semiotic triangle, adapted from (Ogden et al., 2001) 5
Figure 2. The original program abstractions taxonomy adopted from (Eden & Turner, 2007) 16
Figure 3. The revised program abstractions taxonomy 17
Figure 4. Clarifying the polysemy of the term 'software' 18
Figure 5. Program as a sub-class of both Computer Language Expression and Artifact of Computation 19
Figure 6. Laws of Software Evolution, as proposed by Lehman (M. Lehman & Fernandez-Ramil, 2006) 22
Figure 7. An overview of domain factors affecting software maintenance (Kitchenham et al., 1999).....25
Figure 8. An overview structure of the ontology for software maintenance project (RUIZ et al., 2004).....26
Figure 9: An overview of ontology for software maintenance (Anquetil et al., 2007) 26
Figure 10. Further distinction among statements in the problem world (vanLamsweerde, 2009) 31
Figure 11. Syntax structure in Lewis's assumption management system 32
Figure 12. A diagram of DOLCE (Masolo et al., 2003a) 37
Figure 13. Artefactual objects, artefactual kinds and artefactual roles (Guarino, 2014) 38
Figure 14. Reference model for requirements engineering (Gunter et al., 2000) 41
Figure 15. RWSMP framework composition rules 42
Figure 16. World and Machine Framework (WM Framework).....43
Figure 17. An example of a situation transition 44
Figure 18. A transition between two situations according to World and Machine Framework 45
Figure 19. An early Pascaline on display at the Musée des Arts et Métiers, Paris 47
Figure 20. An extract of Jackson and et al.'s WRSPM Framework.....54
Figure 21. Cutting the boundary between worlds and machines (according to WM Framework).....54
Figure 22. Different abstract software artifacts induced by different requirements engineering notions. 56
Figure 23. WM Framework with assumptions 65
Figure 24. The chaining mechanism underlying software engineering enabled by assumptions.....67
Figure 25. A goal model of meeting scheduler system 72

Chapter 1

Introduction

1.1 The Context and Motivations

Software now permeates all aspects of personal and social activities, improving productivity, quality of service, and quality of life for billions of people worldwide. This reliance on software means that it is essential for users -- be they companies, governments, hospitals, or individuals -- that software is kept running. However, the environments of these software applications are continuously changing and so are stakeholder requirements. To survive in such a setting, software needs to continuously evolve.

According to several surveys (Jarzabek, 2007), (Pfleeger & Atlee, 2009), (Kontogiannis, 2010) in the literature, the average cost of software maintenance covers more than 50% of the total budget in a software project. This is largely due to the fact that design knowledge about a software system is lost or forgotten as its developers drift away. Another factor that makes software evolution difficult and expensive is that as software is changed, its quality deteriorates, making it more complex to understand. Hence, maintainers usually spend 40% to 60% of their time to understand the software being maintained (Gašević, Kaviani, & Milanović, 2009). Making things even worse, stakeholders usually understand a software system from their own perspectives. Much of their knowledge is implicit and hard to communicate to the designers. Without making this knowledge explicit, it is hard to answer important questions about software maintenance (Kitchenham et al., 1999).

Although software engineers have been suffering from such kinds of missing knowledge for a long time, software maintenance tools such as Concurrent Versions System (CVS) and Apache Subversion (SVN), the version control systems of choice for almost 30 years, are used primarily for code management and evolution, while requirements, architectural specifications etc. are left out in the cold. It is such code-oriented practices of software maintenance that results in so much knowledge about changes being left unrecorded.

To tackle the aforementioned problems, missing knowledge should be captured and made available to its maintainers. In order to accomplish this, we must first change our conceptualization of software so that it no longer viewed as mere code. This thesis proposes to tackle precisely this problem by exploring an ontology of software that accounts for more than its codebase. Specifically, we propose to study three fundamental questions in this dissertation: (1) What exactly is software; (2) How we can identify and record different kinds of software changes; (3) How can software that operates within a machine change the world by, for example, scheduling a meeting?

1.2 Research Problems

1.2.1 What is Software?

To answer this question through ontological analysis, we need to check the essential properties of software. To do so, we need to distinguish the scenario in which software is changed while keeping its identity, from the scenario in which new software is created due to the changes.

For many, both inside and outside the software engineering community, software is just code, something intangible best defined in contrast with hardware. For example, the Oxford English Dictionary defines software as 'the programs and other information used by a computer' and other dictionaries adopt similar paraphrases.

The question we have posed as title for this sub-section admits several different answers, such as 'source code to be executed on a computer' or 'instructions used for managing tangible objects' or some other answers. This demonstrates that the meaning of the concept of 'software' is still under discussion. In other words, currently there is no shared common understanding of what software is among researchers and practitioners.

Recently, some researchers have proposed to interpret software as an information object (Oberle, 2006), (Smith et al., 2013). This is a promising direction for understanding the nature of software. However, as not enough attention has been paid in this area after decades of study on this topic, it seems that there is still ambiguity about the nature of information, of software and the relationship between them.

As we already mentioned, without a shared understanding of software, it is hard to precisely communicate about any serious question relating to the nature of software among researchers and practitioners. Fortunately, ontological analysis possesses the capability to capture knowledge explicitly and unambiguously, and we propose to use it to understand software as an information artifact. Based on this understanding, we propose an ontology of software capturing the essential properties of software.

1.2.2 How to Identify and Record Different Kinds of Software Changes?

Software changes all the time. Such changes have huge impacts on the software applications, so dealing with software changes is absolutely necessary. In the past, a few researchers have proposed some taxonomies intending to describe the different kinds of software changes (Swanson, 1976), (Chapin, Hale, Kham, Ramil, & Tan, 2001), (Buckley, Mens, Zenger, Rashid, & Kniesel, 2005), but the very nature of software changes is still unclear: What does it mean for software to change? How do we tell that, after a change, it is still the same software or new software is created? The very possibility for software to change while maintaining its identity is in practice ignored by most recent studies, which have mainly focused on the relationships between software code (intended as an abstract information pattern), its physical encoding, and its execution (Eden & Turner, 2007).

Unfortunately, treating software as simply code is not very illuminating. Microsoft (MS) Word turned 30 years old in 2013. During its lifetime it has been numerously changed, as its requirements, code and documentations have continuously evolved. If software is just code, then MS Word of today is not the same software as the original MS Word of 1983. But this defies the common sense that views software as

a persistent object intended to produce effects in the real world, which evolves through complex social processes involving owners, developers, salespeople and users, having to deal with multiple revisions, different variants and customizations, and different maintenance policies. Indeed, software management systems were exactly intended to support such complex processes, but most of them consider software just as code, dealing with software versioning in a way not much different than ordinary documents: the criteria underlying the versioning scheme are largely heuristic, and the change rationale remains obscure.

Yet, differently from ordinary documents, software changes are deeply bound to the nature of the whole software development process, which includes both a requirements engineering phase and subsequent design and implementation phases. This means that, making a change to a software application may be motivated by the need to fix a bug, to adopt a more efficient algorithm or improve its functionality, adapt it to a new regulation and so on. As a result, different kinds of software changes are separated from each other and treated with different kinds of methodologies and technologies.

Although the idea of classifying software changes into different kinds is a promising contribution in providing guidance for software engineers with different purposes, the ambiguities in the concepts make it difficult to be efficiently applied in practice, as researchers and practitioners hold their own criteria in classifying software changes, and sometimes no clear distinctions are provided but intuitions are adopted as they like. For example, the difference between the terms ‘software evolution’ and ‘software maintenance’ is usually vague. Sometimes, they are used interchangeably (Chapin et al., 2001); sometimes, maintenance subsumes evolution (Bennett & Rajlich, 2000); sometime, evolution subsumes maintenance (Godfrey & German, 2008); or more abstract words ‘change’ or ‘aging’ are used to avoid the misinterpretations (Buckley et al., 2005), (Parnas, 1994). Besides that, the interpretations of other relating concepts, such as ‘software reengineering’, ‘software refactoring’ and ‘software adaptation’, are also treated ambiguously.

To remedy this situation, as we shall see in the following parts of this dissertation, we recognize different kinds of software changes that affect different kinds of *software artifacts* created within a software development process. As a solution, we shall present an ontology of software that describes what these different software artifacts are, and furthermore identify and record the different kinds of software changes according to their effects on different kinds of software artifacts respectively.

Currently, the tools and methods used to manage software changes are usually designed as file-based, and this limits their capability to track the semantics of the changes. Taking the concurrent versions system (CVS) as an example, it compares files by lines. In other words, it is only a syntax comparing tool without providing any higher semantics. To tackle this problem, this dissertation tries to show the possibility of adopting a suitable language (for the particular purpose of a software engineer) to represent the history of the changes based on the ontology of software stated above, and this could be integrated within the existing and future tools for managing software changes with higher semantics. We believe that it will be will be helpful to provide such knowledge about software during its changes for the software engineers, keeping their knowledge about the software updated, or recalling the forgotten knowledge easier.

1.2.3 How Does Software Change the World?

In addition to the essential properties of the different kinds of software artifacts recorded during the different kinds of software changes aforementioned, there is another kind of knowledge that deserves spe-

cial attention: these are the assumptions made during the software engineering process. Without explicit representations of these assumptions, the description about the software is incomplete, something that can result in great difficulties when managing a software system, as argued below.

Consider a software application that schedules meetings upon request. Its basic requirement, which the application is mandated to fulfill, is to bring about a change in the social world within which it operates that consists of a new meeting that satisfies timetable and other constraints provided by the requester. But the software program, by its very nature, can only change the states of the machine within which it operates.

There seems to be a paradox here. The requirements for most software systems, the intended states-of-affairs these systems are supposed to bring about, concern their operational environment, usually a social one. But these systems don't have any direct means to change that environment in order to bring about the intended states-of-affairs[2]. In what sense then can we say that such systems fulfill their requirements?

It seems that a software program possesses a peculiar characteristic compared with other kinds of information artifacts (e.g. recipes or laws) in that it plays the role of a bridge between the abstract states of a machine and the outside world. More specifically, other kinds of information artifacts directly manipulate the objects in the world; instead of that, software program directly manipulates the virtual variables in a machine, and in turns, the result of this manipulation in the machine affects the outside world.

A software program is embedded and operates in a machine, and in this sense machines are software-driven. However, the purpose of a software program (its requirements) is usually intended to affect the phenomena of its environment external to the software-driven machine. This machine monitors and controls the environment by means of transducers bridging the gap between symbolic data and physical properties. For simplicity, we hereafter refer to the software-driven machine as machine, following (Michael Jackson, 2000) and (Axel Van Lamsweerde, 2009).

In the case of a stand-alone personal computer (PC) such transducers only concern the human-computer interface and the standard I/O devices; for mobile systems they may also include location and acceleration sensors, while in the case of embedded systems they take the form of ad-hoc physical sensors and actuators. So, in the general case, the software's ultimate purpose is achieved by running a software program that produces certain effects inside a computer, which drives a physical machine, which in turn produces certain effects on its external environment.

Understanding this indirect effect of software on the world is essential, as our modern society depends on software for almost every aspect of our lives (e.g. in business, hospital and et al.). A money transfer from a person to another through software, becomes a data change in one account and a corresponding data change in another account, even if there is no physical object, in the form of a paper receipt. Moreover, in most modern financial systems, only about 3% of the money exists in paper form, while the other 97% is just electronic data stored in computers (Ryan-Collins, Greenham, Werner, & Jackson, 2014).

[2] We are focusing on 'pure' software systems that consist of software and various interfaces, as opposed to cyber-physical systems (such as robots, drones, etc.) that consist of software and mechanical/robotic components, which do give them the capability to directly change their physical environment.

Several researchers, (Lewis, Mahatham, & Wrage, 2004), (Mamun & Hansson, 2011), (Brown, 2006), (Tun et al., 2015), have emphasized the importance of assumptions, and have proposed techniques for capturing them. Our proposal goes further in that direction as it identifies new classes of assumptions (notably, the dependence ones) that had not been previously accounted for.

1.3 Contributions

1.3.1 Software as an Information Artifact

To better understand the nature of software, we discuss its ontological nature, interpreting it as a special kind of information object. Focusing on informational aspects of software, several researchers, (Eden & Turner, 2007), (Oberle, 2006), have addressed the complex relationships among i) software *code*, consisting of a well-formed expression of a set of computer instructions; ii) a software *copy*, which is a physical inscription of the code; and iii) a *medium*, the hardware medium itself; iv) a *process*, which is the result of executing the software copy.

These works can be viewed as applications of the semiotic triangle proposed by (Ogden, Richards, Malinowski, Constable, & Crookshank, 2001) to express the information communication processes between agents. For example, as shown in Figure 1, a speaker may say the word "Dog" to denote a concept in her mind, and this concept refers to animal dogs in the world; then, this word may invoke a similar concept in the listener's mind referring to animal dogs, as intended by the speaker.

Applying this idea to software, during a software engineering process, a software program is usually encoded in some programming language and corresponds to the symbol in the triangle, and this symbol represents some instructions as the knowledge or concept held in stakeholder minds. This is a simple demonstration to present the intuition and flavor of the rationale why we interpret software as an information object, further detail and concrete explanations are left as one of our main contributions in Chapter 4.

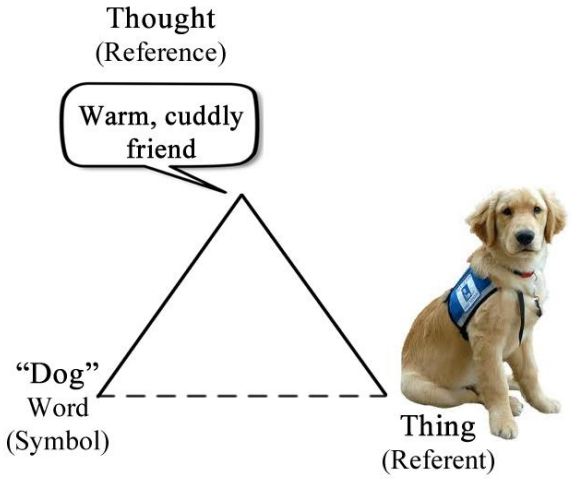


Figure 1. The semiotic triangle, adapted from (Ogden et al., 2001)

Yet, according to (Irmak, 2013), software is synonymous to program and can be understood in terms of the concepts of algorithm, code, copy and process, but none of these notions can be identified with

software, mainly because, due to its artifactual nature, software has different identity criteria. We share many of Irmak's intuitions, as well as the methodology he adopts to motivate his conclusions, based on an analysis of the condition under which software maintains its identity despite change.

Hence, another main contribution of this dissertation consists of an argument, supported by ontological analysis, that software has a complex artifactual nature, as many artifacts result from a design process, each having an intended purpose that characterizes its identity. This is what distinguishes software artifacts from arbitrary code: they are recognizable as having purposes, and they are the results of intentional acts. Combining the informatical nature and artifactual nature of software, we interpret software as a special kind of information artifact, inheriting the essential properties of both an information object and an artifact.

1.3.2 Identify and Record Changes in Different Kinds of Software Artifacts

Based on the analysis of the ontological nature of software as an information artifact as aforementioned, we are going to answer the research questions left open by (Irmak, 2013): 'work still needs to be done on questions such as how software changes, what the identity conditions for software are, and more'. So we shall focus on the identity criteria for software originated by its specific artifactual nature, and motivated by the need to properly account for software changes.

We start with studying a peculiar aspect of software with respect to other information artifacts such as laws or recipes, as (Eden & Turner, 2007) observe, it is the *bridging role* of it between the abstract machine and the concrete world: despite the fact that it has an abstract nature, it is designed to produce specific results in the world. Therefore, it seems natural to us to adopt a requirements engineering perspective while analyzing the essence of software, looking at the whole software engineering process, including requirements analysis, instead of focusing on its computational aspects only. Our analysis is founded on a revisit of Jackson and Zave's seminal work on the foundations of requirements engineering (Michael Jackson & Zave, 1995), (Zave & Jackson, 1997), (Gunter, Jackson, & Zave, 2000), which clearly distinguishes the *external environment* that constitutes the subject matter of requirements, the (computer-based) *machine* where software functions fulfill such requirements, and the interface between the two.

Jackson and Zave define the terms 'requirements' and 'specification' as referring to the intended behaviors in the environment and at the interface, respectively. Here we refine their terminology using 'requirements' to refer to the intended behaviors in the environment independently of the machine, *excluding* therefore the interface, 'external specification' to point to the expected behaviors *at the interface*, and 'internal specification' referring to the specific behaviors *inside the machine*, namely that of the system that drives the machine.

As shown in Table 1, we shall rely on these refined notions to determine the essential properties of three different kinds of software artifacts: *software products*, *software systems*, and *software programs*. In addition, to account for the social nature of software products in the present software market, a further kind of software artifact will be introduced, namely *licensed software product*, whose essential property is a mutual pattern of commitments between the software vendor and the software customer.

Table 1. Essential properties of software artifacts

Object	Essential Properties
Licensed Software Product	Mutual Vendor-Customer Commitments
Software Product	Requirements
Software System	External Specification
Software Program	Internal Specification
Code	Syntactic Structure

As an output of the preceding ontological analysis, we propose an ontology of software to capture the essential properties of different kinds of software artifacts. This ontology could be used to semantically annotate the logs of software behaviors, or to document empirical studies so that they can be classified, understood and replicated, or serve as a groundwork to develop the evolution oriented model processes, requirement engineering, workload assignment and other software engineering activities.

For instance, traditionally, revisions and variants are managed by means of naming conventions and version codes which are usually decided on the basis of the *perceived* significance of changes between versions without any clear criterion (e.g. CVS, SVN). We believe that the classification of different kinds of software artifacts introduced in this dissertation can make an important contribution to make this process more disciplined by providing a general mechanism to explicitly express what is changed when a new version is created.

1.3.3 Assumptions as a Bridge between the World and Machine

As mentioned in the research problems, in addition to the essential properties about the different kinds of software artifacts, there is another kind of knowledge deserves special attention, and it is the assumptions made during a software engineering process. Any machine designed to solve a problem makes assumptions. Some of these assumptions capture expectations about the world that are always supposed to be valid, such as natural laws, and can be exploited in the design. Other assumptions circumscribe the limits of the solution. For example, for meeting scheduling, we may assume that there are enough rooms available for all meeting requests and design a solution that only finds a suitable time slot and selects a room. Such an assumption means that our solution may not work when there is no room available (e.g. during a busy period with many meeting requests).

Yet other assumptions may relate the interface behaviors to some expected behaviors in the world. For instance, we may assume that, if the computer says (by means of a suitable message on the screen) that a certain room is reserved for a certain meeting at a certain time, the room will not be used for any other meeting at that time. However, this system doesn't have any direct means to change that environment in order to bring about the intended states-of-affairs, and the paradox aforementioned lies here. Hence, without explicit representations of these assumptions to clarify the world, the machine, and the relations between them, the description about the software is incomplete, which could result in the difficulty in managing a software application.

According to the statements above, the main purpose of this contribution is to account for this paradox. We do so by proposing a preliminary ontology of assumptions that are implicitly used in software

engineering practice to establish that a system specification *S* fulfills its requirements *R* given a set of assumptions *A*. Adopting the formula of the requirements problem proposed by (Michael Jackson & Zave, 1995), our task is to characterize the assumptions used and needed to establish that

$$A, S \vDash R$$

given that the requirements are about world states (e.g., meetings, participants, timetables, rooms, and etc.), while the specification is about machine states (database tables, tuples) and manipulations thereof.

Several researchers , (Lewis et al., 2004), (Mamun & Hansson, 2011), (Brown, 2006), (Tun et al., 2015), have emphasized the importance of assumptions, and have proposed techniques for capturing them. Our proposal goes further in that direction, as it identifies new classes of assumptions (notably, the dependence ones) that had not been accounted for, and the specific contributions in this part of the dissertation are listed as follows:

1) A preliminary ontology of assumptions is proposed, introducing four kinds of assumptions. Two of them are proposed based on the literature work, including world assumptions and machine assumptions. Taking a further step, two new kinds of assumptions are discovered and integrated into the ontology, including world dependence assumptions and machine dependence assumptions respectively. We claim that these four kinds of assumptions are the key to solve the aforementioned paradox, and we elaborate the role of them in linking the world states and machine states together;

2) We clarify the concept of 'assumption', identifying two possible senses of interpretations that are both important for software engineering processes, namely the *assumptions-used* and the *assumptions-needed*, and provide an update in Jackson and Zave's original formula to capture the software engineering activities more precisely;

3) We discuss how our results can be employed methodologically, suggesting how software developers should systematically and explicitly manage all the four kinds of assumptions proposed here. We suggest that these assumptions should be explicitly identified and systematically guaranteed to hold throughout the useful lifetime of their software system.

1.4 Structure of the Dissertation

To present the contributions summarized in the preceding sub-chapter, we have arranged the rest contents of the dissertation as follows:

Chapter 2 serves as a survey and review of the related work, including their understanding of software, their understanding and management methodologies of software changes, and their understanding of assumptions.

Chapter 3 introduces our research baseline that is taken as the starting point of this dissertation, consisting of the adopted basic ontological concepts, the World and Machine Framework derived from the reference model for requirements and specifications proposed by Jackson and et al. (Gunter et al., 2000),

and situation calculus adopted as our representation language proposed by (McCarthy & Laboratory, 1963).

Chapter 4 discusses the ontological nature of software, distinguishing it from hardware, and also demonstrating the idea that the rapid changing speed of software is supported by the easy changeability of its medium hardware. Meanwhile, we also discuss about the ontological nature of information artifact, and show in what sense software could be recognized as an information artifact.

Chapter 5 proposes a preliminary ontology of software. Three different kinds of software artifacts are identified according to their essential properties, including software products, software systems, and software programs. This classification is developed based on the idea of cutting the world and the software-driven machine with a clear boundary, we name it as WM framework which is derived from the reference model for requirements and specifications proposed by Jackson and Zave. Different kinds of software artifacts refer to the phenomena in the different parts of WM framework (outside world, interface, and inside machine). In addition, there is a fourth kind of software artifact reflecting the social nature of software products, whose essential properties are based on the mutual commitments between vendors and customers. Besides contributing to clarify concepts and terminologies in the software engineering community, we also demonstrate the possibility that our work could also be used as a foundation for software change management, especially for identifying and recording the changing histories of these different kinds of software artifacts.

Chapter 6 proposes a preliminary ontology of assumptions, illustrating the four kinds of assumptions that enable the link between the world and machine crossing the boundary between them. Also, in this chapter, we explain our interpretation of the assumptions in the formula '$A, S \vDash R$' as 'assumptions-used' and 'assumptions-needed', from which we can derive the importance of making such assumptions explicit, and distinguishing these two kinds of interpretations from each other. At the end of this chapter, we propose a meeting scheduling case study in situation calculus representing the requirements, external specification, and internal specification, and meanwhile elaborating on the role of these assumptions in establishing the link between the world and machine states.

Chapter 7 summarizes the main contributions of this dissertation and indicates the possible promising directions of future work.

1.5 Corresponding Publications

Recognizing software as an information artifact:

Wang, X., Guarino, N., Guizzardi, G., & Mylopoulos, J. (2014). Software as an Information Artifact What is An Information Artifact. In Workshop on Information Artifact Ontologies. Rio de Janeiro.

Developing an ontology of software:

Wang, X. (2012). An Ontology of Software Evolution. In R. Cornet & R. Stevens (Eds.), Proceedings of the 3rd International Conference on Biomedical Ontology (ICBO 2012), KR-MED Series. Graz. Retrieved from http://ceur-ws.org/Vol-897/

(Runner-up for best paper award, out of 88 submissions.) Wang, X., Guarino, N., Guizzardi, G., & Mylopoulos, J. (2014). Towards an Ontology of Software: a Requirements Engineering Perspective. In O. K. Pawel Garbacz (Ed.), 8th International Conference on Formal Ontology in Information Systems (pp. 317–329). Rio de Janeiro: IOS Press. http://doi.org/10.3233/978-1-61499-438-1-317

Wang, X., Guarino, N., Guizzardi, G., & Mylopoulos, J. (2014). Software as a Social Artifact: A Management and Evolution Perspective. In E. Yu, G. Dobbie, M. Jarke, & S. Purao (Eds.), 33rd International Conference on Conceptual Modeling (Vol. 8824, pp. 321 – 334). Atlanta: Springer International Publishing. http://doi.org/10.1007/978-3-319-12206-9_27

Explaining the role of assumptions in linking world and machine states:

(Accepted) Wang, X., Guarino, N., Guizzardi, G., & Mylopoulos, (2016). How Software Changes the World: the Role of Assumptions. Proceeding of IEEE Tenth International Conference on Research Challenges in Information Science, Grenoble, France.

Chapter 2

Related Work

2.1 Understanding Software

2.1.1 Software Interpreted in a General Sense

In the literature of Computer Science, the earliest use of the term 'software' is attributed to (Tukey, 1958), who was also famous for proposing the term 'bit' for an atomic data unit (Buchholz, 2000). The term 'software' was incorporated into the *Oxford English Dictionary* in 1960. Historically, this term was used in a more general way, independently of computers. For example, it was used by rubbish-tip pickers around 1850 to indicate vegetable and animal matters that are decomposable.

Many interpretations of the term 'software' were proposed by researchers, and some of which are listed and discussed below.

Osterweil believes that, in addition to computer software, there are other kinds of software, such as processes, recipes, laws, assembly instructions, and driving directions (Osterweil, 2008). He characterizes all these kinds of software as follows:

'While software is itself non-physical and intangible, a principal goal for instances of the type software is for them to contain one or more components whose execution effects the management and control of tangible entities (Osterweil, 2008)'.

Following Osterweil's account above, software can be divided into two main categories, computer software and other kinds of software. The categorization is intuitive: computer software is intended to be executed on a computer, while other kinds of software execute on different physical manifestations other than computers. For example, laws are intended to be executed by government bureaucracies, and recipes are intended to be executed on cooking devices (Osterweil, 2008) .

By examining the characteristics of different kinds of software, Osterweil proposes an interesting view that computer software engineering can contribute to other forms of software engineering, and oppositely, computer software engineers can learn a lot from the study of other forms of software. For example, software engineering formalisms and approaches could be applied to laws, such as the attempt with a workflow language (Georgakopoulos, Hornick, & Sheth, 1995). On the other hand, general project process management methods could also be good lessons for computer software development processes.

Suber interpreted software[3] as an even more general concept based on his interpretation of the term 'pattern' (Suber, 1988). For him, software is any abstract pattern formulated/stored in a medium, and could be the embodied medium itself. In his proposal, 'pattern' is used in a broad sense that anything '… signifying any definite structure, not in the narrow sense that requires some recurrence, regularity, or

[3] Suber used the terms "program" and "software" interchangeably.

symmetry.' In a word, whenever there is a difference existing in the current situation, there exists a pattern accordingly.

Then, based on this definition of pattern, he interprets software as 'patterns, readable and executable by a machine, and liftable.' According to this interpretation, we can derive some extremely counter-intuitive cases, such as the ones stated by Suber himself that 'all circuits deserve the name software, since they are physical embodiments of patterns, readable and executable by a machine, and liftable' and 'firmware is one of the most important examples of hardware that is software.'

However, we want to point out that Suber doesn't distinguish a 'pattern' from 'the physical embodiments of the pattern', and this brings about ambiguities in understanding his interpretation of software, as it is defined based on the concept of 'pattern'. For him, there is no difference between software and hardware, yet we do not want to mix the boundary to such an extreme scale. For us, to recognize a hard disk as software is quite counter-intuitive, and it is more desirable to separate the physical medium as hardware, from the abstract representations (as patterns) which are materialized in the hardware medium. For example, in the situation where some sentences are printed on a piece of paper, the writing structure is not equal to the ink and the paper, as the same structure could also be shown on a monitor which is a total different hardware medium.

2.1.2 Software Interpreted in a Limited Sense as Computer Software

Although the ideas stated in the preceding paragraphs are certainly intriguing, we focus on a proper ontological account of computer software, which is still missing in the literature. Focusing on the computational aspects, several scholars have addressed the complex relationships among i) a software code, understood as a set of computer instructions; ii) a software copy, which is the embodiment of a set of instructions through a hard medium; iii) a medium, the hardware medium itself; iv) a process, which is the result of executing the software copy.

Moor's Work

Moor's work is a good point to start with, as he believes that to understand software-related notions, one needs to understand the conceptual framework of Computer Science, and if the notions are misunderstood, sloppy research conclusions might be derived, especially in the realm of Artificial Intelligence (AI) (Moor, 1978).

Moor interprets a computer program from two levels: 1) from the physical level, computer programs can be embodied in the form of series of holes in punched cards, configurations on magnetic tape, or in any number of other forms; 2) from the symbol level, computer programs could be understood as symbolic representations of instructions to a computer.

Note that, the computer programs stated above possess both the physical and symbolic characteristics at the same time, as Moor doesn't separate the symbolic representation of instructions from the embodiment of the symbolic representation in some physical medium. Hence, according to that, the changeability to the instructions is reduced into the changeability to the embodiment of the instructions.

Underlying the above understandings of a computer program, Moor proposed his definition of it as 'a computer program is a set of instructions which a computer can follow (or at least there is an acknowledged effective procedure for putting them into a form which the computer can follow) to perform an activity.'

As we can see, this definition is based on an unexplained preliminary term 'computer', and the task of deciding what is a computer is left as a practical question for the software engineers. In other words, to judge the identity of a computer program, a context containing a person and a computer should be given first. The representations of the instructions as computer programs could be embodied in any forms, as long as it could be accepted and processed by the computer, or could be effectively transformed into some forms that could be accepted by the computer.

Besides the understanding of a computer program, there is an orthogonal pair of concepts 'software' and 'hardware' proposed by Moor. As aforementioned, the changeability to the instructions is reduced into the changeability to the embodiment of the instructions for Moor. Consequently, for a computer program as a set of instructions, he interpreted it as software or hardware according to the changeability to the instructions possessed by the software engineers or the users of it. For example, in an extreme condition, a person at a factory who can replace circuits in a computer understands her activity as giving instructions, then for her the programmable circuits could be interpreted as software.

Although Moor's idea about software and hardware was clearly explained, according to his view the distinction between software and hardware is quite subjective and not stable over time. What is considered hardware by one person, may be regarded as software by another.

We accept Moor's key point that the boundary between software and hardware is illusive. However, we'd like to avoid subjective definitions of software, as they invariably lead to confusion and misunderstandings.

Colburn's Work

Colburn launches his argument with an interesting example to show the importance of developing a clear and shared set of software-related notions. His example was first presented by (Wallich, 1997), talking about a book printed in hard copies with related floppy disks attached. One of the algorithms introduced in the book is a powerful encryption algorithm, and this algorithm was printed both on paper and stored in the attached floppy disks. The U.S. government prohibited the export of the book because the algorithm stored in the floppy disks was so powerful that the government was not able to decrypt the contents encrypted by it. The interesting part is that although the U.S. government recognized the algorithm as a dangerous machine stored in the floppy disks, this book would have been freely exportable without the floppy disks even with the same algorithm printed in the book.

From the preceding example, we can derive the conclusion that although we intuitively share the idea that there is an abstract representation as the same software, the embodiments of it in different physical forms could be treated differently. To address this controversy, Colburn suggests interpreting software as a 'concrete abstraction' (Colburn, 1999). As such, software possesses a dual nature that on one hand it is concrete because it is encoded in physical memory elements, and on the other hand it is abstract, as it is a text representation abstracting itself from any particular physical embodiment.

Colburn cites Hailperin's textbook 'Concrete Abstractions: An Introduction to Computer Science' as an example to reinforce his proposal, as this author intentionally switches between the poles of this duality of software throughout the book (Hailperin, Kaiser, & Knight, 1999). Also, Colburn refers to the dual nature of microphenomena asserted by Copenhagen interpretation in the physics field to show the rationality of interpreting software similarly with a dual nature.

The concrete nature of software proposed by Colburn is intuitive and easy to understand, yet the abstract nature of software was not so clearly explained, as he distinguished software abstraction from mathematical abstraction. For him, although both kinds of abstractions are used to hide details (called contents by Colburn), mathematical abstraction is used to eliminate empirical details, and only focus on the syntactic form transitions, and this is a restriction on their contents. Yet, software abstraction does the opposite, by hiding the details, they provide the possibility to replace or modify the details without affecting the abstractions, and this is an enlargement of their contents. For us, this distinction is subtle and tricky, as mathematical methods could also be used to guide physical implementations, and the only difference between the two kinds of abstraction is that one comes with a compiler, and the other does not. Hence, we don't see the need to distinguish them.

Colburn's proposal is interesting, and might be illuminating. However, the cognitive understanding of the world need not be the same as the world. Some social entities only exist in people's minds, hence making metaphors between different disciplines is not necessarily a useful research method.

On the other hand, the examples about the encryption algorithm and the textbook interpreting software as a concrete abstraction provided by Colburn, could also be used to demonstrate the ambiguities of software-related concepts within many communities of research or practice, and this underscores the importance of providing clear criteria for distinguishing among such software-related notions.

Duncan's Work

Similarly with Moor and Colburn, Duncan also recognizes the dual nature of software, although this was implicitly stated, we still can find clear proof of this from his definition of software. As he clarifies in his paper, the term 'software' should be interpreted as a concept to refer to 'computer programs that are encoded on … physical objects' (Duncan, 2011), and to avoid producing ambiguities, he coined the term 'software program' to replace the original one.

Stated more specifically, a software program consists of a set of instructions written in some programming language. Moreover, this set of instructions should be encoded in some physical medium as patterns, such as holes on a punch card, pattern of 1s and 0s in the magnetic coating of a hard disk, or the pits and lands on a CD.

However, unlike Moor, Duncan's 'software program' should be accepted by a computer directly. For Duncan, a software program generally depends on a kind of 'computational hardware'. As we stated above, a software program must be encoded on some physical medium, and the physical medium must be an instance of the kind of 'computational hardware'. A computational hardware is intentionally designed for computation, for example, a hard disk is designed to be used for computations within a computer, and a piece of paper with printed symbols may not be recognized as a computational hardware, as it is usually

not designed for computations within a computer. This narrows down the range of meanings of a software program, excluding the ones that could not be accepted by the computer directly.

For Moor, a piece of source code printed on a piece of paper could be interpreted as a computer program, as it could be translated into a form as an input to a computer; yet for Duncan, it could not be interpreted as a software program, because the paper is not designed for computations within the computer, and it cannot be accepted by the computer directly.

Another point on which Duncan disagrees with Moor is that Moor adopts the changeability of an entity as the criterion for deciding if the entity is software or hardware, yet Duncan thought this was implausible, given the general ontological nature of them. However, Duncan made a step in this direction, proposing that the ontological dependency of an entity could be used as the criterion to distinguish software from hardware.

More specifically, Duncan states that 'a piece of computational hardware is an ontologically independent entity, whereas a software program is an ontologically dependent entity'. For him, computational hardware can exist independently of any other entity, such as a hard disk, it exists by itself; yet, a software program cannot exists by itself, it must be encoded on an computational hardware instance. For example, a software program could be encoded on many different hard disks, yet when all hard disks are destroyed, the software program ceases to exist.

Duncan's proposals are interesting, yet many issues still need to be dealt with. For example, it might be counter-intuitive for many that a piece of source code printed on a piece of paper cannot be interpreted as a computer program; or, as he proposes, a software program as an entity generally depends on a kind of computational hardware, all the software programs with the same instruction syntax encoded in CDs are identical to each other, yet they are different from the software program with the same instruction syntax encoded in a hard disk, and this also might be quite counter-intuitive.

Eden and Turner's Work

Different from previously discussed researchers who believed that the dual nature of a computer program consisted of an abstract syntax and its physical embodiments, Eden and Turner recognize a similar but different duality of a computer program including the abstract syntax as program-scripts and the executions of the program-scripts as program-processes (Eden & Turner, 2007).

In other words, for them, the term 'program' is therefore polysemic in a different way: a program-script is a well-formed expression based on a Turing-complete programming language[4], which is static (timeless); while a program-process is an execution of a program-script, which is dynamic (extending in time). They describe the relation between a program-script and a program-process as that a program-script is an 'abstraction' from the program-process, or reversely a program-process is a 'concretization' from the program-script.

[4] The notion of 'Turing-completeness' was provided by Martin, which requires a computer program supports a non-trivial set of instructions (Martin, 2010).

As shown in Figure 2, the distinction between a program-script and a program-process contributed as a small part of an ontology of computer programs consisting of several other inter-related notions, and the concept of 'abstraction' aforementioned is the key criterion to distinguish all these notions from each other in the so-called program abstractions taxonomy.

However, for Eden and Turner, the term 'abstraction' was interpreted in a very general sense, and the meanings of it were not stable, as they stated that any of the combinations of the following interpretations was acceptable, including I) Intangible (namely un-touchable), II) Generalized (category vs. elements), III) Underspecified (namely subsumption relationship), IV) Immanently meaningful to humans, V) It from bit (namely instances of information), VI) A-temporal (timeless).

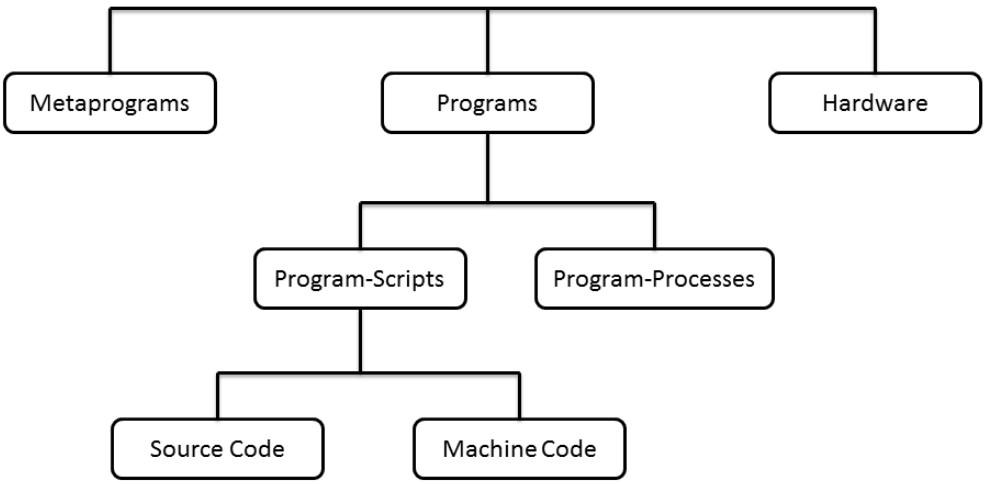


Figure 2. The original program abstractions taxonomy adopted from (Eden & Turner, 2007)

Although there have been many interpretations of the relationship 'abstraction of', as shown in Figure 2, they were all represented by a unique diagram legend (linked solid line without direction). We find this way of representation is a bit misleading, as the readers may be not capable of distinguishing one link from another with a different meaning. To provide a better understanding of Eden and Turner's ideas, we revised their original diagram into a modified version as shown in the following Figure 3. As you shall see, firstly we replace each solid line in the original diagram with an arrowed solid line; then, each of them is labeled with a specified interpretation of the concept 'abstraction'.

For example, a program is abstract from a hardware, as the program is intangible syntax and the hardware is the tangible physical storage/execution medium; a metaprogram is abstract from a program, as a metaprogram is a specification describing the characteristics which a program should possess, hence a possible set of programs could be developed satisfying the specification, and each of these programs becomes a member of this metaprogram; similarly but differently, a program is an abstraction from a program-script or a program-process, as they are both interpreted as subtypes of program, a program-script is a program, and a program-process is also a program.

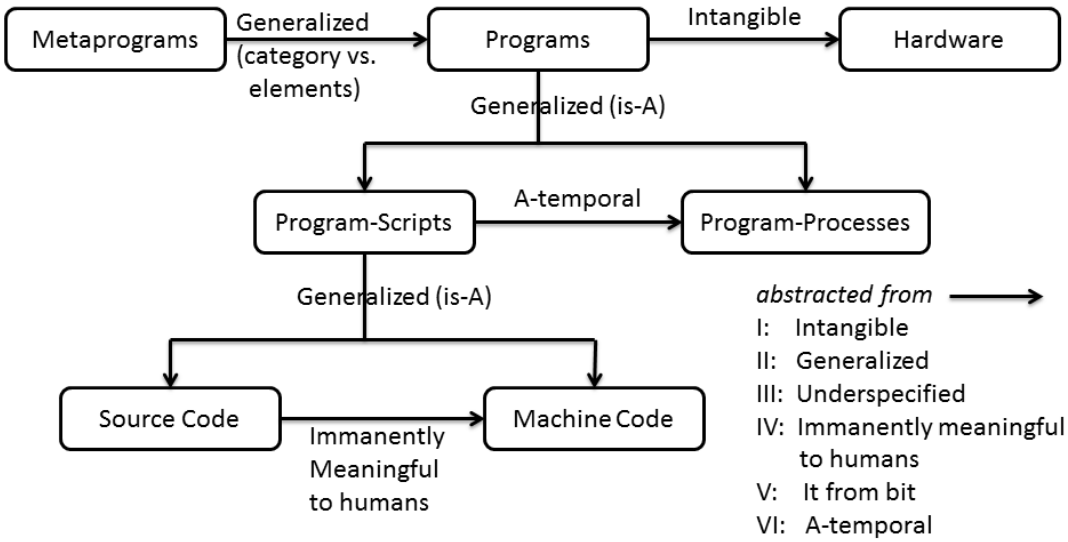


Figure 3. The revised program abstractions taxonomy

In a word, Eden and Turner recognize the pair of concepts 'abstraction and concretization' as the key to understand the computer program related notions, especially they claim that it is its bridging role linking the abstract and the concrete together that provides the unique philosophical identity of software distinguishing itself from other kinds of objects. We share this idea that there exists a paradox underlying the situations where world phenomena can be affected by a computer program that can only change machine states directly, and has no physical means to change the outside world. However, we will try to avoid overloading a term with so many different interpretations, and we shall give a detailed explanation of this paradox by illustrating the role of assumptions in software engineering processes.

Oberle's Work

Similar as the previous reviewed research, Oberle also spotted the ambiguities in the interpretations of software-related notions, and emphasized the importance of clarifying them. According to his view, the term 'software' is heavily overloaded, with at least three different interpretations used in the software engineering community, including the abstract syntax code expression, the physical realization of the code, and the execution of the code, and he coined three terms *SoftwareAsCode*, *ComputationalObject*, and *ComputationalActivity* to refer to these interpretations respectively.

For Oberle, a piece of *SoftwareAsCode* should be the encoding of an algorithm specification. For example, the Bubble Sorting algorithm could be encoded in Java, and the abstract source code in Java is the so-called *SoftwareAsCode* which, Oberle believes, deserves the name 'software' best. This interpretation of software denies the dual nature view of software, and only recognizing the syntactic side of it.

The physical side of the dual nature of software was separated out and referred by the term *ComputationalObject* as the physical realization of the syntactic code in some concrete hardware. The 'realization' mentioned here means the physical inscription or the embodiment of the code, and the supporting physical medium of the realization could be a hard disk, a memory card, or et al. Note that Oberle also

restricts the scope of suitable realizations, as previous researchers did, to the ones that can be loaded and executed by the Central Processing Unit (CPU).

Finally, he proposes that the term *ComputationalActivity* which denotes the software execution processes. Differently from Eden and Turner's view who recognized program-process as a sub-class of program, Oberle adopted the relationships of *realizes* and *participantIn* to link the different software related notions together. As shown in Figure 4, a piece of *SoftwareAsCode* could be *realized* by a corresponding *ComputationalObject*, and whenever the *ComputationalObject* is called and executed in a computer, we can say the *ComputationalObject participates in* a corresponding *ComputationalActivity*.

Figure 4. Clarifying the polysemy of the term 'software'

Yet, differently from others, Oberle proposes his own ontology of software based on two specific principles: 1) developing the ontology on top of some well-formed foundational and domain ontologies; 2) restricting the scope of the ontology as small as possible (Oberle, 2006), (Oberle, Grimm, & Staab, 2009).

The first principle was chosen because well-formed foundational and domain ontologies usually are consisted of preliminary concepts which are well-delimited with clear philosophical analysis. The reliability of these ontologies had been examined and proved by many other researchers in the literature, so that by referring to them, the newly developed ontology could inherit these well-formed ontological commitments directly from the previous work.

The second principle was chosen because there were so many notions used in the software engineering community, and that makes it hard to provide a complete ontology that covers all of them. According to that, he tries to capture the core notions only, and calls his new ontology Core Software Ontology (CSO). This core ontology can be extended in various directions depending on its intended uses.

After reviewing Oberle's ontology, we would say that he makes a significant step in the direction of pinning down different software related notions, separating them from each other, escaping from the trap of mixing two contradictory notions into the so-called dual nature of software.

Final point to notice from Oberle, he recognizes software as an information object based on the work of Gangemi's work (Gangemi, Borgo, Catenacci, & Lehmann, 2004). We share a similar view on this point, yet the ontological nature of information object itself is still under considerable debate, and we will try to clarify this concept as a contribution of this thesis in Chapter 4. Still, there is another missing piece in the puzzle of an ontology of software: the artifactual nature of software.

2.1.3 Software Interpreted as an Artifact

Lando's Work

Similarly as Oberle, Lando builds upon some historical foundational and domain ontologies in the literature and restricts the scope of his ontology to a limited core, Lando and his colleagues develop their

own ontology, named Core Ontology of Programs and Software (COPS). According to this ontology, a program[5] is a *Computer Language Expression* (similar as *SoftwareAsCode*). Besides that, Lando makes a further step in that direction, as he recognizes a program as an artifact. We believe this opens a door to reach a genuine and shared understanding of software (Lando, Lapujade, Kassel, & Fürst, 2007), (Lando, Lapujade, Kassel, & Fürst, 2009).

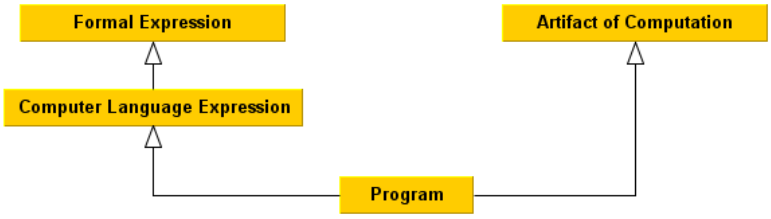


Figure 5. Program as a sub-class of both Computer Language Expression and Artifact of Computation

As shown in Figure 5, a program is a piece of computer language expression, which in turns is a piece of formal expression. By taking this position, similar as previous researchers, a program could be interpreted as a piece of abstract syntactic expression encoded in some programming language. This choice makes the identity of a program depend on its encoding programming language, and this expression should be acceptable by some compiler and then executed by a computer.

More importantly, a program is also interpreted as an artifact of computation by Lando. For him, an artifact is an object[6] to which a function is assigned. A function is some capability assigned by the agent who crafted the artifact, and through the function assignment the agent expresses her purpose to use the artifact to carry out some actions. The actions could affect the physical world or the non-physical world, and Lando believes that a program should be interpreted as an artifact being only capable to modify the non-physical world.

Lando restricted his study within the interpretation of a program as an artifact of computation that can only affect corresponding abstract computing activities. Although this choice is clean and tidy, differently from Lando, we believe it is also important to capture the effects of programs to the physical world. After all, most software applications today are developed to solve real-world problems. According to this view, as we shall see, different kinds of software artifacts are proposed, considering their intentional effects to the different parts among a computer and its outside world.

Irmak's work

Similarly as Lando, yet Irmak proposes a different approach to account for the artifactual nature of software that distinguishes it from other kinds of objects which were closely related to but totally different from software (e.g., algorithm, code, copy, and execution process), yet most of the historical work in

[5] Lando treated the term 'software' differently from 'program', as he interprets a program as a piece of Computer Language Expression, yet software is a collection or library of programs. This distinction seems counter-intuitive to us, and to avoid ambiguity, this distinction is ignored here, and a set of consistent terminology for the software related notions will be provided and discussed as a contribution in Chapter 5.

[6] More precisely, it is an endurant, which is something cannot be instantiated further and can exist in a time point, and keeps its identity during time periods. Details are much more complex and left as explanations in our baseline chapter.

understanding the nature of software fails to capture this point. He disagrees with both the interpretations of software in a general sense or in a limited sense presented earlier, denying the proposals about the dual nature of software, as he believes that such proposals are self-contradictory. Instead of those interpretations, for Irmak, software should be interpreted as an abstract artifact (Irmak, 2013).

Firstly, software is an artifact, and being an artifact it is supposed to be the result of some intentional human activities. Irmak defended his proposal by pointing out that any ontology should be developed considering the common minds and ideas existing in a community. It is the ontologists' job to make the implicit and ambiguous notions explicit and clear, but not to study what is the real true nature of the physical world which is the physicists' job. This claim is intuitive to us, because in the modern society software is indeed developed by software engineers with specific purposes in mind. Randomly-generated source code may be accepted by a compiler by chance, yet it means nothing to human users unless they understand what it does. Similarly, a natural stone from a riverbank is not an artifact by itself, yet whenever it is used as a paperweight on the table, an artifact comes to exist.

Furthermore, software is an abstract artifact. Differently from the Platonic view of abstract objects that are eternal and independent, software as an artifact depends on the intentions of human beings. Unlike Platonic abstract objects that lack both spatial and temporal properties, software only lacks spatial properties, as it possesses temporal ones, and has in addition intentional properties.

To make this more intuitive, Irmak illustrates his ideas through a demonstration of the similarities between software and music. For him, both software and musical works are abstract artifacts, the former is created by software engineers and the latter is created by composers, yet both are created with intentions. Both can be created or destroyed, and when they are destroyed, the following conditions hold: 1) their authors cease to exist in the world, 2) all of their physical copies are destroyed, 3) they are not executed or performed ever again, and 4) they are forgotten by everyone.

This idea of interpreting software as an artifact was also acknowledged by (Turner, 2013) in his recent, comprehensive entry on the philosophy of Computer Science published in the Stanford Encyclopaedia of Philosophy[7]. We also share very much Irmak's intuitions, as well as the methodology he adopts to motivate his conclusions, based on the analysis of the conditions under which a software maintains its identity despite changes. However, he leaves the question of 'what is the identity of software' open, and we shall answer this question in this dissertation. Making a further step, based upon the understanding of software as an abstract artifact, we interpret it as an information artifact, emphasizing both its informational nature and artifactual nature. Besides that, by checking the effects of these software artifacts to the computer states and the world, we classify them into different categories.

2.2 Understanding Software Change

As we mentioned in the beginning of the dissertation, software has become an essential and indispensible part of the modern society. To meet the needs of a rapidly changing society, software has to continuously evolve. (Parnas, 1994) adopted a metaphor to characterize this continuous change of software as 'software aging'. For him, programs get old while time passes by, decaying in their efficiency and

[7] The Philosophy of Computer Science, http://plato.stanford.edu/entries/computer-science/

productivity. We cannot prevent this process, and as a consequence, the older the software gets, the more it costs to be maintained, until it becomes unaffordable and replaced by new software.

Considering the importance of software, and the huge cost of its continuous evolution, researchers have been trying, for decades, to get better understandings of this phenomenon, and to provide proper methods and technologies to manage these changes in software, yet it is still a young and challenging topic because of the undervaluation of it (Mens et al., 2005), (Mens, Gueheneuc, Fernandez-Ramil, & D'Hondt, 2010). In the following, we will go through several representative works in the literature.

However, as the main purpose of this dissertation is not in the area of software evolution, we only discuss how an ontology of software could contribute to better management of software evolution. Specifically, our proposed ontology of software provides the foundation to identify and record different kinds of changes in different kinds software artifacts during their life spans. Hence, within this section, we just make a brief summary of the literature work on software evolution, without diving into detailed discussions. In other words, the main purpose of this section is to provide a general view of the state of the art on software evolution, and locate our work within this framework as a basic and foundational contribution by capturing rich and clear semantics of the software during the processes of software evolution.

2.2.1 Laws of Software Evolution

Lehman was recognized as the 'father of software evolution' by the editors of the journal 'Software Evolution and Feedback: Theory and Practice' (Canfora et al., 2011), as he founded the principles for empirical research on software evolution with his colleague Belady. Together, they studied the evolution processes of IBM's OS 360 (Belady & Lehman, 1976), and extended this study into the famous eight laws of software evolution that profoundly influenced the ways in which software was understood (M. M. Lehman, 1980), (M. M. Lehman, 1996), (M. Lehman & Fernandez-Ramil, 2006).

To start with, Lehman recognized the unexpected and unplanned phenomena occurring during the development and evolution processes of IBM's OS 360, and used these as resource to study software evolution and propose that software should be studied as a natural phenomenon (analogously to physical phenomena). For him, the properties of software are intrinsic and primary, and the effects by human beings are external and secondary. As he stated in an interview, 'evolution process and system control the managers rather than managers controlling the system'.

To nail down the scope of his study, he firstly proposed a classification of software programs, including: 1) S-type programs that possess formal specifications which strictly define the problems to be solved in terms of some programming language, and the S-type programs serve as their solutions; 2) P-type programs that are similar as S-type programs, yet the given specifications for the problems are expressed in terms of the terminology of the problems in the real world instead of the programming language scenario; 3) E-type programs, differently from S-type and P-type programs, treat themselves as a component of the outside problem world. The interpretation of an E-type program is quite similar as the idea of socio-technical system, according to which, an E-type program serves as the technical support within a social environment, communicating and interacting with social agents.

For Lehman, only the E-type programs were adopted as the research subject, as they are placed into a changing environment, and by studying their reactions to the changes in the environments, he developed eight general laws of the software evolution as shown the following Figure 6.

No.	Name	Statement
1	Continuing Change	An *E*-type system must be continually adapted, else it becomes progressively less satisfactory in use
2	Increasing Complexity	As an *E*-type system is changed its complexity increases and becomes more difficult to evolve unless work is done to maintain or reduce the complexity
3	Self Regulation	Global *E*-type system evolution is feedback regulated
4	Conservation of Organisational Stability	The work rate of an organisation evolving an *E*-type software system tends to be constant over the operational lifetime of that system or phases of that lifetime
5	Conservation of Familiarity	In general, the incremental growth (growth rate trend) of *E*-type systems is constrained by the need to maintain familiarity
6	Continuing Growth	The functional capability of *E*-type systems must be continually enhanced to maintain user satisfaction over system lifetime
7	Declining Quality	Unless rigorously adapted and evolved to take into account changes in the operational environment, the quality of an *E*-type system will appear to be declining
8	Feedback System	*E*-type evolution processes are multi-level, multi-loop, multi-agent feedback systems

Figure 6. Laws of Software Evolution, as proposed by Lehman (M. Lehman & Fernandez-Ramil, 2006)

These laws are derived from an empirical study of software evolution, and all of them are quite intuitive to understand, such as the *Continuing Change Law* that indicates an E-type program must be continually changed to adapt to the changing environment, or the *Increasing Complexity Law* that indicates an E-type program becomes more and more complex as the program grows older. The eight laws now are adopted as baseline by many researchers in this field, yet we shall point out that Lehman's study was all based on an objective view, which must be obeyed by the software engineers. For us, this view underestimated the importance of the role played by the software engineers, as the creators of software, it is their intentions that decide the properties of software, hence the interpretation of software becomes quite subjective for us, and we adopt it into the requirements engineering domain to study the nature of software, as we shall see in Chapter 4 and Chapter 5.

2.2.2 The Metaphor between Software Evolution and Biological Evolution

Another interesting branch of the study of software evolution tried to make a metaphor linking it to the biology evolution, such as what Godfrey and German did, they proposed a model of software evolution where source code serves as gene, and software functions are the result of explaining the gene, and a version of software is recognized as an individual (Godfrey & German, 2008).

Based on this model, they pointed out that software maintenance and software evolution are different concepts: software maintenance was a process that keeps software running without changing the gene (code) of the software, and software evolution meant adding essential changes to the software by changing the gene (code) of the software. Although this idea is interesting, yet we shall point out that a biology species is usually characterized by a pool of gene serials from multiple creature individuals but not only a gene serial from just one software individual.

Similarly as Godfrey and German, Mens attempts to convey the concept of 'software ecosystem'. For him, a software ecosystem is 'a collection of software projects developed and used by the same community' (Mens & Grosjean, 2015), and this definition was derived from two older works: 1) Messerschmitt and Szyperski's definition, 'a collection of software products that have some given degree of symbiotic relationship' (Messerschmitt & Szyperski, 2003); 2) Lungu's definition, as 'a collection of software projects which are developed and evolve together in the same environment' (Lungu, 2008). Whichever of the aforementioned definition is adopted, the general idea is the same: studying interactions between software and its environment. By comparing with biological ecosystems, new strategies can be developed to improve the effectiveness and resilience of software.

For Mens, there are two possible ways to make comparisons, depending on the roles software plays in a trophic web (food world), as producer or a consumer: 1) biological species ≈ software components, this is a technical view, interpreting all the software and hardware as components of a trophic web (e.g., application websites are content producers, and a search engine web site consumes these contents and indexes them as its own products for other possible consumers in a higher level); 2) biological species ≈ project contributors, this is a social view, classifying software engineers into different kinds according to their roles in a trophic like web (e.g., some engineers develop core library packages acting as producers, and some other engineers may develop applications based on such libraries acting as consumers).

Both Godfrey's and Mens' work are interesting, and indeed such kinds of comparisons might contribute to a better understanding of software practice. For example, to make a software company efficient and robust, the structure of the role arrangement should be carefully designed that the dependency between the producers and consumers should be balanced well, neither too many producers, nor too many consumers. Similar examples could be fund in the bionics field that bats' acoustic detection system was studied and used as a prototype of the modern radar systems (Bin-bin, Hai, Xiaoping, & Hesheng, 2012).

However, not all aspects of the two species, software and biology, can be compared to make a complete metaphor. To achieve a shared set of foundational understandings of software evolution, we still need to get back to its original nature. Following this view, many researchers proposed taxonomies and ontologies for software change, and we shall list out some of their main contributions as follows to delineate an outline of the study in this field.

2.2.3 Taxonomies and Ontologies for Software Change

Swanson and Lientz's Work

Researchers have been trying, for decades, to unify the concepts and terminologies of software maintenance and evolution. Swanson and Lientz are recognized as the pioneers who firstly provided an exclusive and exhaustive typology for software maintenance. They divided the maintenance activities of

application software into three categories: 1) corrective maintenance, consisting of the activities of fixing bugs; 2) adaptive maintenance, referring to the changes made to adapt to the new technical environment (e.g. operating system software, frameworks); 3) perfective maintenance, referring to the changes of eliminating processing inefficiencies, enhancing performance, improving maintainability and other enhancements on functions (Swanson, 1976), (Lientz & Swanson, 1980).

Although Swanson and Lientz's work has been widely accepted, later works revised the initial meanings of these concepts inconsistently sometimes, and even the standards of the IEEE (Institute of Electrical and Electronics Engineers) have been revised it in this way. In 1990, a new category called 'preventive maintenance' was added in the main body of the IEEE standard. The new term means 'maintenance performed for the purpose of preventing problems before they occur'. Later in 1998, it was removed from the main body of the standard and only mentioned in its appendix ('IEEE Standard Glossary of Software Engineering Terminology,' 1990), ('IEEE Standard for Software Maintenance,' 1998).

Chapin's Work

Following Swanson and Lientz's work, Chapin et al. proposes a classification from a different perspective, which is not based on peoples' intentions but on the objective observations of the differences before and after the changes occur (Chapin et al., 2001). This classification is composed of four main clusters and refined into 12 different types, including: 1) support interface cluster (types: training, consultive, evaluative); 2) documentation cluster (types: reformative, updative); 3) software properties cluster (types: groomative, preventive, performance, adaptive); 4) business rules cluster (types: reductive, corrective, enhancive).

Generally speaking, Chapin's work constitutes an extension of Swanson's work, refining the initial classification of software maintenance into a finer granularity. The whole work is based on an objective view, identifying different kinds of software changes according to the ascertainable evidences observed, and the process of deciding the type of a software change activity is quite intuitive and reliable for the software maintainers, including all practitioners, managers, and researchers. However, when we recognize software as an artifact, we cannot understand the true nature of software change without considering the intentions of its stakeholders. In this dissertation, we demonstrate this view in further detail in Chapter 4 and Chapter 5.

Buckley et al.'s Work

Buckley et al. proposed another taxonomy in 2005. Compared with previous work, this taxonomy focuses more on the technical characteristics rather than on the general concept of software evolution (Buckley et al., 2005). The life cycle of software is partitioned into three phases: compile-time, load-time and run-time. Furthermore, several other dimensions of software change are proposed which could be grouped into two main categories: 'characteristics of software change mechanisms and the factors that influence these mechanisms.'

Generally speaking, these authors view previous works as trying to answer why software changes occurred, and their work contributed in answering the how, when, what and where software changes occur. According to this view, they discuss software changing mechanisms, and the factors that influence

these mechanisms. This work was restricted more likely as a list of criteria to justify the qualities of different kinds of software change supporting tools, although this is intuitive and handy for practitioners, the research subject was shifted from the software change itself to its supporting tools. Instead of improvement, it was a supplementation of the previous works, and what we do is to provide foundations for all such kinds of research and practices.

Kitchenham, Ruiz and Anquetil's Work

Ontology constitutes a more rigorous method to explicitly represent the meanings of concepts than a taxonomy. It has been used widely to capture knowledge in many research areas, and several researchers have tried to provide ontologies for software maintenance.

Kitchenham is the researcher who firstly provided a carefully crafted ontology for software maintenance in 1999. For her, software development is different from software maintenance, as the later refers about the activities applied to later releases delivered after the software has been deployed. In her paper, she identifies several factors that influence software maintenance, and classifies them into four dimensions as shown in Figure 7, including product, peopleware, process organization, and maintenance activity types (Kitchenham et al., 1999). According to this classification, general questions could be answered, such as what is under maintenance (e.g., product with size, age, and etc.), who is maintaining the product (a software maintainer with some skills, attitudes, and etc.), under what kind of organization the product is maintained (e.g., a maintenance group with some resource, and technologies, and etc.), and finally what kind of maintenance activity is taken out (e.g., corrections, new requirements, and etc.). Instead of talking about the ontological nature of software maintenance, it is more like a project management oriented work for the team members of a maintenance group.

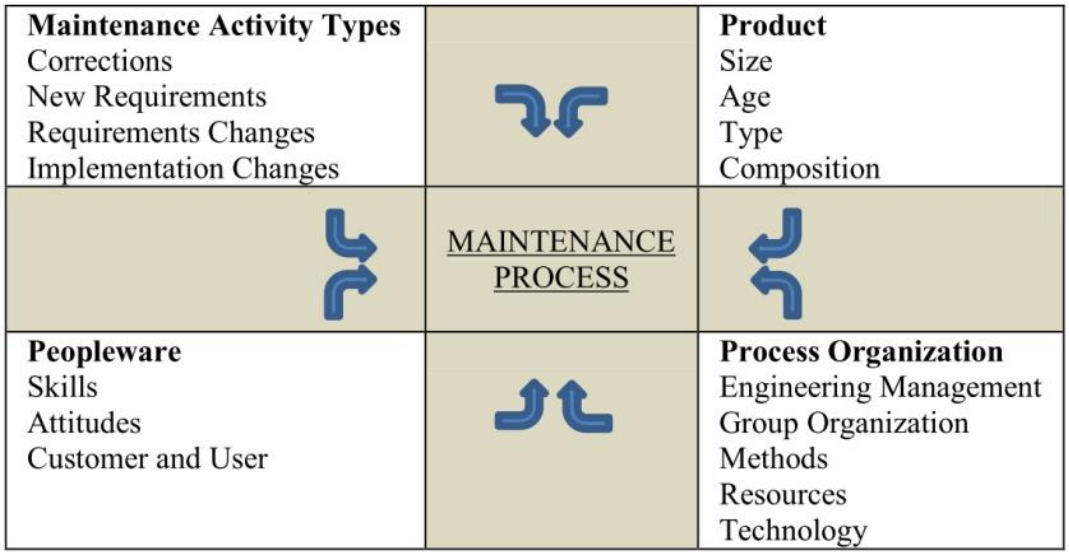


Figure 7. An overview of domain factors affecting software maintenance (Kitchenham et al., 1999)

Five years later, Kitchenham's work was refined and enlarged by Ruiz in 2004. This work extends the four dimensions of software maintenance proposed by Kitchenham into four sub-ontologies respectively: Products Sub-ontology, Agents Sub-ontology, Process Sub-ontology, and Activities Sub-ontology. Besides these refinements, additional Workflow Ontology and Measure Ontology are introduced in order

to support maintenance projects in organizations, as shown in Figure 8 (RUIZ, VIZCA ÍNO, PIATTINI, & GARC ÍA, 2004). These can be used to provide meta-level guidance for managing software maintenance projects.

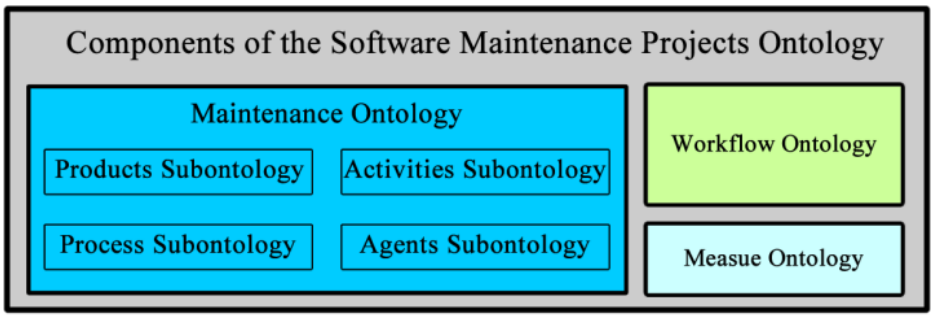


Figure 8. An overview structure of the ontology for software maintenance project (RUIZ et al., 2004)

Comparing with Kitchenham and Ruiz' work, Anquetil and et al. provided a similar ontology in the book *Ontologies in Software Engineering and Software Technology* (Anquetil, de Oliveira, de Sousa, & Batista Dias, 2007). They shared the same idea that after the initial development phase, software maintenance phase will follow and will last for a long period of time, and during which lots of changes will be adopted to better suit stakeholders' needs. Yet, Anquetil deals with these software maintenance activities from a knowledge management perspective. In other words, he emphasizes the importance to provide suitable knowledge according to a specific maintenance scenario for the proper software maintainers. As shown in Figure 9, a software system is implemented to solve some problems in an application domain, the knowledge about the application domain and the system itself will be elicited and stored in a knowledge base (KB). Additionally, the knowledge about the maintenance project will also be added into this KB. Consequently, according to such a KB, a modification process (task) to a software system will be assigned to the proper software maintainers who possess the required computer science skills.

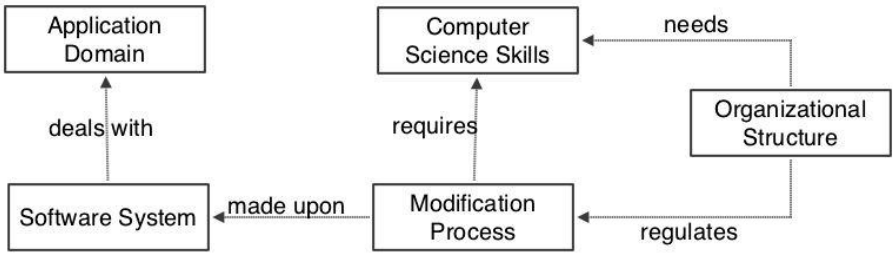


Figure 9: An overview of ontology for software maintenance (Anquetil et al., 2007)

Similarly to Buckley et al.'s work, Kitchenham, Ruiz and Anquetil's work was also a supplementation of the earlier work, because instead of explaining the nature of software and software evolution, they paid more attention to the issues about software project management, clarifying the roles of different kinds of stakeholders who took different kinds of software maintenance activities during the life span of a software project. In other words, their work was project oriented, and the most possible consumers of their work should be software managers instead of software practitioners.

Tappolet et al.'s Work

Similarly, Tappolet and his colleagues lunched the project OntEvo[8] trying to remedy the problems caused by software change. More specifically, they divided the sources of problems into two categories, including: 1) the internal source, referring to the difficulty to study or recall the meaning underlying a piece of source code without sufficient comments or documents; 2) and the external source, referring to the difficulty to manage the dependencies among so many libraries that are developed and maintained by different groups or organizations (Tappolet, Kiefer, & Bernstein, 2010).

For Tappolet at et al., both the two kinds of problem sources are derived from the lack of knowledge about the source code, hence they propose a set of ontologies, including a software, version and bug ontology. With these ontologies, on one side, software engineers can encode relevant knowledge into sharable files (or repositories); and on the other side, others could query the files (or repositories) to extract useful knowledge. As the knowledge is stored in sharable and query-able files, this solution provides help for both two kinds of problem sources in the process of software evolution. However, as this approach is source-code oriented, it completely ignores the software artifact perspective (e.g., the architectural components, behavioural models and requirements ones). To capture more semantics necessary, as demonstrated in Chapter 5, we propose different kinds of software artifacts at different abstraction layers.

2.2.4 Identifying and Recording Software Changes

The Version Control Systems (VCSs) are widely used in the industry to record the development history of software projects, and according to a survey published by Ohloh[9] concerning open source software projects, 70% of them use Concurrent Versions System (CVS) or Subversions, and over 25% of them use Git (Kleine, Hirschfeld, & Bracha, 2012). However, as we stated earlier, most of the literature work in understanding of software and software change have been limited to source code level. Consequently, the work in identifying and recording software changes was also limited to source code level.

For instance, all the VCS tools mentioned above were developed on a unique core mechanism, checking the syntactic difference between two versions of a file. In this context, a version of a file could be understood as a snapshot of the syntactic content of the file with a time stamp. Whenever a programmer commits a new version of a file, the syntactic difference between these two versions will be calculated, and recorded in a repository with some additional limited meta-data (e.g., commit user, commit time, and etc.) and some description of the commit (e.g., usually a few nature language sentences).

As aforementioned, all the information collected is source code oriented, this is because such tools are not initially developed to identify and record software changes, but to provide a cooperation platform for the project members (some time they are distributed geographically), committing their contributions to the project simultaneously, and meanwhile solving the possible conflicts they may encounter among each other. For example, if two programmers commit to the same version of a file, rewriting the same line of the source code differently, only one of the new commits could be used as the latest version, based on which further updates of the file could be made.

The data collected by such VCS tools could be used to do analysis about the evolution process of the software, yet as we stated, this is a subsidiary function we can get from them, and it is not practical to use

[8] A Software Evolution Ontology, https://files.ifi.uzh.ch/ddis/oldweb/ddis/research/evoont/index.html
[9] Ohloh, http://www.ohloh.net

them as a main means to study the evolution process of software, especially for the project managers without sufficient technical skills to navigate within these tools. To remedy this situation, some software change analysis tools were developed, such as (Fischer, Pinzger, & Gall, 2003) and (Ghezzi, Wursch, Giger, & Gall, 2012), who provide interfaces for users to query and navigate within a library of software changing history.

To enhance these VCS tools used as software change analysis tools, other technologies were proposed. One of the branches is to detect changes automatically. In other words, the programming activities of the programmers are monitored, analyzed, and then recorded by such 'change-aware' tools. Comparing with the history data provided by VCS tools, these new tools adopt the change of software as the center of a software developing process, and the collected data about the programmers' activities could be used as another source to analyze software change processes (Robbes & Lanza, 2008), (Wloka, Ryder, & Tip, 2009), (Omori & Maruyama, 2008).

Compared with automatic software change detection, the other branch of software change analysis tools tries to visualize the software change history, providing more intuitive ways to examine the way software evolves. For example, some of the these tools collect change activities on the software modules during a period of time, then create a static figure about the changing path of the modules (G îba & Ducasse, 2006), (D'Ambros & Lanza, 2009). Taking a further step, Beyer and his colleagues reused the records from CVS to create a dynamic visualization (e.g., an animation/movie of a change process) of software changing history (Beyer & Hassan, 2006).

Although many works attempted to identify and record software changes, they are generally limited to the source code level. As such, these tools can process semantic granularities at the file, class, method level, but not at a component or requirement level. This limitation in the semantic granularity causes a lot of knowledge to go unrecorded and therefore missed during a software change process. This missing knowledge constitutes one of the main reasons of the high cost of software change.

Parnas shares the same idea, and states that the primary cause of a poor state-of-the-art in software engineering practice is the failure to produce good documentation to record the multi-faceted knowledge that comes with a software system (Parnas, 2011). In his paper *Software Aging*, several practical reasons related to documentation were proposed explaining why software maintenance is costly and may result in chaos. Besides that, he emphasizes that although documentation is an unpopular topic which is often neglected, systematic documentation might be essential to ameliorate the current situation (Parnas, 1994). But of course, to do proper documentation one needs to decide first what knowledge about a software should be identified and recorded.

Although little work has been done in capturing richer semantics during the process of identifying and recording software changes, it seems a promising research topic to provide foundational support for software change. Some attempts were made, such as Altmanninger's work, in which the changing granularity was extended to the conceptual models about the software instead of source code itself (Altmanninger, 2008); or the Semantic Versioning standard provided by OSGi[10], indicating the dependency status of a version of software (e.g., from v-1.2.3 to v-1.2.4, nothing happened at interface, no need to adjust the calls of the interface methods; from v-1.2.4 to v-1.3.0, some changes happened at the inter-

[10] OSGi, https://www.osgi.org/

face, however it is compatible with the old calls of the interface methods; from v-1.3.0 to v-2.0.0, essential changes happened at the interface, and the old calls of the interface methods won't work anymore).

Different from all the related work stated above, we start our research with studying the ontological nature of software, based on which we examine the essential properties of different kinds of software artifacts, according to the fact that each of these software artifacts (software program, software system, software product, and licensed software product) is constantly dependent on a different intentional entity. Each of these intentional entities refers to a kind of expected behaviors involving different parts of a complex socio-technical system (namely, the inside machine, the interface, and the outside world), which in turn emerges from the interaction between a software-driven machine and a social environment.

In other words, we extend the interpretation of software from the perspective of source code to the perspective of socio-technical system. By recognizing the changes in these different kinds of software artifacts as different kinds of software changes accordingly, the task of identifying and recording different kinds of software changes is reduced to the task of identifying and recording the changes in the different kinds of software artifacts. Meanwhile, this approach will help to clarify some terminology ambiguities. For example, we may define the following kinds of software changes: 1) *refactoring* refers to the creation of new codes, keeping the identity of the software program; 2) *re-engineering* refers to the creation of new software programs, keeping the identity of the software system; 3) *software evolution* refers to the creation of new software systems, keeping the identity of the software product.

These changes in different software artifacts happen at different abstraction levels within the socio-technical system, providing help in understanding the software and software change for different kinds of stakeholders who play different roles in a software project (e.g., a company manager may focus on the software evolution of software products in the social environment, a project manager may focus on the software re-engineering of software systems at the interface, and a programmer may focus on the software refactoring of software programs inside a software-driven machine).

2.3 Understanding Assumptions

As mentioned in the introduction chapter, in addition to the essential properties about the different kinds of software artifacts, there is another kind of knowledge that deserves special attention, and it is the assumptions made during a software engineering process. The requirements for most software applications -- the intended states-of-affairs these applications are supposed to bring about -- concern their operational environment, usually a social world. But these applications don't have any direct means to change that environment in order to bring about the intended states-of-affairs. In what sense then can we say that such applications fulfill their requirements? One of the main contributions of this dissertation is to account for this paradox. We do so by proposing a preliminary Ontology of Assumptions. Before diving into the details of this ontology characterizing and making explicit a number of notions that are used implicitly in software engineering practice, we illustrate several similar related works as follows.

2.3.1 Interpretations from Linguistic and Cognitive Science Perspectives

'Assumption' is a severely overloaded term used in many communities (e.g., research, industry, and etc.) as well as in our daily lives. The interpretations of this term diverge significantly in different contexts. Nkwake authored a chapter named 'What are Assumptions?' in the book 'Working with Assump-

tions in International Development Program Evaluation', in which he discusses the nature of assumptions, and grouped assumptions into several categories, including: Ontological Assumptions, Epistemological Assumptions, Axiological Assumptions, Cultural Assumptions, Idiosyncratic Assumptions, Legal Presumptions, Metaphoric Assumptions, Intellectual Assumptions, and Causal Assumptions. (Nkwake, 2013) Among so many ways to interpret assumptions, in the sequel, we present a few examples of these possible interpretations, relying on the language use of the term (Ennis, 1982):

Conclusion: e.g. Tom said: 'my assumption is that you are going out, since you are wearing your cap.' The conclusion of 'going out' is derived from the current situation 'wearing your cap'.

Less-than-fully established proposition, in an accusation sense: e.g. Mike answers: 'that is only your assumption, you don't know it.' Mike's reply suggests that it only looks like he's going out, and that was only Tom's guess, with no guarantee that it holds.

Adopted in order to deceive, fictitious, pretended: e.g. 'although bad things did happen, please assume that they never happened.' The term assumption is interpreted as a kind of 'self-deception' here that 'you can deceive yourself that nothing bad happened'.

Another dimension of the work tries to interpret assumptions from the perspective of Cognitive Science. More specifically, many researchers try to explain the meaning of an assumption as a proposition that is created from a particular kind of mental state. In the literature, mental states are usually classified into three categories, including Belief (B), Desire (D), Intention (I), and this understanding of mental states is referred as the BDI model. A proposition in a belief is the knowledge of an agent about the world, a proposition in a desire represents the states an agent wants to reach (in a derived sense), and a proposition in an intention represents a desire content that an agent is committed to achieve (Ferrario & Oltramari, 2005).

According to this BDI model, Jureta and et al. systematically analyzed the role of assumptions in requirements engineering, which is reported in (Jureta, Mylopoulos, & Faulkner, 2009). Three basic concepts in requirements engineering are matched to those three kinds of mental states respectively: 1) an assumption is matched to a believed proposition; 2) a requirement is matched to a desired proposition; 3) and a task is matched to an intended proposition.

Although it is a promising research direction to address such a detailed ontological analysis of the nature of assumptions as well as of their relations to cognitive and social agents, we prefer to leave it as another contribution in the future work, and in this dissertation we focus on explaining the bridging role of assumptions in software engineering, solving the paradox aforementioned. In other words, when we refer to the propositional contents of assumptions, we make them neutral with the discussions about mental states. We simply say that the assumptions are composed of propositions, yet we don't discuss which kind of propositions is concerned in this dissertation. We believe this topic deserves another specialized paper, and we are working on it in parallel.

2.3.2 van Lamsweerde's Interpretation of Assumptions in Requirements Engineering

van Lamsweerde interprets assumptions within the requirements engineering framework proposed by (Michael Jackson & Zave, 1995), in which the process of requirements engineering is composed of the

following procedures: 1) anchoring the machine in the problem world; 2) characterizing the problem world; 3) delimiting and structuring the problem world; 4) chaining satisfaction arguments; and 5) deriving specifications from requirements. For him, domain assumptions characterize partial properties of the problem world, hence a set of domain assumptions should be elicited and expressed as a set of statements about the problem world (van Lamsweerde, 2009).

For Jackson and van Lamsweerde, the basic elements used to characterize the problem world are statements about the world. A statement could be understood as a piece of expression in some language, and usually statements could be classified into three categories as shown in Figure 10: 1) a prescriptive statement states desirable properties of the world in the optative mood (an agent wants to do something); 2) a descriptive statement states properties about the world in the indicative mood (an agent possesses some understanding about the world); 3) and a definition is a statement assigning the precise meanings to the terms used in the problem world without mood.

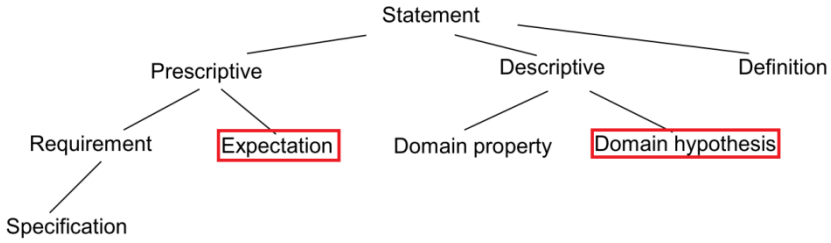


Figure 10. Further distinction among statements in the problem world (vanLamsweerde, 2009)

As shown in Figure 10, the two kinds of statements rounded by red rectangles represent two kinds of domain assumptions proposed by van Lamsweerde. An *expectation* is a prescriptive statement, it prescribes a specific behavior of the problem world that the machine cannot reach (e.g., a passenger will press the buttons when he/she is in an elevator). A *domain hypothesis* is a descriptive statement about the problem world, this hypothesis is an estimation of a behavior of the problem world (e.g., the possible temperature of the room is always between 10 degree to 20 degree). Hence, a domain hypothesis is not expected to hold invariably, unlike descriptive the *domain properties* resulting from natural laws.

According to such statement based characterization of the problem world, and the other procedures within a requirements engineering process, van Lamsweerde made an extension of Jackson and Zave's original formula $A, S \vDash R$ (A: Assumptions, S: specification, R: Requirements), and refined it into: $\{Specification, Assumption, Domain\ Property\} \vDash Requirement$. According to this new formula, a requirement engineer should ensure that the requirements will be satisfied whenever the specification is met, provided the domain assumptions and domain properties hold.

However, van Lamsweerde's discussion focuses only on world assumptions. In this dissertation, instead, we extend it into machine assumptions, and propose two additional kinds of dependence assumptions which, as we have mentioned, are essential for explaining how software can affect the social world.

2.3.3 Lewis's Assumption Management System

Lewis states that 'assumptions are made concerning how the software will be used, …, what environments it will operate in', as well as 'the incompatibilities between the assumptions and the assumed operation environment will cause failures' (Lewis et al., 2004).

Based upon this understanding, she also emphasizes the importance of making assumptions explicitly represented as we do in this paper. To achieve this goal, she proposes a system called *Assumption Management System* (AMS) to insert and extract assumptions in and from the source code. This system is supported by the functions of storing the extracted assumptions in a repository, querying the repository, and making management decisions based on the assumptions recorded in the repository.

```
/*-
    <assumption>
        <type>
            Assumption type.
        </type>
        <description>
            Assumption description.
        </description>
    </assumption>
*/
```

Figure 11. Syntax structure in Lewis's assumption management system

The syntax structure of an assumption assertion is adopted here from Lewis's work as shown in Figure 11. Simply put, assumptions are recorded as comments to the source code, and they are encoded in XML. As shown in Figure 11, and following the usual XML convention, the pair of '/* ... */' indicates the comment area, the pair of labels '<*assumption*>' indicates the assumption area, the pair of labels '<*type*>' indicates the type of the assumption, and the pair of labels '<*description*>' indicates the natural language description of the assumption.

The idea underlying this work is that assumptions are recorded by software engineers while they are writing source code. When the source code is ready, a XML parser can be used to extract the assumptions and store them into a repository in a structured way for future queries. This is useful for sharing purposes with all members of a software project, reducing the chance of misunderstanding, and helping to ensure global consistency of the system.

Although, as stated by Lewis, recording and parsing of the assumptions as contents of comments in source code has been proved very useful in the coding process in a software engineering project, this is too late a stage for uncovering possible inconsistencies in the project. This is noted by Lewis who states that 'to address interoperability requirements, the use of assumptions management would have to be moved to other activities and artifacts of software development, such as requirements analysis, architecture, and design'.

The problem pointed out by Lewis is essential to software engineering projects, as the evidence has proved that the errors in requirements, such as misinterpreting or neglecting some implicit assumptions, are much more expensive to fix at a later stage than in the early stages of a software project (Nuseibeh &

Easterbrook, 2000). What we do in this dissertation is to capture and analyze the assumptions in the requirements engineering stage of the process, i.e., in the earliest stages of the software engineering process.

Instead of dealing with source code, we capture and record assumptions in the process of deriving specifications from the requirements. In this stage, requirements are decomposed or refined into specifications including an external specification and an internal specification. Moreover, we provide here a more refined categorization of assumptions that should be discovered by software engineers in the requirements engineering stage. Furthermore, and more importantly, we illustrate the key role that such assumptions play in linking the world and machine states together. To illustrate these points, in the Chapter 6, we present a case study of a meeting scheduler system that demonstrates how and what kinds of assumptions can be captured in a requirements engineering process.

Chapter 3

Baseline

3.1 DOLCE Adopted as the Foundational Ontology

Along with the vigorous development in the field of Semantic Web proposed by (Berners-Lee, Hendler, & Lassila, 2001), ontology also has been pervasively adopted in many science fields as a means of knowledge repository, especially in the fields in which huge amount of information should be classified and maintained. For example, in the biology field, Gene Ontology Consortium aims at producing ontologies covering a set of dynamic and controlled vocabulary as shared knowledge of the roles of gene and protein in cells (Ashburner et al., 2000); and in the astronomy field, NASA[11] launched a project called Semantic Web for Earth and Environment (SWEET), in which a collection of ontologies are developed, including many basic concepts, such as space, time, Earth realms, physical quantities, and etc. (Raskin & Pan, 2005).

Although the term 'ontology' is widely used in many fields, the meaning of this term itself was not clear enough till recent years. Thanks to the work of many ontologists clarifying this terms for many years, especially a series of works made by Nicola Guarino, a precise definition is emerged and accepted by many researchers in the ontology field (Guarino & Giaretta, 1995), (Guarino, 1995), (Guarino, 1998), (Guarino, 2009). Among his works, a paper titled 'What Is an Ontology?' is published in 2009, in which the meaning of the 'ontology' was thoroughly discussed (Guarino, Oberle, & Staab, 2009).

For him, the term 'ontology' could be interpreted in two senses: 1) Ontology[12] is a philosophical discipline, a research filed similarly as Physics, Chemistry, Biology, and etc. More precisely Ontology is a research discipline studies the nature and structure of objects, as Aristotle defined this term 'Ontology' in his Metaphysics as the science of 'being qua being'. By given a domain, Ontology discusses about the entities and relations existing in it; 2) an ontology[13] is an special kind of computational artifact, as it is created by people with the purpose to represent the understanding of a given domain. In this sense of interpretation, an ontology is a product within the Ontology research discipline. In other words, by studying the domain, people can get some knowledge about it, and by representing such knowledge in some form of language expressions, they create an ontology about the domain as a result.

The prevalent use of this term 'ontology' in Computer Science should refer to the interpretation in the second sense. Intuitively, a conceptual model about the domain in concern developed during a software engineering process could be understood as an ontology of this kind, according to the definition of conceptual modelling proposed by (Mylopoulos, 1992) that 'conceptual modelling is the activity of formally describing some aspects of the physical and social world around us for purposes of understanding and communication'.

[11] National Aeronautics and Space Administration (in United States of America)
[12] Guarino suggested using the capitalized term to refer to this research discipline.
[13] Accordingly, Guarino suggested using the lowercased term to refer to a product in this research discipline.

This interpretation of ontology from the perspective of conceptualization was firstly proposed by Gruber in 1993 that an ontology is an 'explicit specification of a conceptualization' (Gruber, 1993). Later on, it was extended into a new expression that an ontology is a 'formal specification of a shared conceptualization' proposed by (Borst, 1997), and again this definition was refined by Stuber et al. that 'an ontology is a formal, explicit specification of a shared conceptualization' (Studer, Benjamins, & Fensel, 1998). Guarino adopted Studer's view, and provided a detailed account of the notions of 'conceptualization' and 'specification', and emphasized the importance of 'sharability' of an ontology.

We adopted Guarino's interpretation of ontology as a component of our baseline in this dissertation, and based on that we will discuss about the ontological nature of software, and furthermore provide an ontology of software which could be used as a foundation for identifying and recording changes in different kinds of software artifacts. As an ontology of software, we share similar idea with Irmak, it should be in accordance with the common beliefs and practices held by different kinds of stakeholders who share the related concepts. It must 'be coherent with the way people talk about them, with the things they believe about them, with their practices that involve those objects' (Irmak, 2013).

To clarify the point here, we are not trying to discover the unique true nature of software, like what the physicists do in looking for the true nature of the physical universe. However, we shall say that to question the meanings of software is different from questioning the physical laws of the universe. More likely, it is a problem from the linguistic and cognitive point of view. People in the community usually interpret the terms differently and ambiguously, and what an ontologist should do is to provide definitions and explanations of the terms based on some *widely shared primitive concepts*, then the newly proposed interpretations of the terms could be widely shared and used in communications. After all, we believe an ontologist is different from a physicist.

To develop an ontology of software, we start with looking for a suitable set of *widely shared primitive concepts* for our purpose, and as a result the ontology 'Descriptive Ontology for Linguistic and Cognitive Engineering (DOLCE) (Masolo, Borgo, Gangemi, Guarino, & Oltramari, 2003a)' is chosen as the foundational ontology for our work. On one hand, as a top-level ontology (Guarino, 1997), DOLCE provides preliminary concepts which are well-delimited with clear philosophical analysis. Hence, referring to these finely restricted concepts in DOLCE, our domain ontology could inherit these well-formed ontological commitments, and this makes our proposals clearer and easier to discuss in the community (Jureta et al., 2009).

On the other hand, DOLCE has a clear cognitive bias, for that it is intended to capture the ontological categories that underlie natural language and human commonsense (Masolo et al., 2003a). This makes it more compliant with people's intuition, and the concepts could be easier to be understood and accepted by stakeholders. For these reasons mentioned above, we choose DOLCE as our foundational ontology, and we present a brief introduction of some main concepts in this ontology, and after the introduction they will be reused as preliminary concepts many times in the later contents of this dissertation. Note that, this introduction is presented in an intuitive way, and for the readers who are interested in the details of DOLCE, please refer to the reports D-17 (Masolo, Borgo, Gangemi, Guarino, & Oltramari, 2003b) and D-18 (Masolo et al., 2003a) published in the WonderWeb project[14].

[14] WonderWeb, http://www.istc.cnr.it/project/wonderweb-ontology-infrastructure-semantic-web

As shown in Figure 12, the most general concept in DOLCE is called '*particular*', and a particular could be informally understood as something has no instance. Oppositely, a '*universal*' is something that does have some instances, yet to control the scale of the ontology, only particulars are adopted as the elements in DOLCE. Generally speaking, a particular is in the specific individual level, and a universal is in the abstract class or type level. A particular could be a physical thing, like my cat which is an existing creature in the world; also it could be a non-physical thing, such as stories, laws, and etc.

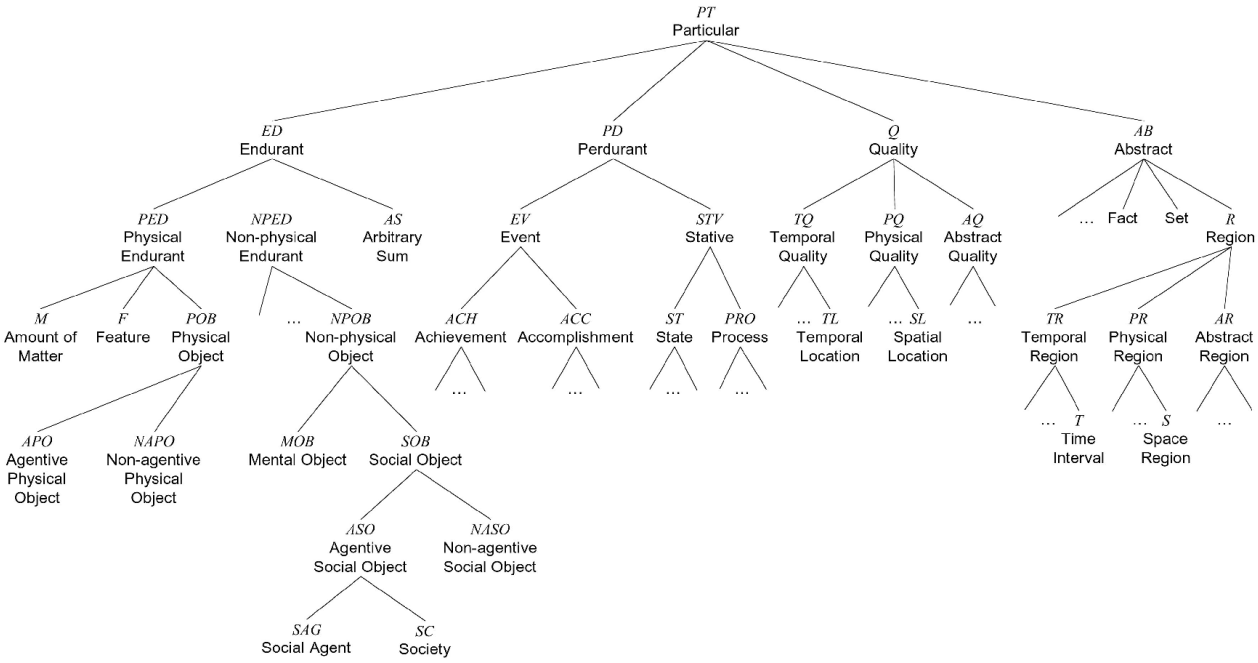


Figure 12. A diagram of DOLCE (Masolo et al., 2003a)

Under the concept particular, there are four specialized sub-concepts of it, including '*endurant*', '*perdurant*', '*quality*' and '*abstract*'. An endurant is a particular that presents its whole at a time point, and as time passing by, it could keep its identity. For example, my cat in last year and the cat in this year, they are the same cat growing up, and in this case it is a 'physical endurant'; a law or an economic system is 'non-physical endurant'; and putting my hands and my shoes together makes an 'arbitrary sum'.

A perdurant, on the other hand, is a particular that presents its whole among a time period. Its identity is associated with this specific time period. For example, a party has its starting part, duration part and the ending part. In each of these parts, the party only shows itself partially. Usually in our daily lives, we use these two concepts (endurant and perdurant) together: an endurant could participate in a perdurant; and meanwhile a perdurant shows itself through some endurants. For example, a 'group of people' is an endurant, a 'party' is a perdurant, then a group of people could participate in a party, and meanwhile this party exists through this group of people.

Quality and abstract form another pair of basic concepts in DOLCE, generally speaking a quality could be understood as an entity that we can perceive or measure, like the 'dimension' referred in (Gruber, 1995). For example, the color of a flower and the height of my body are both qualities. On the other hand, an abstract may provide a value region for a quality. For example, the value of the color could be red, yellow, blue, or any other color which is in this defined color region.

3.2 Artifacts

As we aim at providing an ontology of software, hence the main question we need to answer is 'what is a software'. The term 'software' is usually interpreted as a preliminary concept to refer all the non-physical parts of a computer system by computer science communities. However, as aforementioned, this general and vague understanding of software might cause many ambiguities, and our ontology of software is proposed to remedy this situation. Intuitively, software may be interpreted as a tool. Comparing with a hammer, software is a tool of a different kind which processes different functions. However, this brings about another question that 'what is a tool then'. To answer this question, we borrow the concept of *artefactual object*[15] introduced by (Guarino, 2014). As shown in Figure 13, software might be placed under the node 'Artefactual object', meaning that if a particular[16] is a software then it is also a artefactual object, as shown in the formula '$\forall x(Software(x) \rightarrow Artefactural\ object(x))$'.

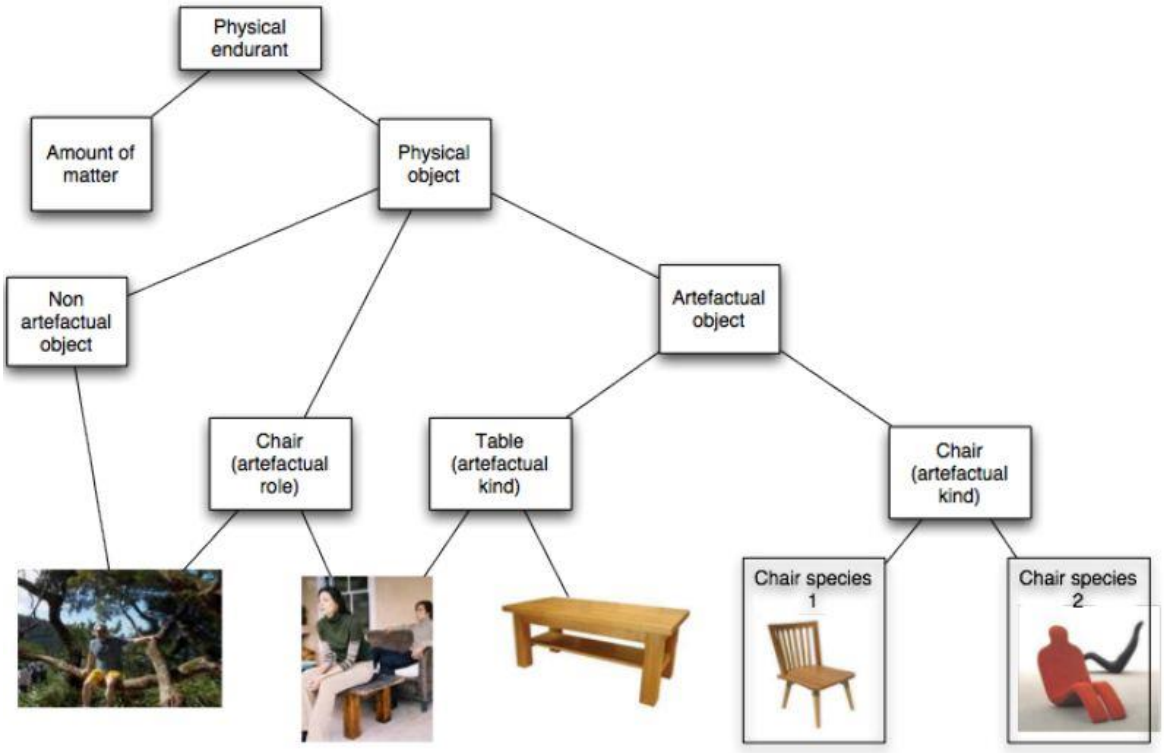


Figure 13. Artefactual objects, artefactual kinds and artefactual roles (Guarino, 2014)

According to the statement from Guarino, an artefactual object is a physical object which exists according to some design specification. In other words, an artefactual object exists only if some rational design choices have been made. Taking an example of a stone which is collected from a river, it may be used as a paperweight, or be used as a hammer, or be used as a piece of material to build a wall, and etc. However, the stone itself is not an artefactual object, because there is no design for the stone that it is an ordinary nature physical object. On the other hand, a paperweight or a hammer could be recognized as an artefactual object because there exist some design specifications for them. Taking the example of a ham-

[15] In this dissertation, we treat term 'artifact' the same as 'artefact', and 'artificial' the same as 'artefactual'.
[16] As Guarino's ontology of artifact is based on DOLCE, the basic elements in the ontology are all particulars.

mer as an artefactual object, a hammer should contain a hard head, and a handle connecting to the hard head.

On the other hand, we also need to make an explicit distinction between the concepts of artefactual kind and artefactual role. An artefactual kind should be rigid, and an artefactual role should be anti-rigid according to our understanding[17]. As illustrated in Figure 13, a trunk could be used as a chair when necessary, and it plays the role of a chair, but it is not an artefactual object because there is no design for it, and furthermore it could stop playing the role of a chair. In this case, the concept of chair is an anti-rigid artefactual role. A chair made by a furniture company is always a chair, because it is built according to an explicit design. In this case, the concept of chair is an artefactual kind which is rigid. In this dissertation, we consider the concept of software as an artefactual kind which is rigid.

Besides the statements aforementioned, Guarino adopted the so-call 'multiplicative approach' from DOLCE while interpreting the concept of artefact. By taking this view, we allow different entities co-locate in the same space-time, and we can ascribe to them incompatible essential properties respectively. For example, there is a classical discussion about the relationship between a status and the amount of clay. The status cannot survive a radical change in its shape, yet the amount of the clay keeps the same according to whatever shape changes. Hence, the status and the amount of clay must be different things, meanwhile co-located in space time.

The relationship between them is called 'constitution', and we can say that 'the status is constituted by the amount of clay, but it is different from the clay'. Through the work on such amount of clay by a status master, a set of essential properties (e.g., shape, or topology) could be assigned to it, and then a new 'status' is created. Similar situation occurs to the relationship between an artifact chair and the wooden material it made of, namely the artifact chair is constituted by such amount of wood material. This 'multiplicative' interpretation of an artifact is also reflected by the common use of natural language that the newly created entity is a genuine different thing. For instance, we can say a status has a head, two hands and etc., yet we rarely say the amount of clay has such a head, two hands, and etc.

As we shall see, we take the similar idea in this dissertation. For example, we interpreted a software program as a software artifact, and it is constituted by a piece of syntactic code. Generally speaking, we treat software and code as different entities, as they possess several different essential properties. The code could be occasionally created by a monkey by typing on a keyboard without any intention, the only thing matters is the syntactic structure which could be accepted by a machine, and that's all; yet, a software program must be created by human intentionally when they are writing code. As the detail discussion will show that this distinction is crucial in clarifying the ambiguities in the computer science community, and we adopt this view as part of our baseline.

Only one thing to notify that Guarino restricted his interpretation of artifact as physical object, although he also referred the possibility of interpreting an artifact as a social and abstract entity, he left it as an open question. Here in this dissertation, we follow Irmak's view that software should be interpreted as a social and abstract artifact, and more precisely we interpret software as an information artifact, and the detail discussion could be found in the contribution part of this dissertation.

[17] For the readers who are interested in the details of rigidity, please refer to (Guarino & Welty, 2009)

3.3 World and Machine Framework

As aforementioned that we interpret software as an artifact, and the stakeholders' intentions underlying a software artifact should be explicitly captured by us. We believe that such intentions play the key roles in identifying software from the things that are not software, and distinguishing different kinds of software artifacts from each other. For example, we can distinguish a software program from a piece of code that a program possesses an intention, and a piece of code doesn't possess any intention. Therefore, it seems natural to us to take a requirements engineering perspective while analyzing the essence of software, instead of focusing on computational aspects only.

According to the statement above, we shall base our further analysis on a revisitation of the seminal works by Jackson and Zave on the foundations of requirements engineering (Michael Jackson & Zave, 1995), (Zave & Jackson, 1997), (Gunter et al., 2000) which clearly distinguish the *external environment* (where the software requirements are typically defined), the *system-to-be* (a software-driven machine intended to fulfill such requirements), and the interface between the two.

Initially, the formula '$A, S \vDash R$'[18] was proposed by Jackson and Zave in 1995 for obtaining a specification from a set of requirements which makes explicit use of environment assumptions. In this formula, in which requirements, specification, and assumptions are represented by R, S, and A respectively, for a given R and A, a specification S needs to be provided such that '$A, S \vDash R$'. In other words, the satisfaction of the requirements can be entailed from the satisfaction of the specification together with the assumptions.

As we can see that the scope of the formula '$A, S \vDash R$' is restricted to the external environment part and the interface part, the situation inside the machine is not mentioned in this formula. To extend the formula into a more general scale, also covering the inside part of the software-driven machine, a reference model for requirements and specifications was proposed by Jackson and et al. in 2000. As we try to adopt it as a part of our baseline, a brief introduction of their original ideas will be briefly discussed as follows, meanwhile accompanied by our understandings of this framework.

Phenomena: individual/state/event

The fundamental concepts (e.g. phenomena, individual, state and event) which are used by Jackson and et al. are not explained, but adopted as preliminary concepts. In other words, they do not distinguish the *individual*, *state* and *event*, by calling all of them *phenomena*. They simply assume everyone knows and shares the same meanings of them, and based on this assumption, a name (e.g. predicate) should be created and designated to a phenomenon. The explanation of the designation process is recommended (e.g. natural language descriptions) by them, or ambiguities might be derived from these unexplained names.

By taking this strategy, a solution to create a WRSPM framework (W: Domain Knowledge, R: Requirement, S: Specification, P: Program, M: Programming Platform), as shown in Figure 14, could be proposed through a pure logical expression. Although natural language is also an alternative solution, to take the advantage of automatic reasoning provided by computers, and to achieve better rationale, the log-

[18] This formula was initially written as '$S, E \vdash R$', here we adjust the entailment symbol and the abbreviation of assumptions according to our interpretation based on their later work.

ical expression (e.g. propositional logic, first order logic or description logic) is preferred by them. For example, each of the phenomena could be designated with a predicate; then, logic expressions could be composed from these predicates, which become the desired WRSPM artifacts as stated by the authors.

Controllable/Visible

Again, the meanings of *controllable* and *visible* are not explained in their papers, and the authors assume everyone shares the same meanings of them. Then, the task to decide whether a phenomenon is controllable or visible by/to the environment or the machine is assigned to the stakeholders who will design, build and use the machine.

Environment and Machine

Environment and *machine* could be understood as two inter-related systems, environment system and machine system. This is another basic idea proposed by the authors that usually a new machine is embedded into the original environment, and then a new solution emerges from such embedment. For example, at one time, there was an environment system *es1* in which there was no coffee machine, and we make coffee manually from coffee bean. After several days, we thought a coffee machine might save our life. We bought a coffee machine *ms* and embedded it into the original environment system *es1*, and now we have a new environment system *es2* which has a coffee machine *ms* embedded as one of its subsystems, and finally we can get our coffee easier than before.

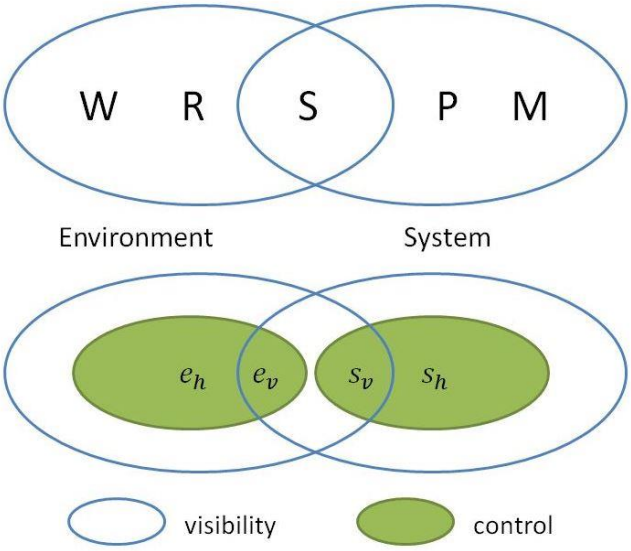


Figure 14. Reference model for requirements engineering (Gunter et al., 2000)

In the top part of Figure 14, each point in the ovals represents a statement. A simple statement example could be '*when a coin is inserted, and then a cup of coffee is provided*', and its alternative first order expression could be $\forall i.\,(CoinInserted(i) \rightarrow \exists c.\,CoffeeProvided(c))$.

In this example, the predicates *CoinInserted* and *CoffeeProvided* are used as the names to represent the phenomena of '*a coin is inserted*' *and 'a cup of coffee is provided'* respectively. By linking them with logic constructors, the logic expressions like the example could be developed and used as elements of

WRSPM framework. In other words, each of W, R, S, P, and M represents a set of statements created from the names which are designated to the phenomena.

In the process of developing a WRSPM framework, the statements created from the names are not randomly classified as elements of them. Firstly, the names should be assigned a label from e_h, e_v, s_v or s_h; then, for a statement belongs to any one of W, R, S, P, and M, only specific labeled names could be used as its components. For example, for a statement belongs to S, the allowed component names are limited to the ones labeled with e_v or s_v.

The meanings of the labels (e_h, e_v, s_v and s_h) are illustrated in the bottom part of Figure 14, any point in the ovals here represents a phenomenon (designated with a name) which is visible to environment and/or machine, and controllable to environment and/or machine. The bottom part of Figure 14 is further explained in Figure 15 that any one of e_h, e_v, s_v or s_h represents a set of phenomena (designated with names) which have different visibility and controllability according to environment and machine. In Figure 15, the composing rules of the statements in a WRSPM framework are also introduced, which could be summarized in the list followed.

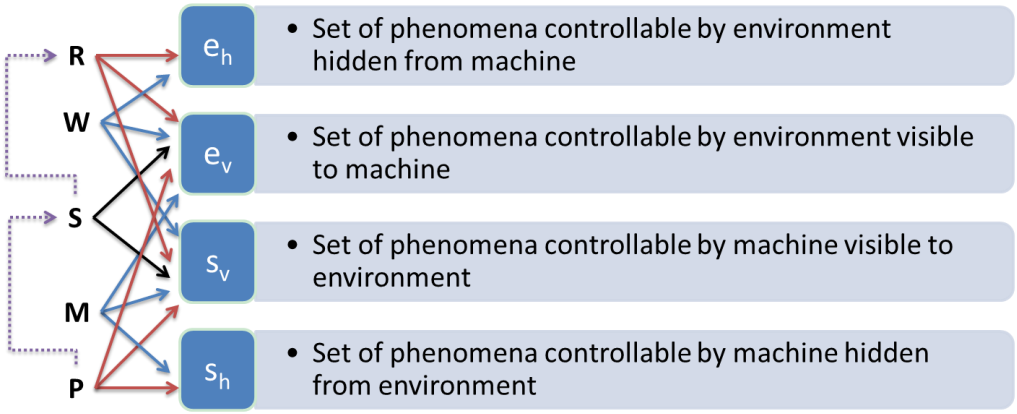


Figure 15. RWSMP framework composition rules

- Statements in W: uses names from e_h, e_v, s_v
- Statements in R: uses names from e_h, e_v, s_v
- Statements in S: uses names from e_v, s_v
- Statements in P: uses names from e_v, s_v, s_h
- Statements in M: uses names from e_v, s_v, s_h

Besides the composition rules shown above, the statements created should be kept in consistency as the formulas $P \wedge M \vDash S, S \wedge W \vDash R$ indicate: *If (i) S properly takes W into account in saying what is needed to obtain R, and (ii) P is an implementation of S underlying M, then (iii) P implements R as desired.*

Till now, we introduced the original diagram proposed by Jackson et al., by admitting most of their views we made several modifications to the diagram, and the detail descriptions and revisions of this model could be found in the corresponding contribution parts (e.g., Chapter 5, and Chapter 6), and here is a brief idea of our understanding and usage of this model. As shown Figure 16, in a software engineering scenario, the phenomena of interest could be classified into three categories according to their controllability and visibility, namely:

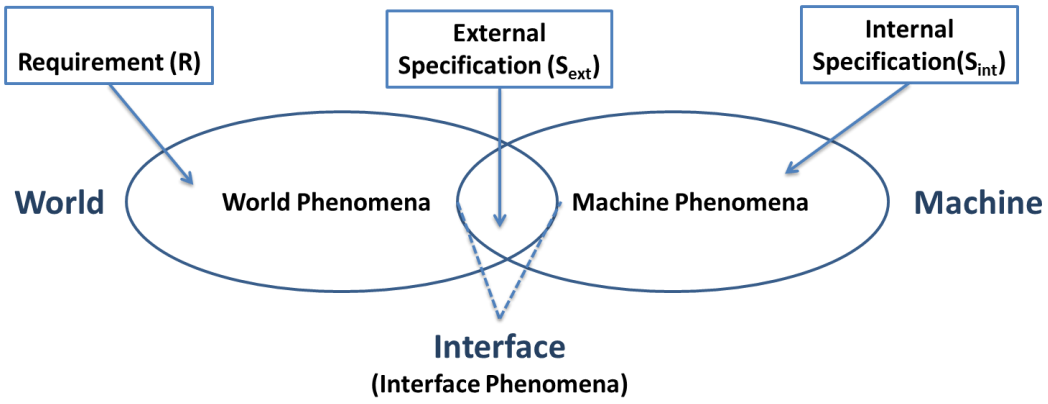


Figure 16. World and Machine Framework (WM Framework)

World Phenomena, which can only be seen and controlled by the world; *Interface Phenomena*, which can be seen both by the world and the machine, and can be controlled either by the world or by the machine; *Machine Phenomena*, which can only be seen and controlled by the machine. In (Wang, Guarino, Guizzardi, & Mylopoulos, 2014b) and (Wang, Guarino, Guizzardi, & Mylopoulos, 2014a), we proposed the term 'internal specification' to refer to a specification that only constrains the phenomena happening inside the machine; we distinguish it from the 'external specification' (originally called simply 'specification'), which constrains the phenomena happening at the interface; and a requirement only refers to phenomena in the outside world, excluding the phenomena at the interface; for the domain knowledge or assumption, we shall ignore it here, and then use a separating chapter (Chapter 6) to discuss about it, according to the essential role played by them in software engineering processes.

3.4 Situation Calculus

In this dissertation, we show the possibility of recording the requirements, external specification and internal specification as the essential properties of different kinds of software artifacts, and furthermore recording different kinds of assumptions and demonstrating the important role played by then in a software engineering process. To achieve that, we choose *Situation Calculus* as our representation language. Situation Calculus is a logic formalism proposed by McCarthy in 1993 to represent and reason about dynamic domains (McCarthy & Laboratory, 1963). It is chosen here because it is a well-known language, used in artificial intelligence to capture changes in states. This makes it quite suitable to demonstrate the link between state changes in the world and state changes in a software-driven machine.

We highlight however that, while the formalization in situation calculus may be useful here to better understand the nature of software, such formalization is not mandatory in the practice of software design, unless you plan to reason on the represented elements (e.g., to check somehow whether the aforementioned formula $A \wedge S \models R$ holds during a software developing or evolution process). In contrast, what is

important in software engineering process is to: 1) on one hand, record richer semantics in a software project for different kinds of stakeholders; 2) on the other hand, make the assumptions explicit being aware of their different kinds (we shall isolate four of them), so that the people who maintain or evolve the system can understand the role they play in the proper system's functioning. A brief introduction to Situation Calculus is presented as follows.

A *situation* (*s*), intuitively, is the complete state of affairs at some instant of time. In other words, it is a snapshot of the world described by a certain configuration of properties, and a property that takes a truth-value in a situation is called a *fluent*. A fluent (e.g., $meeting\_scheduled_W(Mtg, S_n)$) is usually represented by a predicate (e.g., $meeting\_scheduled_W$) having a situation (e.g., S_n) as its final argument.

An *action* (*a*) is the only cause of a transition between two situations, resulting in value changes in the fluents within them. Similarly to the notion of *function* in software engineering, an action may have a *pre-condition* described by 'precondition axioms' and a *post-condition* described by 'effect axioms'. The former indicates the situation in which the action (*a*) can possibly be executed; the later indicates the changes in values of fluents after the execution of the action in the new situation.

Figure 17. An example of a situation transition

The new situation achieved by executing an action (*a*) in a certain situation (*s*) is denoted by '*do(a, s)*'. Figure 17 shows what happens when an action is executed in a certain situation, resulting in a situation transition. In this example, in the initial situation S_0, the value of the fluent $meeting\_scheduled_W(Mtg)$ is *False*. By executing the action *schedule(Mtg)* in that situation, a new situation is achieved denoted by $do(schedule(Mtg), S_0)$. In this new situation, the value of the fluent $meeting\_scheduled_W(Mtg)$ becomes *True*, and this can be represented as $meeting\_scheduled_W(Mtg, do(schedule(Mtg), S_0))$.

These core concepts in situation calculus provide a foundation to describe the phenomena in the World and Machine Framework. As the phenomena in this framework are classified into world phenomena, interface phenomena, and machine phenomena, according to the boundary between the world and the embedded machine, we can manually assign labels (e.g. W for World, I for Interface, M for Machine) as subscripts to the fluents and actions according to their visibility and controllability to the world and the machine.

As shown in Figure 18, this assignment process results in a classification on the fluents (e.g. World Fluent, Interface Fluent, and Machine Fluent) and a classification on the actions (e.g. Interface Action, and Machine Action). Specifically, the interface phenomena *I* includes the fluents and actions that are observable both by the world (e.g., users of the machine) and the machine. For example, a displayed message or a message typed in a laptop, both are fluents at interface, yet the former is perpetrated by the machine, and the latter by some user of the machine.

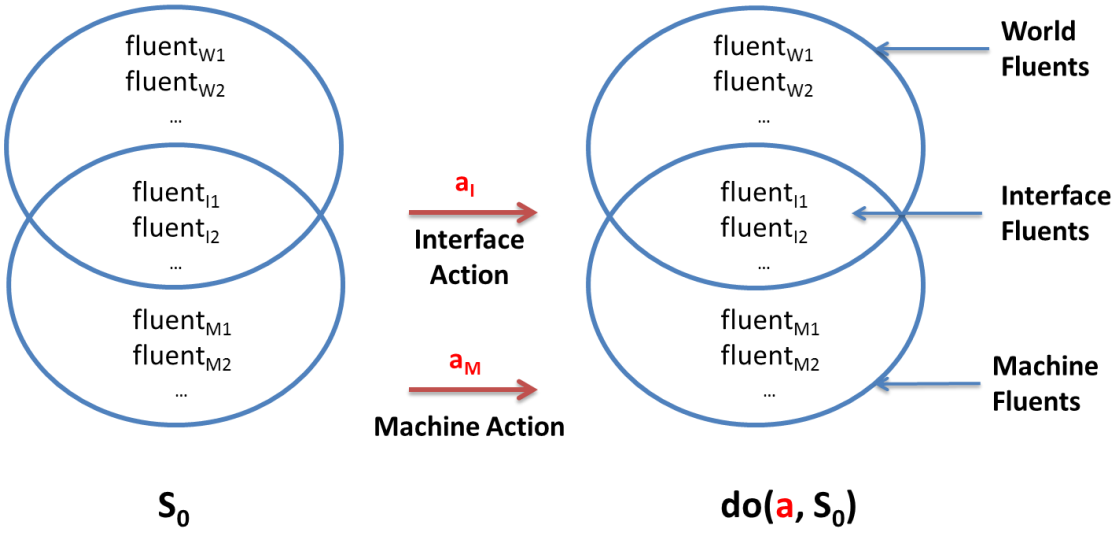


Figure 18. A transition between two situations according to World and Machine Framework

With the statements above, it becomes possible to compose requirements, external specification and internal specification. Requirements should only refer only to World Fluents (e.g. fluent$_{W1}$, fluent$_{W2}$), the External Specification should only refer to Interface Fluents and Interface Actions (e.g. fluent$_{I1}$, fluent$_{I2}$, a_I), and the Internal Specification should only refer to Machine Fluents and Machine Actions (e.g. fluent$_{M1}$, fluent$_{M2}$, a_M). Besides that, the assumptions can refer to all kinds of fluents and actions, and as we shall see in Chapter 6, this provides a clear way to explain how the assumptions play an essential role to link the world and machine states together, and consequently explain how these assumptions produce significant effects on our social lives that a software-driven machine could affect the outside social world without any direct physical means to change it. Before adopting it in the case study, some additional basic rules about the usage of situation calculus are summarized as follows:

- logical symbols used:
  - $\forall, \exists, \wedge, \vee, \neg, \rightarrow, \leftrightarrow, =$;
- second order logic keywords:
  - Poss(a, s): used as the element to compose pre and post conditions.
  - do(a, s): used to indicate the achieved situation by executing the action *a*.
- abbreviations rules:
  - PRE: pre-condition, or precondition axioms
  - POS: post-condition, or effect axioms
- vocabulary rules:
  - variables: lower cased (e.g., meeting)
  - Constants: first letter capitalized (e.g., Meeting, or Mtg)
- quantification rules:
  - all free variables are implicitly universally quantified, unless stated otherwise.

Chapter 4

Towards an Ontological Analysis of Software

4.1 Software Changeability and Hardware Changeability

As we mentioned in the related work, some researchers tried to adopt the 'changeability' as the criterion to distinguish a software object from a hardware object. Such as what Moor proposed, for a computer program as a set of instructions, he interpreted it as software or hardware according to the changeability to the instructions possessed by the software engineers or the users of it (Moor, 1978). For example, in an extreme condition, a person at a factory who can replace circuits in a computer understands his/her activity as giving instructions, then for him/her the programmable circuits could be interpreted as software.

However, Moor's idea is not illuminating for us, as it is counter-intuitive according to the common sense, and this common sense is what we are trying to capture in this dissertation. For us, according to the most general common understanding of the difference between software and hardware, we interpret a software object as a non-physical object, and meanwhile interpreting a hardware object as a physical object. Although we reject the idea to adopt the 'changeability' as the distinguishing criterion, it is indeed a property that links software and hardware tightly. By checking the developing history of automatic computers, we can derive the conclusion that the rapid change of software is supported by the changeability of the hardware, in which the software is embodied.

Pascalines

The first automatic computer dates back to 1642, which was developed by Blaise Pascal to help his father's calculations in business. This machine is called an automatic computer, which is different from a computation assisting tool (e.g. a Chinese abacus), because the user of it does not need to know the *addition* or *subtraction* rules, whenever the numbers are correctly input, the computing result will be provided, while the computing process is made transparent[19] to the user. To memory the name of Pascal, the following similar computing machines are called *Pascalines*, and one of the survivors of these machines is currently presented at the *Musée des Arts et Métiers* as shown in Figure 19.

Figure 19. An early Pascaline on display at the Musée des Arts et Métiers, Paris[20]

[19] We follow the common use of the term 'transparency', meaning the user cannot see the details inside.
[20] Image source: https://commons.wikimedia.org/wiki/File:Arts_et_Metiers_Pascaline_dsc03869.jpg

As we can see that although a pascaline computer can process some kinds of automatic calculations, only one calculation could be processed a time. If the user wants to do the same calculation (e.g., an *addition* calculation), he/she needs to input the *addend* and *summand* manually again. In other words, the command (or instruction) is stored in the user's mind, and every time he/she needs to realize his/her command through the machine's interface.

At this stage, it costs a long time to test a large series of calculations for an ultimate purpose, for every single change in one of these calculations, the user needs to repeat all the calculation inputs manually again. According to this situation, a program as a set of instructions (here we have calculations) is theoretically changeable in the user's mind, but technically hardly changeable according to the interface provided by the pascalines.

Charles Babbage's Computer

According to the changeability of the medium hardware of the corresponding software, the second generation of automatic computers should be the ones that were invented by Charles Babbage around 1830s. He spent nearly all his working time to design such a machine which is different from Pascal's computer that a sequence of calculations should be materialized on some medium (e.g., a paper punched card), and by reading this punched card, the computers can process the preset series of calculations automatically.

Following this idea, the change of the sequence of calculations is switched from operating with the interface of a pascaline to operating on the corresponding punched cards. As long as these punched cards are built or edited, they could be tested on a Babbage's automatic computer directly. With some advanced punched card editing tools, the easy changeability of the punched cards enables quick changes in a program (as a sequence of instructions/calculations).

John von Neumann's Computer

Consequently, the third generation of computers, and also the prototype of the modern computers, were designed and developed by John von Neumann around 1952, which was called IAS (Institute for Advanced Study). He's idea could be summarized as that as the modern computers had been adopting electronic devices as components, it is not reasonable to keep the *Decimal Notation* anymore, and the *Binary Notation* should be a better choice, because it matches the status of a circuit better.

On the other hand, he believed that it is better to store such binarized sequence of calculations (a program) within one of the components of a computer, and this component is called the memory of the computer later. According to this design, the programmer can call and edit the program directly within the computer, without dealing with the punched card anymore. At this stage, the changeability of the program is supported by the easily changeable states of the circuits.

Along with the developments of changeability of the physical mediums, the embodied programs experienced the progress of technically hard to change, then technically relatively easy to change, and finally till now technically very easy to change. Such progress saves the programmers from the cost of long time editing, persuading them to try possible changes as much as possible, and meanwhile developing their potential innovations and creativities as much as possible.

4.2 Software as an Artifact: from Code to Programs

From the changeability of software and hardware, we turn back to the genuine nature of software in this sub-chapter. In the literature, there are a number of entities that are typically conflated with the notion of software. Prominent among them are the notions of *program* and *code*. In the sequel, we argue that these two notions are distinct. In the next chapter, we argue that other notions are required.

Let us start with computer code. We take *computer code* as a well-formed[21] sequence of instructions in a Turing-complete language. Since such instructions are mere sequences of symbols, the identity of code is defined accordingly: two codes are identical if and only if they have exactly the same syntactic structure. So, any syntactic change in a code *c1* results in a different code *c2*. These changes may include variable renaming, order changes in declarative definitions, inclusion and deletion of comments, etc.

A code *implements* an algorithm. Following (Irmak, 2013), we take here an algorithm to mean a pattern of instructions, i.e. an abstract entity, a sort of process universal that is then correlated to a class of possible process executions. So, two different codes *c1* and *c2* can be *semantically equivalent* (e.g., by being able to generate the same class of possible process executions) if they implement the same algorithm, while being different codes. For instance, if *c2* is produced from *c1* by variable renaming, inclusion of comments and modularization, *c2* can possess a number of properties (e.g., in terms of understandability, maintainability, aesthetics) that are lacking in *c1*.

As we have seen, there are proposals (Lando et al., 2009) that identify the notions of program and computer code, while others (Eden & Turner, 2007), (Oberle, 2006) distinguish program-script (a program code) from program-process (whose abstraction is an algorithm). However, we agree with Irmak that we cannot identify a program either with a code, a process, or an algorithm. The reason is that this view conflicts with common sense, since the same program usually consists of different codes at different times, as a result of updates[22]. What these different codes have in common is that they are selected as constituents of a program that is intended to implement the same algorithm. To account for this intuition, we need the notion of (technical) artifact.

We are aware that many such notions have been discussed in the literature, but the one proposed by (Guarino, 2014) derived from (Baker, 2004) works well for us: Firstly, they make clear that artifacts are the results of intentional processes, which, in turn, are motivated by intentions (mental states) of (agentive) creators. Moreover, they connect the identity of an artifact to its functions defined in the *design*, i.e., what the artifact is intended to perform. Finally, they recognize that the relation between an artifact and its proper function exists even if the artifact does not perform its proper function. In other words, the connection is established by the intentional act through which the artifact is created.

In the light of these observations, a code is not necessarily an artifact. If we accidentally delete a line of code, the result might still be a computer code. It will not, however, be 'intentionally made to serve a given purpose' according to a design. Moreover, we can clearly conceive the possibility of codes generated randomly or by chance (for instance, suppose that, by mistake, two code files are accidentally merged into one). In contrast, a program is *necessarily* an artifact. A computer program is created with the pur-

[21] So, we do not consider so called 'ill-formed code' as code, but just as text.
[22] Irmak also admits that the same program may have different algorithms at different times.

pose of playing a particular *proper function.* But, what kind of function? Well, of course the *ultimate* function of a program is –typically– that of producing useful effects for the prospective users of a computer system or a computer-driven machine, but there is an *immediate* function which belongs to the very essence of a program: producing a desired behavior, when the program is executed, *inside* a computer endowed with a given *programming environment* (such as an operating system). We insist on the fact that such behavior is first of all *inside* the computer, as it concerns phenomena affecting the *internal* states of its I/O ports and memory structures, not the *external* states of its I/O devices. Examples of such behaviors can be changes inside a file or a data structure, resulting from the application of certain algorithm. In summary, a program has the essential property of being intended to play an internal function inside a computer. Such function can be specified by an *internal specification* consisting of a data structure and the desired changes within such data structure[23]. For every program we assume the existence of a unique *specification* of such expected behavior, called *internal specification*. In order for a program to exist, this specification must exist, even if only in the programmer's mind.

Since code and program differ in their essential properties (programs are necessarily artifacts and possess essential proper functions; codes are not necessarily artifacts), we have to conclude that a program is not identical to a code. However, if program and code are different entities, what is the relation between the two? In general, the relation between an artifact and its material substrata is taken to be one of *constitution*. As noted in (Baker, 2004), the basic idea of constitution is that whenever a certain aggregate of things of a given kind is in certain circumstances, a new entity of a different kind comes into being.

So, when a code is in the circumstances that somebody, with a kind of *act of baptism,* intends to produce certain effects on a computer, then a new entity emerges, constituted by the code: a *software program*[24]. If the code does not actually produce such effects, it is the program that is faulty, not the code.

Consider now a program, constituted by a certain code. Let us observe first that this is not a physical object, as it lacks a location in space. So, in pace with Irmak, we take a program to be a particular kind of *abstract (non-physical) artifact*. On one hand, it behaves like a type (or universal) since it defines patterns that are repeatable in a number of copies. On the other hand, unlike a type, a program is not outside space and time. A program does not exist eternally like other abstract entities such as numbers and set; in particular, a program does not pre-date its creator. As previously mentioned, it is in fact historically dependent on an intentional 'act of baptism' and, hence, on its creator. In addition to such historical dependence, we shall assume that a program constantly depends on some substratum in order to exist, in the sense that at least a physical encoding (a *copy*) of its code needs to exist. Finally, we shall also assume that, whenever a program exists, its underlying intention to implement the internal specification is recognizable by somebody (possibly thanks to suitable annotations in the code). So, a program *p* is present at time *t* whenever: i) a particular code *c* is present at *t* (which means that at least a copy of *c* exists at *t*); ii) an internal specification *s* exists at *t*; and iii) at *t*, there is somebody who recognizes *c* as intended to implement *s*, or there is an explicit description of this intention (e.g., via a documentation in the code or in an explicitly described internal specification) which is recognizable by someone.

[23] Such specification covers the functional aspects of a program. Although currently not introduced in this work, a full specification may also include non-functional aspects, such as time and security constraints.

[24] We adopt the term 'software program' to keep the terminology consistent in the following contents of the dissertation, and sometimes mentioned as 'program' in short if the meaning is clear in the context.

In conclusion, a syntactic structure could be used as an identity criterion of a code, and an internal specification along with the intentional creation act could be used as the identity criteria of a program. As we have seen, one of the interesting aspects distinguishing program from code is the possibility to honor the commonsense idea shared among software engineering practitioners that a program can change its code without altering its identity.

4.3 Software as an Information Artifact

Taking a further step, from interpreting software as an abstract artifact, we interpret it as is an abstract information artifact. Since the famous semiotic triangle was proposed by (Ogden et al., 2001), it has been more than 10 years researchers trying to figure out the true nature of information. (Smith, 2004) and (Ferrario & Oltramari, 2005) tried to explain what are 'mind' and 'concept' respectively. Following them, (Maass, Goyal, & Behrendt, 2004), (Gangemi et al., 2004) and (Jureta et al., 2009) provided several proposals about 'information object'. Among the researchers, Fortier and Kassel (F&K hereafter) proposed an ontology of Information and Discourse Acts (I&DA) (Fortier & Kassel, 2004), in which a systematic ontological analysis was provided about the information and relating concepts. In their paper, three main concepts were proposed: Information (called 'Content' originally), a non-physical object, the mind or knowledge itself; Expression (called 'ContentBearingObjects' originally), a syntactic representation of the information, which is also a non-physical object; Inscription, the physical realization of the syntactic representation which is the physical support of the information.

F&K's proposal is clear and intuitive enough to explain the information related concepts, and we share the similar idea with some terminology modification. An example could make it clearer: I have an idea that I love my dog, and this thinking content/knowledge is the *information* in my mind; then I want to express this idea to someone else, then I figure out an sentence as 'I love my dog'. I haven't written it down anywhere, this syntactic sequence of symbols according to English is called *expression*; finally, I write down the sentence on a piece of paper, the structure of the ink on the paper forms the *inscription* of the *information*. Note the inscription depends on the physical *medium* but it is not the *medium*.

Recently, by considering both concepts of information and artifact, (Smith et al., 2013) provided a definition of *Information Artifact* (IA) based on their Information Artifact Ontology (IAO)[25] which was proposed for the Human Genome Project, '*an information artifact is an entity that has been created through some deliberate act or acts by one or more human beings, and which endures through time, potentially in multiple copies.*' For the case of simplification, we translate this definition with our terminologies as '*an information artifact is an artifact which is constituted by an inscription of the information*.'

Smith's proposal is suitable to deal with the information which is inscribed with physical support, such as my passport, and this passport depends on the paper and ink but does not equal to them. Like Dublin Core[26], they try to provide terminologies to describe such kind of artifacts, such as '*IA #12345 is-about some given person, or uses-symbols-from some specified symbology, or links-to some second IA #56789* (Smith et al., 2013).' Although the '*information artifact*' proposed by Smith is essential is many cases, we treat it from a different perspective that an *information artifact* is an artifact which is constituted by a syntactic expression of the information.

[25] IAO, https://code.google.com/p/information-artifact-ontology/
[26] Dublin Core, http://wiki.dublincore.org/index.php/User_Guide

We believe it is also important to independently consider the syntactic expression of information as the constituent of IA. Furthermore, as stated by (Kassel, 2010), 'we treat information artifact (e.g., especially software) as non-physical artifact instead of physical artifact'. Similar view was hold by (Faulkner & Runde, 2011) that software is non-material object and which is 'a *syntactic* entity … in consisting of a set of well-formed expressions (Lando et al., 2009) written in an appropriate language, and where *well-formed* means that these expressions adhere to the syntactic and semantic rules of that language.' Consequently, the statements above lead us to our proposal that software is an abstract information artifact.

Chapter 5

From Software Programs to Software Products

5.1 Identifying Different Kinds of Software Artifacts

5.1.1 From Software Programs to Software Systems

As we have seen, the identity criteria of software programs are bound to the *internal* behaviors of a computer. On the other hand, software is usually intended as an artifact whose ultimate purpose is constraining the behaviors of an environment *external* to the computer, which the computer monitors and controls by means of *transducers* bridging between symbolic data and physical properties. In the case of a *stand-alone computer* such transducers just concern the human-computer interface and the standard I/O devices; for *mobile* systems they may also include position and acceleration sensors, while in the case of *embedded systems* they take the form of ad-hoc physical sensors and actuators. So, in the general case, the software's ultimate purpose is achieved by executing a software program that produces certain effects inside a computer, which drives a physical machine, which in turn produces certain effects on its external environment.

In software engineering, the desired effects the software is intended to have on the environment are called *requirements*. The role of the sub-discipline of software engineering called *requirements engineering* is to elicit, make explicit and analyze these requirements in order to produce a *specification* that describes the behavior of a (computer-based) machine. We assume that a software specification is a functional specification, as defined in standards such as IEEE-STD-830-1993. From an ontological point of view, functions are existentially dependent entities that are intimately related to the ontological notion of disposition (capacity, capability) exhibited by an object. Functions and dispositions are potential (realizable) properties such that when a situation (state of the world) of a particular kind obtains they are manifested through the occurrence of an *event*, determining in this way the object's behavior.

As we stated in the baseline, this view has been described in several papers by Jackson and Zave (Michael Jackson & Zave, 1995), (M Jackson, 2007), (Zave & Jackson, 1997), which draw a clear distinction between the environment and the computer-driven machine. Their goal is to show that the relationship between the two, within their WRSPM Framework, can be defined by establishing a logical connection between the intended behaviors at the interface (described by a *specification S*), the relevant world assumptions about environmental properties (described by a body of *world knowledge W*), and the intended environmental behaviors (described by a set of *requirements R*). Such connection is captured by the formula $S \wedge W \vDash R$, and in this case that *S satisfies R* under the world assumptions *W*.

As shown in Figure 20, the picture gets clearer in a more recent paper (Gunter et al., 2000), where the inside part of the software-driven machine is connected to the interface through a generic *programming platform* (described as a set of statements in *M*) which is composed of a physical programming environment part (e.g., multiple transducers), as well as a non-physical programming environment part (i.e., an operating system). Such a description of the programming platform could be understood as a set of *machine assumptions*, based on which the interface behaviors could be affected by means of a software

program (described as a set of statements in *P*), and this extension of the original model was summarized in the formula $P \wedge M \vDash S$.

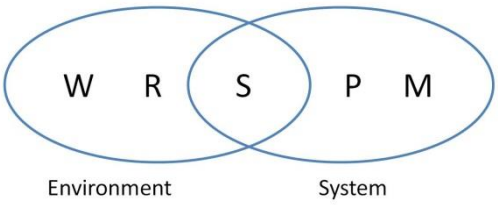


Figure 20. An extract of Jackson and et al.'s WRSPM Framework

Nowadays, although this model has been adopted as a fundamental model for *requirements engineering*, for requirements engineers, they usually focus on the role of the specification of interface behaviors, and the interactions between the interface and the outside world. Differently from that, by taking the view of software engineering, we are interested not only in the interface behaviors, but also in internal machine behaviors which drive the interface, and ultimately in the relationship between the machine behaviors and the world behaviors. As (Gunter et al., 2000) observe, such a relationship can be obtained as a composition of the two aforementioned formulas that $P \wedge M \vDash S, S \wedge W \vDash R$ which indicates: *If (i) S properly takes W into account in saying what is needed to obtain R, and (ii) P is an implementation of S underlying M, then (iii) P implements R as desired.*

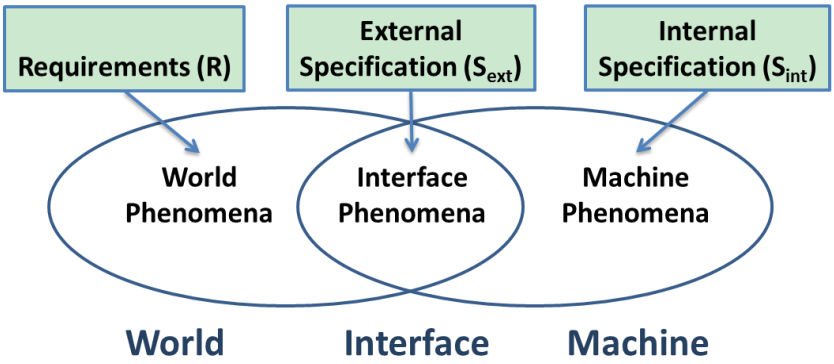


Figure 21. Cutting the boundary between worlds and machines (according to WM Framework)

In a word, Jackson and et al.'s work pay attention to the boundary between worlds and machines, drawing a clear distinction between the world and software-driven machine. According to their idea and the software engineering scenario, we classify and name the phenomena of interest into three categories according to their controllability and visibility as shown in Figure 21, namely: *world phenomena*, which can only be seen and controlled by the world; *interface phenomena*, which can be seen both by the world and the machine, and can be controlled either by the world or by the machine; *machine phenomena*, which can only be seen and controlled by the machine. In (Wang et al., 2014b) and (Wang et al., 2014a), we proposed the term 'internal specification' (S_{int}) to refer to a specification that only constrains the phenomena happening inside the machine; we distinguish it from the 'external specification' (S_{ext}) (originally called simply 'specification'), which constrains the phenomena happening at the interface.

In conclusion, while in the previous chapter we focused on the *immediate* function of *software programs* as information artifacts, producing some effects inside a software-driven machine, and here the

WM Framework allows us to understand how software programs play their *ultimate* function, which is producing certain effects in the external environment (i.e., satisfying the requirements).

Such function is realized in two steps: first, the internal computer behavior resulting from running the software program generates some effects at the interface (i.e., through the programming platform, including the I/O transducers, for instance, a message appears on the screen); second, under suitable assumptions concerning the environment (for instance, there are people able to perceive and understand the message), the ultimate effects of the program are produced (e.g., the person who reads the message performs a certain action).

The presence of these two steps in realizing the ultimate function of a program suggests us to introduce two further artifacts, a *software system*, whose essential property is being intended to determine the intended behaviors at the interface; and a *software product*, whose essential property is being intended to determine some desired effects in the environment *by means* of the behaviors at the interface, given certain domain assumptions. So, the Jackson and Zave's model can be replicated in our WM Framework at three different levels as shown in Figure 21, each corresponding, in our proposal, to a different kind of software artifact, based on the different reasons for why a certain piece of code is written, which are summarized in the following list:

- a *software program* is constituted by some *code* intended to determine a specific behavior inside the computer. Such behavior is specified by an *internal specification*.
- a *software system* is constituted by a *software program* intended to determine a specific interface behavior (between the machine and its outside environment). Such behavior is specified by an *external specification*.
- a *software product* is constituted by a *software system* designed to determine specific effects in the environment as a result of the interface behavior, under given domain assumptions. Such effects are specified by the *requirements*.

5.1.2 From Software Systems to Software Products

Let us now focus on the notion of *software product* previously introduced. While the essential function of a software system is to control the interface behaviors between a certain machine and the environment (i.e., according to the Jackson et al.'s approach, that part of the behavior that is 'visible' to both the environment and machine), the essential function of a software product is to control the environment's behaviors which are not visible to the machine, but can be influenced by it, under given environment (domain) assumptions, as a result of the interaction between the interface and the environment.

It is important to note that a software product is intended to achieve some effects in the external environment *by means of a given machine*, and *under given world assumptions*. So, assuming they have exactly the same high-level requirements, MS Word for Mac and MS Word for PC are different software products (may belong to the same *product family*), since they are intended for different kinds of machines. Similarly, country-oriented customizations of Word for Mac may be understood as different products, since they presuppose different language skills, unless the requirements already explicitly include the possibility to interact with the system in multiple different languages.

Consider now one such software product, say MS Word for Mac. Starting with version *V1*, which denotes the specific software system constituting the software product at the time of its first release, this product will suffer a number of possible changes in its constituents in order to fix bugs, to include new functionalities, to improve performance, security, precision, etc. Each of these changes leads to distinct code, but some of them (those that are not just bug fixings) will also lead to a new software program, while others (those that concern changes in the interface functions) will also lead to a distinct software system, namely to a different version (*V2*) of *the same product*.

To reflect these changes, *the code* will be marked in order to distinguish itself from the former codes, and in order to identify the program, the software system, and the software product. According to the usual conventions, the software system could be identified by the *version number*, the software program by the *release number*, and the code by the *sub-release number*. In this way, we see how an ontology of software artifacts, based on the WM Framework, can produce a version numbering system which reflects the changes in the different kinds of software artifacts.

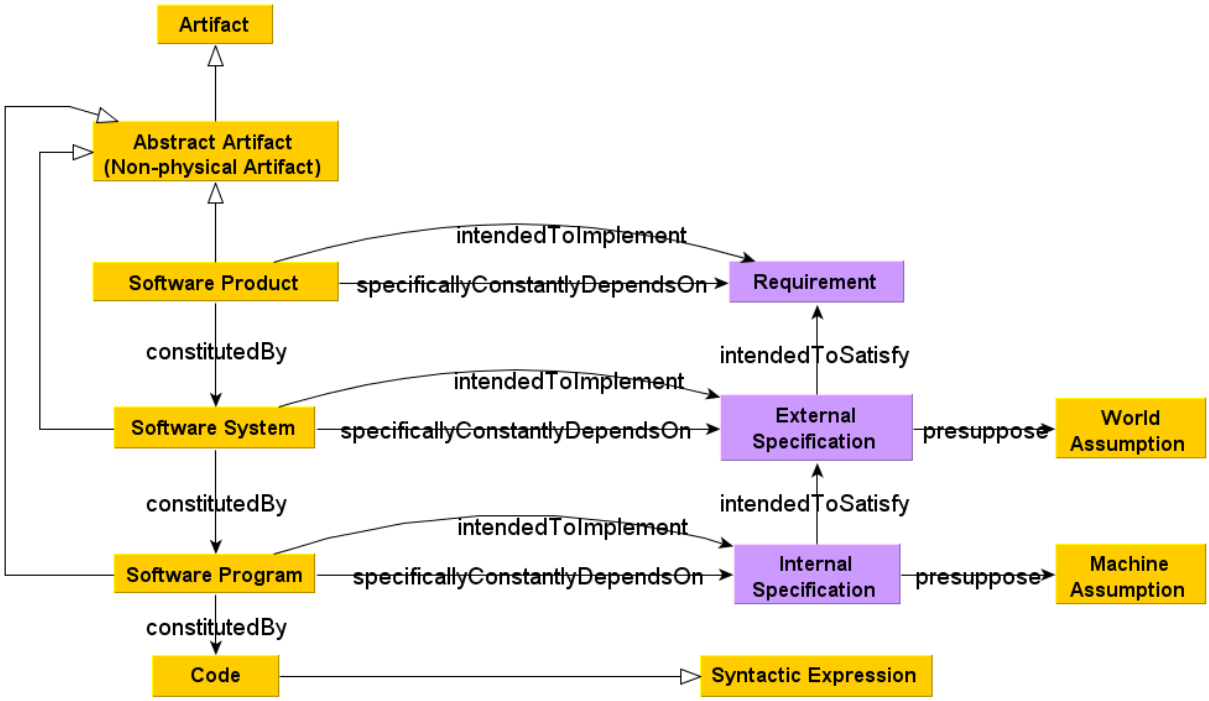


Figure 22. Different abstract software artifacts induced by different requirements engineering notions.

In summary, the core ontological distinctions induced by the different requirements engineering perspectives we have discussed are illustrated in Figure 22. To the left, we see different software artifacts all ultimately constituted by some code (which is a specific syntactic expression). They have different essential properties, resulting from the fact that each of them is constantly dependent on a different intentional entity. Each of these entities refers to an expected behavior involving different parts of a complex socio-technical system, which in turn emerges from the interaction of a software-driven machine and a social environment (namely the WM Framework).

A brief account of the main relations appearing in the picture is reported below. As usual, the subsumption relation is represented by an open-headed arrow. The closed-headed arrows represent some of

the basic relations discussed in the paper. For some of them (constitution and specific constant dependence), the intended semantics is rather standard, while for others we just sketch their intended meaning, postponing a formal characterization to the future work.

constitutedBy: We mean here the relation described extensively by (Guarino, 2014) and (Baker, 2004). We just assume it being a kind of generic dependence relation that is both asymmetric and non-reflexive, and does not imply part-hood. We can borrow a minimal axiomatization from the DOLCE ontology.

specificallyConstantlyDependsOn: If *x* is specifically constantly depending on *y*, then, necessarily, at each time *x* is present also *y* must be present. Again, we can borrow the DOLCE axiomatization. When this relation holds, being dependent on *y* is for *x* an essential property.

intendedToImplement: This relation links an artifact to its specification or requirement, as a result of an intentional act. Note that the intention to implement does not imply that the implementation will be the correct one (e.g., bugs may exist).

intendedToSatisfy and **presuppose.** These two relations are proposed to capture the structure of the formula proposed by Jackson et al. to describe the nature of requirements engineering, $A, S \vDash R$. *S*, which presupposes *W*, is intended to satisfy *R*. Presupposition is a kind of historical dependence on certain knowledge states.

5.1.3 The Social Nature of Software

In addition to its artifactual nature, discussed in the previous chapter, software –at least software used every day in our society– has also a strong social nature, which impacts on the way it is produced, sold, used and maintained. There are two main social aspects of software we shall consider under our software artifactual perspective: *social recognition* and *social commitment.*

Social Recognition and Software Identity

We have seen the key role the constitution relation plays in accounting for the artifactual nature of software. But how is this constitution relation represented and recognized? In the simplest of cases, we can think of a program produced by a single programmer for personal use. In this case, we can imagine that the constitution relationship binding a program with its constituting code exists solely in the mind of this programmer. Likewise, if this program comes to constitute a software system, then this constitution relation, again, exists only in the mind of the programmer. Yet, in order for a software artifact to exist in a social context, we shall assume that the constitution relation between the artifact at hand and its constituent needs to be explicitly communicated by the software author, and recognizable by a community of people. As a minimal situation, we consider these communications about constitution and intentions to satisfy specifications as true communicative acts that create expectations, beliefs and contribute to the creation of commitments, claims and a minimal social structure (possibly reflecting division of labor) between the software creator(s) and the potential users or stakeholders. Once this social structure exists, the creators' actions become social actions and are subject to social and legal norms that support expectations and rights. To cite one example, we use the motion picture 'The Social Network' based on the book 'Accidental Billionaires' (Mezrich, 2010) reporting on the creation of Facebook. As shown there, the legal

battle involving the authorship rights in Facebook was at moments based on the discussion of shared authorship between M. Zuckerberg and E. Saverin regarding an initial program (Saverin was allegedly a prominent proposer of the algorithm) and software system, much before the product Facebook existed. At other times, the legal battle between Zuckerberg and the Winklevoss brothers was based on a shared system specification of another program even if, as argued by Zuckerberg, no lines of the original code had been used by Facebook.

In more disciplined software engineering settings, anyway, the constitution relationships and the intended specifications are documented by program headers and possibly user manuals or separate product documentation. Notice that, without the explicit documentation of these relationships, the software artifacts will depend on their creators in order to exist, since the constitution relationships are sustained by their intentional states. Once these relationships are documented, these artifacts can outlive their creators, as long as this documentation can be properly recognized and understood. So, for instance, although Joseph Weizenbaum is no longer alive, by looking to a copy of the ELIZA[27] code, one can still reconstruct the chain of intentions from the informal requirements specification all the way down to the code. In formal ontological terms, this means that software artifacts are just historically (but not constantly) depending on their authors, and in addition they are generically constantly depending on a community of people who recognize their essential properties. If such community of people ceases to exist, the artifact ceases to exist.

Social Commitment and Software Licensing

As we have seen, the different kinds of software artifacts we have discussed are based on a requirements engineering perspective. We cannot ignore however another perspective that deeply affects the current practice of software engineering, namely the *marketing perspective*. In the present software market, software products do not come alone, since what companies sell are not just software products: in the vast majority of cases, a purchase contract for a software product includes a number of rights and duties on both parties, including the right to download updates for a certain period of time, the prohibition to give copies away, the right to hold the clients' personal data and to automatically charge them for specific financial transactions, and so on. Indeed, the very same software product can be sold at different prices by different companies, under different *licensing policies*. The result is that software products come to the market in the form of *service offerings*, which concern *product-service bundles.* According to (Nardi et al., 2013), a *service offering* is in turn based on the notion of *service*, which is a social commitment concerning in our case maintenance actions. Service offerings are therefore meta-commitments, i.e., they are commitments to engage in specific commitments (namely, the delivery of certain services) once a contract is signed. So, before the contract is signed we have another software entity emerging: a *Licensable Software Product.* After the contract is signed, we have a *Licensed Software Product*. Notice that the services regulated by the contract may not only concern the proper functioning of software (involving the right to updates), but also the availability of certain resources in the environment where the software is supposed to operate, such as remote servers (used, e.g., for Web searching, VOIP communication, cloud syncing...). So, when Skype Inc. releases Skype, it publicly commits to engage in such kind of commitments. By the way, this means that, when buying Skype from Skype Inc., Microsoft is not only buying the software product, but it is also buying all the rights Skype Inc. has regarding its clients.

[27] ELIZA, http://en.wikipedia.org/wiki/ELIZA

Note that, even in absence of a purchasing contract, when releasing a product as licensable product, the owner creates already social commitments and expectations towards a community of users and re-users of the product. For example, take the Protégé Ontology editor, which is a free open-source product released under the Mozilla Public License (MPL[28]). This grants the members of the user community the right to change Protégé's code and to incorporate it even in commercial products.

5.2 Recording Different Kinds of Software Artifacts

5.2.1 Representation of Different Kinds of Software Artifacts

As we have identified the essential properties of the different kinds of software artifacts, then they can be used to record the corresponding software artifacts. Hence, in the remainder of this section, we elaborate on how a formalism such as situation calculus could be used to represent these requirements, as well as the external and the internal specifications in the sense put forth by the aforementioned classification of different kinds of phenomena (according to WM framework).

Requirements (R)

We interpret a requirement as a set of (conditional) states of affairs that are intended by stakeholders. According to this view, we can represent a set of states of affairs as a set of situations. In an intended situation, the concerned world fluents have the specific values intended by the stakeholders. If the requirement has a conditional nature, a situation transition will be specified. For example, a requirement may be 'a meeting shall be scheduled after a meeting initiator intends to schedule one', meaning that if we start in a situation where a meeting has been intended by an initiator, some actions will get us to a situation where the intended meeting is scheduled. Following this view, we can represent this requirement in a situation calculus expression such as the following one:

$$
\begin{aligned}
R_0 : \exists a\{ & Poss(a,s) \leftrightarrow \\
& meeting\_intended_W(initiator, meeting, s) \\
& \wedge Poss(a,s) \rightarrow [meeting\_scheduled_W(meeting, s') \\
& \wedge s' = do(a,s)]\}
\end{aligned}
$$

This formula means that there exists an action *a* such that, starting from an initial situation *s* in which a *meeting* is intended by an *initiator*, then, by executing it, a new situation in which the meeting is scheduled is reached. The action *a* here in the formula refers to an action variable in terms of second order logic representation syntax. Also, note that we use the subscripts *w (world), i (interface),* and *m (machine)* to distinguish among fluents concerning the different kinds of phenomena described in the WM Framework.

In a software engineering process, such an action could be understood as an alias of a function, and this decision is adopted by many software engineering standards, such as IEEE-STD-830-1993. Hence a set of functions defined in the specifications becomes a solution to the requirements. As previously stated,

[28] MPL, http://www.mozilla.org/MPL/

according to world-machine distinction, we classify specifications into external specification and internal specification respectively, and here we introduce the representation of each of them as follows[29]:

External Specification (S_{ext})

An external specification contains a set of actions $S_{ext} = \{a_{I1}, a_{I2}, \ldots, a_{In}\}$, that are supposed to occur at the interface in order to satisfy the requirements. If the specification is implemented correctly, executing an interface action will bring about the desired situation transition, resulting in changes of the interface fluents, that (by definition), will be visible both from the machine and outside world.

Besides that, it is important to highlight that, an action in S_{ext} may be applied several times to get the stakeholders to an intended situation, and here the detailed executing order of this series of actions is ignored. For example, for a meeting scheduling system, if S_{ext} includes an interface action 'receive timetable from a participant', this action will have to be applied many times to get the timetables from all the participants for a meeting, so that the stakeholders can reach the situation where all timetables have been collected.

Internal Specification (S_{int})

An internal specification contains a set of machine actions $S_{int} = \{a_{M1}, a_{M2}, \ldots, a_{Mn}\}$, concerning the inside part of the machine (e.g., the variables in I/O registers). Executing a machine action will bring about a situation transition, resulting in intended changes in the machine fluents. Although a user of a software system may not be interested in the implementation details underlying the interface, software engineers care about it, and they need to provide the actions inside the machine supplementing the interface actions. Together with both the external specification and internal specification, we get a complete solution to the requirements. Similarly as stated in external specification, the machine actions in an internal specification are also introduced here without providing the details about their execution orders.

5.2.2 Ontology-Driven Software Configuration Management

According to (Dart, 1991), Software Configuration Management (SCM) is 'a discipline for controlling the evolution of software systems', and is considered as a core supporting process for software development (Chrissis, Konrad, & Shrum, 2011). A basic notion of any SCM system is the concept of version (Estublier et al., 2005). The IEEE *Software Engineering Body of Knowledge* states (Bourque & Fairley, 2014) that 'a version of a software item is an identified instance of an item. It can be thought of as a state of an evolving item'. In the past, the same source distinguished, within versions, between revisions and variants (Abran & Moore, 2004): 'A *revision* is a new version of an item that is intended to replace the old version of the item. A *variant* is a new version of an item that will be added to the configuration without replacing the old version'. In our approach, these two kinds of version can be described as follows:

[29] Note that the action mentioned here is actually a concept at the type level, and an action instance could be understood as an execution of the action at the instance level.

Revision Process. Suppose that at time *t* we have a program *p1* constituted by code *c1*; when at time *t'* we replace the code *c1* as the constituent of *p1* by code *c2*, we are not creating a distinct program *p2*, but we are simply breaking the constitution relation between *p1* and *c1*. Thus, at *t'*, *c1* is not a constituent of the program anymore; rather it is merely a code, so that at *t'* we are still left with the same program *p1*, but now constituted by a different code *c2*.

Variant Process. Suppose that we have a software system *s1* ('MST[30]-Finder A'), and we develop a software system *s2* ('MST-Finder B') from *s1* by adopting a new algorithm. Now *s1* and *s2* are constituted by different programs. Of course, *s1* will not be identical to *s2*, since they are constituted by different programs at the same time, and by Leibniz's Law if two individuals have incompatible properties at the same time they are not identical. Indeed, the two software systems may have independent reasons to exist at the same time.

Traditionally, revisions and variants are managed by means of naming conventions and version codes which are usually decided on the basis of the *perceived* significance of changes between versions without any clear criterion (e.g. CVS, SVN). We believe that the layered ontology introduced in this dissertation can make an important contribution to make this process more disciplined by providing a general mechanism to explicitly express what is changed when a new version is created. This can be simply done by pointing to the software artifact that is affected by the change, and can be reflected by a simple versioning scheme (e.g. v 1.5.3.2: 1 - software product release number; 5 – software system release number, 3 – software program release number; 2 – code release number). In addition to this scheme, we can document the *rationale* why a certain software artifact has been changed according to the WM Framework, meanwhile pointing to the specific source of change.

We believe that this ability to account both for what and why software is changed is essential for software engineering, because managing software and software evolution requires much more than managing code. For example, as Licensed Software Products are based on a chain of dependent artifacts culminating with a computer code, a software management system must be able to manage the impact that changes in the code ultimately have in terms of legal and financial consequences at the level of licensed products.

[30] MST: Minimum Spanning Tree

Chapter 6

How Software Changes the World through Assumptions

Jackson and et al.'s work pay attention to the boundary between worlds and machines, drawing a clear distinction between the environment, which is where the ultimate effects of software program are expected, and the software-driven machine where software program operates. Van Lamsweerde recognizes their work in a paper entitled 'from worlds to machines'(Axel Van Lamsweerde, 2009), which elaborates on how a specification concerning the behavior of the machine could be derived from a set of requirements concerning the external world. In this dissertation, we take the reversed perspective, focusing more on how software-driven machines could affect the world. In other words, given a specification and a set of requirements, we explain in what sense the requirements can be entailed from the specification.

As stated in our previous work (Wang et al., 2014a, 2014b), a software program possesses a peculiar characteristic when compared to other kinds of information artifacts (e.g. recipes or laws): it plays the role of a bridge between symbols in a machine and its outside world. More specifically, while other kinds of information artifacts directly refer to the objects in the world (so that executing a recipe or a law implies a manipulation of objects in the world), software programs refer to virtual variables in a machine, whose manipulation inside the machine affects the outside world in an indirect way.

When a software program is embedded in a machine to control its external behavior, we have a *software-driven machine.* The ultimate purpose of a software-driven machine program is to constrain the phenomena of its external environment. The machine monitors and controls the environment by means of *transducers* bridging between symbolic data and physical properties of the environment (hereafter the software-driven machine will be referred as 'machine' if the meaning is clear in the context).

In the case of a stand-alone personal computer (PC) such transducers just concern the human-computer interface and the standard I/O devices; for mobile systems they may also include position and acceleration sensors, while in the case of embedded systems they take the form of ad-hoc physical sensors and actuators. So, in the general case, the software's ultimate purpose is achieved by running a software program that produces certain effects inside a computer, which drives a physical machine, which in turn produces physical effects on its external environment.

However, we may wonder, similarly to what happens to the physical environment, whether software-driven machines can also affect the social environment. Indeed, as we have seen for the case of the meeting scheduler, in many cases the ultimate purpose of software is to produce such changes in the social world. But how is this possible, given that the machine has no direct means to affect the social world? This is the paradox we mentioned at the beginning of this dissertation.

To solve this, let us first consider another scenario in which the social world is affected by artifacts other than software-driven machines. For example, in the modern monetary system, some kinds of colored paper are used as money in commercial activities. As we are all aware of, giving a piece of such paper to a person produces certain effects in the social world (Ryan-Collins et al., 2014). This happens be-

cause we share the same *assumption* that owning one of those pieces of colored paper produces some relevant social effects: that piece of paper *counts as* money.

The *count-as* relationship was proposed by the philosopher John Searle to link what he terms a *brute physical fact* (e.g., presenting a bill) with what he terms an *institutional fact* (having the right to obtain a good in exchange of the bill). As another example, two hands joining in a handshake movement counts as an institutional relationship (e.g., an agreement) being created in a given context. We would say therefore that the agreement *depends* on the handshake (Franken, Karakus, & Michel, 2010) if the assumption that handshakes count as agreements holds.

Moving now to software-driven machines, we can observe that, according to the way the monetary system evolved after the advent of electronic computers, a new assumption is pervasively made by almost all the nations and areas in the world: electronic data in banks' databases *counts as* money. In other words, our commercial world is actually controlled by the software-driven machines in the banks, which literally have that power of changing the world, thanks to our count-as assumptions (Ryan-Collins et al., 2014).

Going back to the meeting scheduler example, again we have a case in which symbolic facts count-as institutional facts, since a meeting record marked as scheduled on the computer *counts-as* as the collective belief that the meeting is actually scheduled, with the corresponding beliefs and commitments from the meeting participants, as well as the corresponding social (and sometimes legal) consequences.

6.1 A Preliminary Ontology of Assumptions

Consider a software system that schedules meetings upon requests. Its ultimate requirement, which the system is mandated to fulfill, is not only to produce some information, consisting of a schedule that satisfies the given constraints, but also to bring about a change in the social world where the software operates, so that the suggestion made by the software is actually understood and assimilated by the working environment, and the proper actions necessary for the meeting organization (such as invitations to the participants and room allocation) are effectively undertaken according to the software suggestions. The actual effectiveness of the software will be evaluated on the basis of its impact in the social world; however, a software system, by its very nature, can only change the states of the machine within which it operates.

There seems to be a paradox here. The requirements for most software systems, the intended states-of-affairs these systems are supposed to bring about, concern their operational environment, usually a social one. But these systems don't have any direct means to change that environment in order to bring about the intended states-of-affairs. In what sense then can we say that such systems fulfill their requirements? The main purpose of this chapter is to account for this paradox. We do so by proposing an ontology of the kinds of assumptions that are implicitly used in software engineering practice, especially the newly proposed two kinds of dependence assumptions.

We explain the key role of these assumptions played in the relationship between the social world and a software-driven machine, which solves the paradox mentioned at the beginning of the dissertation. Besides that, we introduce the interpretations of the term 'assumption' adopted in the software engineering community as 'assumption-needed' and 'assumption-used' in our ontology of assumptions. As we demonstrate in the end of this sub-chapter, these interpretations are corroborated by literature in the legal domain. We emphasize that this ontology is a partial attempt in systematizing these notions. In a future

paper, we expect to elaborate on the detailed ontological nature of these assumptions, in particular, in aspects dealing with the mental states/attitudes.

6.1.1 A Classification of Assumptions

As shown in Figure 23, four kinds of assumptions are added into the WM Framework, including world assumptions (*WA*), machine assumptions (*MA*), world dependence assumptions (*WDA*), and machine dependence assumptions (*MDA*). We introduce each of them in the sequel.

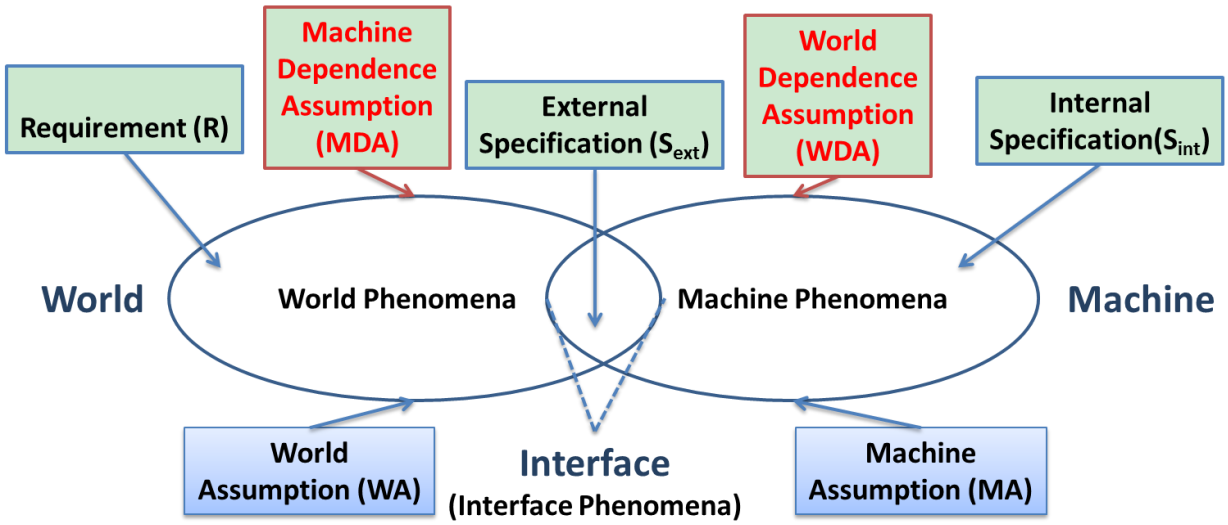


Figure 23. WM Framework with assumptions

A *world assumption* is an assumption about and only about the world phenomena that are not visible to the machine. It constrains the suitable environment context for the software-driven machine to produce the effects specified in the requirements. For example, for a meeting scheduling system, we may assume that the system only works if a room is available for the requested meeting, and design a basic solution that only finds a suitable time slot and selects a room from the available ones. Such an assumption means that our solution does not work when no room is available (e.g. during a busy period with many meeting requests). Note that another implicit world assumption related to this is that all meetings have the same importance, so that the system does not attempt any special recovery strategy (e.g., re-negotiate previously scheduled meetings) based on the meeting's importance.

A *machine assumption* is an assumption about and only about the machine's internal phenomena, i.e., those that are only visible to the machine. For instance, an action specified at the internal interface level (the I/O register) will produce its intended effects only if the assumption that such action is correctly implemented holds. It describes the machine conditions that should be ensured by something or somebody else. In other words, these machine assumptions are usually used to ensure the availability of resources in/for the machine. For example, we main assume that there is always a power supply for the machine, which means that machine should not be expected to work when there is no power supply.

Since world assumptions and machine assumptions are either about the world (e.g., the assumption of enough rooms for the meetings) or about the machine (e.g., the assumption of the power supply for the machine) independently, they are not sufficient to describe the causal connection between the machine

states and the world states: how can the machine change the world? To answer this question we need of course to focus on the interface and consider another two kinds of assumptions: *world dependence assumptions* and *machine dependence assumptions*.

These two additional kinds of assumption are proposed specifically to constrain the relationship between worlds and machines, which are different from world and machine assumptions that only constrain the world and machine phenomena independently. For example, as dependence assumptions, the value reported by a sensor depends on the environment conditions (e.g. the air conditioner can sense the temperature in the environment and translate it into the corresponding value); and, oppositely, the action taken by an actuator depends on the value of a variable in the computer's memory (e.g. the air conditioner will make cool air when the value of the variable is bigger than 30 in the memory).

A *machine dependence assumption* states that an external world phenomenon depends on some machine phenomenon. For instance, we may assume that a certain room is reserved for a certain meeting at a certain time (and therefore it will not be used for any other purpose at that time) if the computer registers that (by means of a suitable message shown on the screen). Another possible example might be that 'the meeting is considered to be scheduled' as soon as all the messages to the participants have been sent. In other words, a certain state of world is *assumed* to exist if a certain phenomenon in the symbolic machine world exists (hence, the direction of *dependence*).

In contrast to a machine dependence assumption, a *world dependence assumption* states that a machine phenomenon depends on some world phenomena. For example, we may assume that whenever a certain room appears to be free on the machine, it is because the room is actually free in the external world. Similarly, we can assume that whenever a meeting appears to be scheduled on the machine, it is because somebody actually intended to schedule such meeting. In other words, a world dependence assumption is an assumption about the truthfulness of the correspondence between states of the machine and the phenomena in the world these states are supposed to represent.

In the case of a machine dependence assumption, the depending phenomenon in the external world could be either physical or social. When it is a physical phenomenon, it means that there is a path of physical interactions connecting the observed phenomenon in the external world with a physical phenomenon occurring in the machine. This is the common scenario in cyber-physical systems, which interact with the external world by means of actuators and sensors. However we are more interested in the case when the depending phenomenon is a social one, which means a social entity is affected in the world 'because of' the existence of a symbolic entity in the machine. In the following we shall explore the nature of such a link between the social phenomena and digital variables, which accounts for the paradox of a change in the social state being dependent on a change in the machine states without direct physical means.

6.1.2 The Causal Chain Enabled by these Assumptions

Four kinds of assumptions were proposed in the preceding subsection. In this sub-chapter, we explain the key role played by these assumptions in linking the world and machine states. We claim that there is a causal chain enabled by these assumptions: some triggering phenomenon occurs in the outside world, it propagates through the interface, and reaches the symbolic states inside the machine; then it comes back from the inside machine to the outside world by crossing back through the interface. Once more, we shall rely on our meeting scheduler case study to explain the causal role of these assumptions.

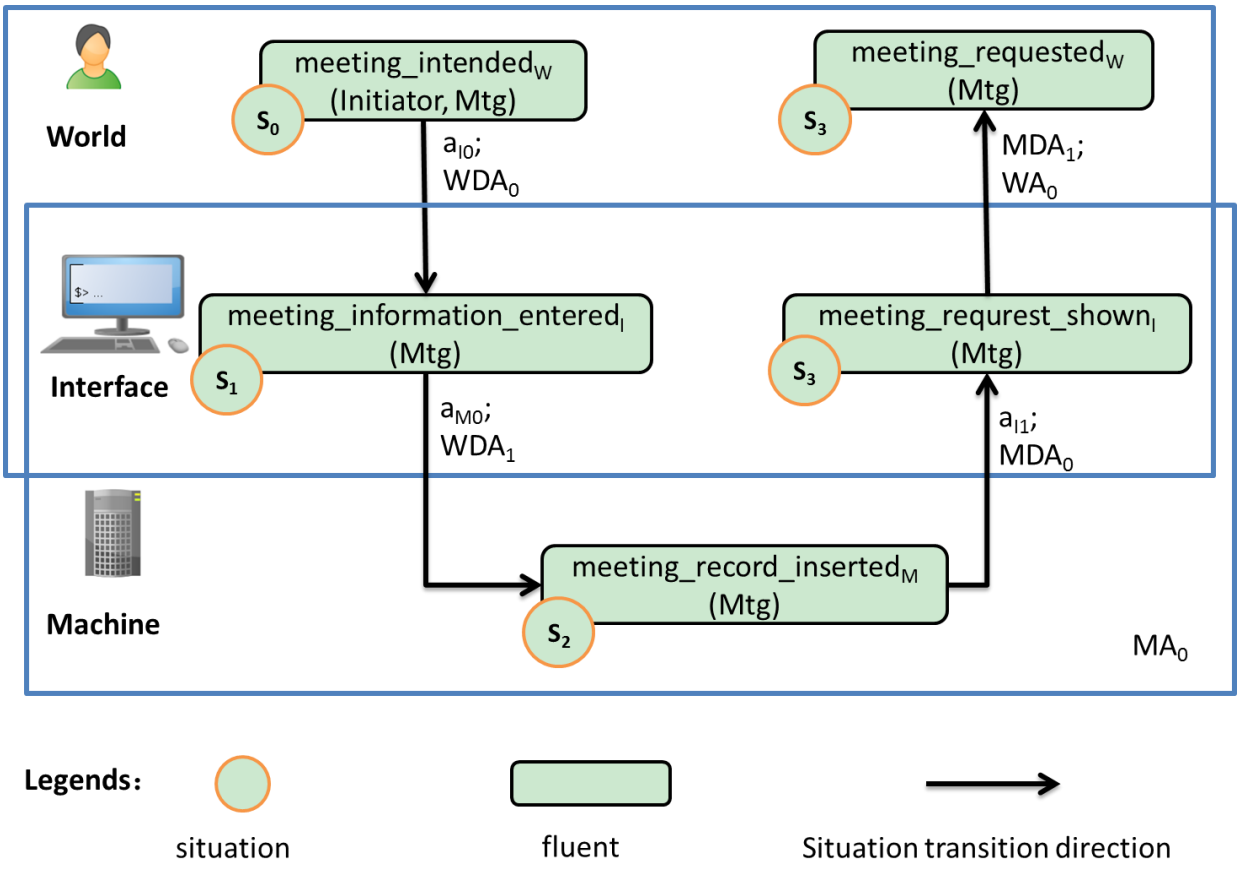


Figure 24. The chaining mechanism underlying software engineering enabled by assumptions

An instance of a whole meeting scheduling process starts from the initial state S_0 in which a meeting initiator intends to schedule a meeting, which could be represented as a world fluent *meeting_intended$_W$(Initiator, Mtg, S_0)*. From this state, the initiator will participate in the action a_{I0} provided by the interface to enter the meeting information at the interface, and this is represented by the action's pre-condition that *PRE: WDA_0: meeting_intended$_W$(Initiator, Mtg, S_0)*, and by finishing the execution, the action will result in a new situation in which the meeting information is entered, represented by the action's post-condition as *POS: meeting_information_entered$_I$(Mtg, do(a_{I0}, S_0))*, and this step can be summarized as a situation transition enabled by the interface action a_{I0} formalized as follows:

$$a_{I0} : Enter\_Meeting\_Information(initiator, meeting)$$

$$PRE : WDA_0 : Poss(a_{I0}, s) \leftrightarrow meeting\_intended_W(initiator, meeting, s)$$

$$POS : Poss(a_{I0}, s) \rightarrow [meeting\_information\_entered_I(meeting, s') \wedge s' = do(a_{I0}, s)]$$

The world dependence assumption WDA_0, asserted in the pre-condition of the interface action a_{I0}, is meant to capture the relation between a state in the machine (representing the possibility of the execution of action a_{I0}) and a state of the world (the mental attitude, i.e., the intention of a given human agent).

Without this assumption, there is no means to constrain the execution of the action a_{I0} to depend on a state of the human mind (represented here by $meeting\_intended_W(initiator, meeting, s)$. In other words, the situation transition from *s* to *s'* enabled by the action a_{I0} won't work without WDA_0 being somehow ensured. This transition can be visualized by the single direction arrow between S_0 and S_1 in Figure 24.

When the meeting information is entered, the machine can sense this change in $meeting\_information\_entered_I(meeting)$, provide an machine function a_{M0} to receive the information from the interface, and then insert a corresponding meeting record into the database, resulting in the change in the machine fluent $meeting\_record\_inserted_M(meeting)$. Once more, the formalization of the action a_{M0} could be summarized as follows:

$$a_{M0} : Insert\_Meeting\_Record(meeting)$$
$$PRE : WDA_1 : Poss(a_{M0}, s) \leftrightarrow$$
$$meeting\_information\_entered_I(meeting, s)$$
$$POS : Poss(a_{M0}, s) \rightarrow$$
$$[meeting\_record\_inserted_M(meeting, s')$$
$$\wedge s' = do(a_{M0}, s)]$$

The action a_{M0}'s pre-condition follows a_{I0}'s post-condition, and this ensures the continuity of the situation transitions from S_0 to S_1 and then to S_2. No surprise that another world dependence assumption WDA_1 is made, and asserted into the pre-condition of the action a_{M0}. This assumption indicates that the execution of the action depends on the physical framework provided by the interface. In other words, when this assumption is fulfilled, the machine could sense the interface fluent and then execute the corresponding action a_{M0}.

Till now, the route of the changes in fluents has travelled from the outside world into the inside machine through the interface. Now, it is time to characterize how it can travel back to the outside. To achieve that, we need to reach the interface first, and another interface action a_{I1} is introduced. By sensing the value of the machine fluent $meeting\_record\_inserted_M(meeting)$ through the machine dependence assumption MDA_0, the action a_{I1} will be executed resulting in the change in the interface fluent $meeting\_request\_shown_I(message)$ indicating the message of 'meeting is successfully requested' is shown on the screen. As before, the action a_{I1} can be represented as follows, yet note that a machine assumption MA_0 is also inserted to ensure the power supply for the whole process of the situation transitions.

$$a_{I1} : Show\_Meeting\_Request(meeting)$$
$$PRE : MDA_0 : Poss(a_{I1}, s) \leftrightarrow$$
$$meeting\_record\_inserted_M(meeting, s)$$

$$POS : Poss(a_{I1}, s) \rightarrow$$

$$[meeting\_request\_shown_I(meeting, s')$$
$$\wedge s' = do(a_{I1}, s)]$$

MA_0 : $power\_supply_M(Machine)$

Now, we reach the interface successfully, and there is only one step left to travel back to the outside world. To fill this gap, we make to two additional assumptions: 1) MDA_1 indicates that as soon as the message is shown on the screen in the interface, we assume that the stakeholders all will agree that the meeting is requested; 2) WA_0 constrains the system is designed for people who are not visually impaired.

$$MDA_1 : meeting\_requested\_message\_shown_I(message, s) \leftrightarrow meeting\_requested_W(meeting, s)$$

$$WA_0 : can\_see_W(people)$$

A brief demonstration of the role of assumptions in a software engineering process is presented in this sub-chapter. As one can see, the fluent changes through the situation transitions follow the sequence of S_1, S_2, and S_3. It travels from the outside world, crossing the interface, reaches the inside machine, then it comes back to the outside world crossing the interface again. This sequence forms a chain-like structure, and we claim that the assumptions proposed here play the key role in ensuring this structure, i.e., linking the world and machine states together.

6.1.3 Interpretations of the Concept of Assumption

'Assumption'[31] is a severely overloaded term used in many communities (e.g., research, industry, and etc.) as well as in our daily lives. The interpretations of this term diverge significantly in different contexts. In the sequel, we revisit the examples of these possible interpretations mentioned in the baseline (relying on the natural language use of the term) (Ennis, 1982):

Conclusion: e.g., Tom said: 'my assumption is that you are going out, since you are wearing your cap.' The conclusion of 'going out' is derived from the current situation 'wearing your cap'.

Less-than-fully established proposition, in an accusation sense: e.g., Mike answered: 'that is only your assumption, you don't know it.' Mike replied that it might look like he's going out, yet that was only Tom's guess and as such it is not guaranteed to hold.

Adopted in order to deceive, fictitious, pretended: e.g., 'although bad things happened, please assume that they didn't ever happen.' The term assumption is interpreted as a kind of 'self-deception' here that 'you can deceive yourself that nothing bad happened'.

The examples aforementioned are only a small part of a full possible list. However, considering the importance of the role played by assumptions in software engineering process, it is necessary for the stakeholders to achieve an agreement on the interpretation of this term. Fortunately, a clarification of this term was proposed by Ennis in 1982, providing clear guidance to interpret this term according to its use in practice (Ennis, 1982). More precisely, he classified the assumptions into two main kinds, namely '*assumptions-used*' and '*assumptions-needed*'. Assumptions-used are the propositions that a person uses *a*

[31] The interpretations of assumption mentioned in this subsection are orthogonal with the preceding four kinds of assumptions.

priori while constructing a new argument. Usually, assumptions of this kind are adopted by scientists as the foundational components of a theory (e.g., the law of gravity). On the other side, assumptions-needed are the propositions that are needed *a posteriori* to support a previous conclusion. In this sense, they circumscribe the context within which the conclusion is reasonable.

In what follows, we make use this distinction between assumptions-used (*AU*) and assumptions-needed (*AN*). We use it to elaborate on the aforementioned Jackson and Zave's formula. In particular, given a set of requirements *R* and a set of assumptions-used *AU*, one needs to find a specification *S* and a set of assumptions-needed *AN* such that $AU, AN, S \vDash R$. To use a simple example, suppose one is given a requirement *R*: 'Fly to the moon' and *AU*: 'laws of gravity'. This engineer then needs to find a specification for a spacecraft *S* that will fulfill *R* provided that the following assumption *AN*: 'spacecraft carries enough fuel' holds.

The choice of interpreting assumptions as assumptions-used or assumptions-needed has strong practical effects, which is illustrated by the lawyers who deal with the disputes between the users and product providers. As software is usually also an instance of such kind of products, provided by software engineers, and used by the software users, this distinction can also be used to illuminate the disputes between software users and software engineers. For instance, with a different terminology, Twerski and his colleagues stress the key issue which is to justify a case on either design defect grounds or failure-to-warn grounds (Twerski, Weinstein, Donaher, & Piehler, 1975).

In a case that an assumption[32] is interpreted as an assumption-used, software engineers observe the possible environments and make hypothesizes about the world according to their understanding and knowledge of the world. If this assumption doesn't match the reality, and leads to malfunction of a system, it is usually recognized as a design defect. For example, on 4th June 1996, a flight of the Ariane 5 launcher ended in a crash. The crash was simply caused by the value of a particular variable received from a sensor exceeded the assumed limit, and this consequently made the computers of the flight cease to work. As stated by Lions, the engineers underestimated the possible environment conditions, and 'it was not analyzed or fully understood which values this particular variable might assume' (Lions, 1996).

In contrast, in a case that an assumption is interpreted as an assumption-needed, an assumption is adopted as a design component that describes the context in which the design solution is reasonable. This interpretation choice grants assumptions the ability to delimit the scope of the solution. This possibility is of substantial practical usefulness for the software engineers facing time and resource limitations. For example, an *intended user assumption* falls exactly into this sense of interpretation: there is a group of target users assumed by the engineers, and it is only for that target group that the software is expected to be guaranteed to work (Schultz et al., 2002). From a legal point of view, as long as the proper disclaimers to the target users are made in a clear manner, the engineers of the software are covered in their legal responsibilities. That is to say that they are not liable for the effects of the software in users outside the assumed target group.

According to the legal issues above, we can derive the importance of making clear these two interpretations of assumptions in software engineering. Without such a distinction, we don't know how to as-

[32] Note that we are considering here assumptions just as propositions, ignoring the mental states of the agents that hold such propositions.

sign responsibility when some undesired consequence is brought about by the software's operation. Moreover, we defend here that a mature software engineering process should make explicit the involved assumptions, analogously to how the requirements, external specification and internal specification are made explicit. In that respect, we defend that the role of assumptions in supporting the *specifications* to satisfy *requirements* should be clearly specified. Only by having access to this information, stakeholders could properly estimate the risks involved in the use of the system at hand, as well as decide whether to adopt this system or not.

6.2 The Meeting Scheduler Case Study

As mentioned in the baseline chapter, instead of dealing with assumptions in the source code level, we emphasize the importance of capturing assumptions in requirements engineering process, to detect errors as early as possible.

Particularly, the requirements engineering process can be further divided into two phases as stated by (Yu, 1997), namely, the early-phase and the late-phase. In the early-phase, software engineers collect requirements from stakeholders. Usually, in this phase, social activities (e.g., such as interviews, surveys, and etc.) are used to gather informal and vague requirements, trying to get the answer to why such a software-driven machine should be implemented. The output of this phase is typically a document in natural language that summarizes all the relevant information collected.

Then, in the late-phase, the initial requirements gathered in the early-phase are used as its input, and further analysis is carried out to check the feasibility and the consistency of the initial requirements. A goal model, such as the one stated by (Yu, 1997), can be adopted in this phase, and within the goal model, the requirements are represented as goals, and could be decomposed/refined into sub-goals and then into tasks. These tasks are operational at the interface between the world and the machine, and we regard these tasks as the interface actions of an external specification.

We agree that it is important to make clear the boundary between the world and machine. However, it is also essential to indicate explicitly the link crossing the boundary, or we will fall into the paradox mentioned at the beginning of this dissertation again. As software can only directly manipulate the virtual variables in a machine, the sense in which the result of this manipulation affects the outside world is neglected by the view of implementation bias.

To solve this paradox, we have to explicitly capture this link between the world and the machine. As the reader can see in Figure 25, a simple meeting scheduler system is adopted as our case study, and we represent it into a goal model. Being different from the literature work (Yu, 1997), in the notation used in this model, we capture not only the requirements and external specification with interface actions, but also capture the internal specification with machine actions. Moreover, the four different kinds of assumptions are explicitly represented in the model in order to link the world and machine together.

We choose the meeting scheduler system as our case study in this paper, because it is well-known in the requirements engineering literature (Yu, 1997), (Silva Souza, 2012), (A van Lamsweerde, Darimont, & Massonet, 1995); and on the other side, it is relatively a simple scenario, and we believe it will make the demonstration easier to understand.

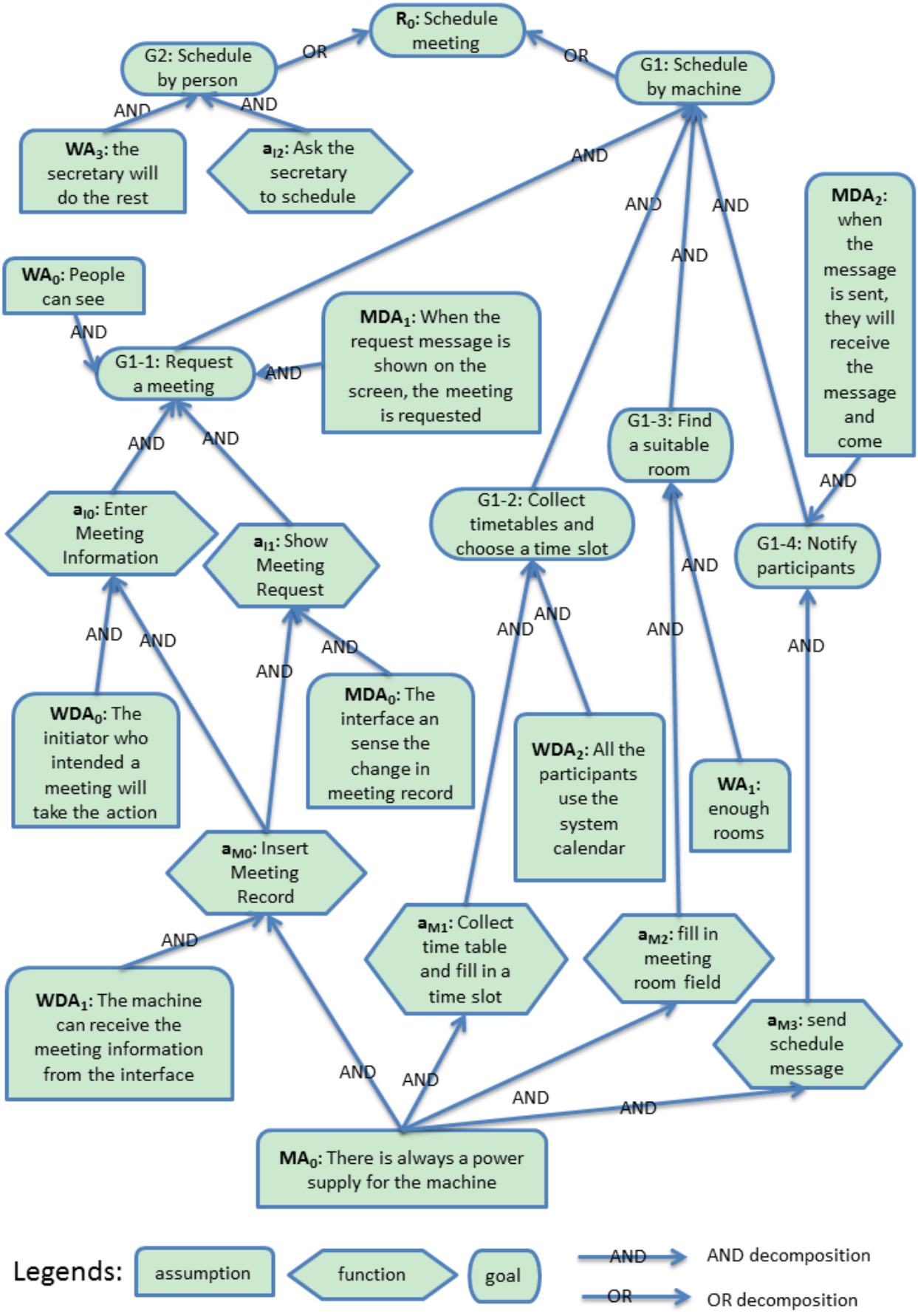


Figure 25. A goal model of meeting scheduler system

Requirements (R): firstly, we start with the requirements. As mentioned in Chapter 5, our single requirement is that 'a meeting shall be scheduled after a meeting initiator intends to schedule one'. This requirement is labeled as R_0 in the goal model. The situation calculus formalization of R_0 has already been shown in Chapter 5, hence, we do not repeat it here. We treat the other expressions in the same way, only the newly proposed ones will be encoded into situation calculus in this sub-chapter.

One should note that the main purpose of this paper is not trying to explain how to translate a goal model into situation calculus expressions, but to demonstrate what kinds of information should be captured in the requirements engineering process, as well as to show the possibility of formalizing them. According to this rationale, we only translate a number of relevant elements from the goal model into situation calculus expressions.

The requirement R_0 is refined into two alternative sub-goals through 'OR decomposition' links: 1) the meeting initiator can do it manually, asking a secretary to arrange a meeting as indicated by the interface action a_{I2}. This choice of solution needs a world assumption WA_3 to ensure the reliability of the secretary who should arrange everything else for the meeting initiator; 2) however, the secretaries might be under work overload, such that they cannot be relied to do this work quickly and correctly. If this is the case, the second choice of adopting a meeting scheduler system might be preferable.

The scenario of using this system is briefly summarized as follows: when a meeting initiator wants to schedule a meeting, she will make a request to the system. For every request, the meeting initiator enters the meeting information into the system through the interface, including a list of intended participants, the title of the meeting, and etc. To schedule a meeting, the system needs to collect timetables from all participants, choose a time slot, and then assign a meeting room for the meeting. Finally, the system must inform all participants in the provided list.

External Specification (S_{ext}): according to the scenario aforementioned, the sub-goal '*G1*: Schedule by machine' could be refined further into four sub-goals, including '*G1-1*: Request a meeting', '*G1-2*: Collect timetables and choose a time slot', '*G1-3*: Find a suitable room', '*G1-4*: Notify participants'. To simplify the work of the initiator as much possible, it is ideal to design such a solution that the only work the initiator would need to do is to make a meeting request to the system with the necessary meeting information. By receiving that request, the system would do everything else for the initiator.

In this case, it seems that the newly proposed system is a complete replacement of the secretary for scheduling meetings. The system provides two interface actions a_{I0} and a_{I1} to implement the sub-goal *G1-1*. Through these actions, the initiator can submit a meeting request to the system and get a request confirmation message from the system. To ensure the fulfillment of this sub-goal through executing these two interface actions, four assumptions are included in the model: WA_0, WDA_0, MDA_0 and MDA_1.

All these elements mentioned here (a_{I0}, a_{I1}, WA_0, WDA_0, MDA_0 and MDA_1) have already been translated into situation calculus in the example that explains the chaining mechanism formed by such actions and assumptions as shown the preceding sub-chapter. Here we only show how they could be asserted in a goal model, and the corresponding translations to situation calculus are not repeated. As another contribution, in the sequel, we explain further how the sub-goals (e.g., *G1-2*, *G1-3*, and *G1-4*) in the goal model could be decomposed into machine actions (e.g., a_{M1}, a_{M2}, and a_{M3}) with the help of different kinds of assumptions.

Internal Specification (S_{int}): the internal specification of the meeting scheduler system contains four machine actions, including a_{M0}, a_{M1}, a_{M2}, and a_{M3}. The first one has already been explained, and we here analyze the last three of these.

$$a_{M1} : Collect\_Timetables\_and\_Fill\_Time\_Slot(meeting)$$
$$PRE : Poss(a_{M1}, s) \leftrightarrow$$
$$\quad meeting\_record\_inserted_M(meeting, s)$$
$$POS : Poss(a_{M1}, s) \rightarrow$$
$$\quad [meeting\_time\_slot\_filled_M(meeting, s')$$
$$\quad \wedge s' = do(a_{M1}, s)]$$

$$WDA_2 : use\_system\_calendar_W(participant)$$

Firstly, the sub-goal *G1-2* is implemented by the machine action a_{M1}. Whenever there is a meeting record inserted into the database, this action a_{M1} will be executed as indicated by its precondition. By executing this action, all the participants' timetables will be automatically collected, and as a calculating result, a suitable time slot will be filled into the meeting record (as indicated in this action's post-condition).

However, this action only works in the situations where all the participants use the system calendars, so we introduce an assumption WDA_2 to constrain the operational situations of this system. This assumption presupposes some interactions between the participants and the machine. In particular, it assumes that the operation of the machine depends on some world phenomena, hence we treat it as a world dependence assumption as indicated in the corresponding situation calculus expression.

$$a_{M2} : Assign\_Meeting\_Room(meeting)$$
$$PRE : Poss(a_{M2}, s) \leftrightarrow$$
$$\quad meeting\_time\_slot\_filled_M(meeting, s)$$
$$POS : Poss(a_{M2}, s) \rightarrow$$
$$\quad [meeting\_room\_filled_M(meeting, s')$$
$$\quad \wedge s' = do(a_{M2}, s)]$$

$$WA_1 : enough\_room_W(meeting)$$

Then, the sub-goal *G1-3* is implemented by the machine action a_{M2}. The post-condition of a_{M1} is used as the pre-condition of a_{M2} such that whenever the time slot of a meeting record is provided, the action a_{M2} is executed. By executing a_{M2}, a new situation will be reached, in which a room number is assigned to the meeting record as shown in its post-condition.

As mentioned several times in this paper, to simplify the work of the engineers in this modeling case, the system only deals with the situations in which there are enough rooms, and it will not work properly in a context in which this is not the case. Thus, a world assumption WA_1 is introduced such that for every meeting that needs to be scheduled, it is assume that there are always rooms available.

$$a_{M3} : Notify\_Particiapnts(meeting)$$
$$PRE : Poss(a_{M3}, s) \leftrightarrow$$
$$meeting\_record\_inserted_M(meeting, s)$$
$$\wedge meeting\_time\_slot\_filled_M(meeting, s)$$
$$\wedge meeting\_room\_filled_M(meeting, s)$$
$$POS : Poss(a_{M3}, s) \rightarrow$$
$$[notification\_messages\_sent_M(meeting, s')$$
$$\wedge s' = do(a_{M3}, s)]$$

$$MDA_2 : notification\_messages\_sent_M(meeting, s)$$
$$\leftrightarrow meeting\_scheduled_W(meeting, s)$$

Finally, the sub-goal *G1-4* is implemented by the machine action a_{M3}. Whenever the meeting record is inserted, filled with a time slot and a room number, the action a_{M3} is executed as indicated by its pre-condition. By executing it, the meeting schedule notification message is sent to all the participants as shown in its post-condition.

However, sending messages does not equate to confirming the schedule with the participants, i.e., the interface action by itself is not directly equivalent to the social action of creating a meeting (a social object) involving all those participants. Once more this gap is filled by a machine dependence assumption MDA_2, which links the *message sending* and the *schedule confirming*. In other words, according to this assumption, the solution is simplified, and whenever the messages are sent, we assume the participants will receive them, confirm them, and at the same time the meeting is also scheduled. Or put it in yet different terms, the message sending *counts as* a schedule being confirmed in this context.

In summary, through the meeting scheduler case study, we have demonstrated how the analysis of assumptions can be introduced in the requirement engineering process instead of only in the code writing process. Additionally, four different kinds of assumptions are represented in a goal model, linking the world and machine together. In this model, we also show how these assumptions can be formally represented.

Chapter 7

Conclusions and Future Work

As we said at the beginning of the dissertation, software has become an indispensable part of our lives, and the degree people relay on software applications is still getting deeper and deeper every day. However, the environments of these software applications are changing continuously, and the stakeholders' requirements are usually unstable and unpredictable. To survive in such a setting, the software applications have to be changed rapidly accordingly, and as a result, the work of maintaining these changes in software applications consumes a large amount of human and financial resources.

We believe that one of the reasons for this unhappy situation is knowledge missing about the software during its life span, which is in turns caused by a lack of consensus on what exactly software is. To remedy the aforementioned situation, we proposed our answer to this question from several basic perspectives, including what are software's defining traits, its fundamental properties and constituent concepts and how do these relate to each other. As a summary, the contributions of this dissertation are presented in the sequel, followed by some possible future works:

In this dissertation, we dive into the ontological nature of software, recognizing it as an abstract information artifact. To support this proposal the first main contribution of the dissertation is demonstrated from three dimensions: (1) We distinguish software (non-physical object) from hardware (physical object), and demonstrate the idea that the rapid changing speed of software is supported by the easy changeability of its medium hardware; (2) Furthermore, we discuss about the artifactual nature of software, addressing the erroneous notion, software is just code, presents both in the ontology of software literature and in the software maintenance tools; (3)At last, we recognize software as an information artifact, and this approach ensures that software inherits all the properties of an information artifact, and the study and research could be reused for software then. For instance, an English story is an information artifact, the information part is the story content, and the syntax part is the set of English words. Accordingly, software is interpreted as an information artifact that its information part is the set of instructions, and the syntax part is the source code.

Then, the second main contribution of this dissertation presented a first attempt to analyze the ontological nature of software artifacts in the light of the Jackson and Zave's model, considered nowadays as a foundation for requirements engineering. Such a model has helped us to provide an answer to the question concerning the identity criteria of software artifacts raised by Irmak. We proposed three different kinds of software artifacts, exhibiting different essential properties depending on stakeholders' intentions to produce effects in different parts of complex software-driven sociotechnical systems (namely, the WM Framework). Such different essential properties of the software artifacts are summarized as shown in Table 2.

In addition, we captured the fourth kind of artifact (*Licensed Software Product*) reflecting the social nature of software products, whose essential properties are based on the mutual commitments between vendors and customers. In the present software market, software products do not come alone, since what

companies sell are not just software products: in the vast majority of cases, a purchase contract for a software product includes a number of rights and duties on both parties, including the right to download updates for a certain period of time, the prohibition to give copies away, and so on. Before the contract is signed we have another software entity emerging: a *Licensable Software Product*; and after the contract is signed, we have a *Licensed Software Product*.

Table 2. Essential Properties of different kinds of software artifacts

Software Artifact	**Essential Properties**
Software Product	Requirements
Software System	External Specification
Software Program	Internal Specification

The third main contribution of the dissertation solves the paradox that 'in what sense a software-driven machine can affect the outside social world without any physical means', and it is presented in three-fold: 1) we propose a preliminary ontology of assumptions classifying four kinds of assumptions. Based on this classification, we elaborate on the role of assumptions in explaining how social facts can be affected by the manipulation of symbolic structures in a machine; 2) we clarify the concept of 'assumption' adopted in software engineering literature, and emphasize the importance of clarifying the interpretations of the assumptions as either *assumptions-used*, or as *assumptions-needed*; 3) as a methodological contribution and, by employing a meeting scheduler case study, we demonstrate how assumptions can be explicitly and systematically elicited and represented as part of the requirements engineering process.

As this case study demonstrates, requirements engineering, in particular, and software engineering in general can benefit from the awareness of the existence of these four kinds of assumptions. Assumptions are of fundamental importance to software engineering. Therefore, we strongly suggest that more efforts should be made to develop a more clear understanding of what assumptions are. So, on the theoretical side, we intend to publish a dedicated future paper exploring the ontological nature of assumptions as well as systematizing the nature of the relations to other elements of our software ontology (Wang et al., 2014a, 2014b).

The current work of this dissertation is mainly presented through natural language expressions, although the software related concepts are thoroughly analyzed and clarified, a formalization of our work as a future work is still a promising research direction, because whenever it is formalized, it could be accepted and reused by computer systems much easier, and consequently a lot of automatic approaches could be proposed. Furthermore, this ontology of software could be extended into an ontology of software changes, which could be used as a foundation to solve the problems caused by software aging.

For instance, on the basis of our analysis, a refined terminology for different kinds of software change may be proposed: 1) *refactoring* refers to the creation of new codes, keeping the identity of the software program; 2) *re-engineering* refers to the creation of new software programs, keeping the identity of the software system; 3) *software evolution* refers to the creation of new software systems, keeping the identity of the software product.

Based on that, a refined versioning methodology and better software versioning control tools dealing with revisions and variants could be developed. As noted several times, traditional tools only focus on

code changes. According to our work, software should be consistently expressed and tracked in multiple abstraction layers. Traditional version codes are usually decided on the basis of the significance of changes between releases, but the decisions of the significances are entirely arbitrary and up to the author. On the basis of our approach, versioning numbers can be established in a rigorous standard way.

By integrating the software configuration management systems and assumptions management systems together, the work proposed here opens up the possibility of developing a next generation of *software management system*s. In these systems, the management of assumptions could be embedded into the requirements engineering process. As a consequence, along with monitoring the changes of software during its life span, errors involving assumptions could be identified and addressed in a much earlier stage of a software engineering process. Moreover, these systems could also support managers in decision-making activities based on such recorded software history.

Bibliography

Abran, A., & Moore, J. W. (2004). *Guide to the software engineering body of knowledge*. IEEE Computer Society. Retrieved from http://books.google.it/books?id=IKZQAAAAMAAJ

Altmanninger, K. (2008). Models in Conflict – Towards a Semantically Enhanced Version Control System for Models. In H. Giese (Ed.), *Models in Software Engineering SE - 31* (Vol. 5002, pp. 293–304). Springer Berlin Heidelberg. http://doi.org/10.1007/978-3-540-69073-3_31

Anquetil, N., de Oliveira, K. M., de Sousa, K. D., & Batista Dias, M. G. (2007). Software maintenance seen as a knowledge management issue. *Information and Software Technology*, *49*(5), 515–529. Retrieved from http://www.sciencedirect.com/science/article/pii/S0950584906001029

Ashburner, M., Ball, C. A., Blake, J. A., Botstein, D., Butler, H., Cherry, J. M., … Eppig, J. T. (2000). Gene Ontology: tool for the unification of biology. *Nature Genetics*, *25*(1), 25–29.

Baker, L. R. (2004). The ontology of artifacts. *Philosophical Explorations*, *7*(2), 99–111. http://doi.org/10.1080/13869790410001694462

Belady, L. A., & Lehman, M. M. (1976). A Model of Large Program Development. *IBM Syst. J.*, *15*(3), 225–252. http://doi.org/10.1147/sj.153.0225

Bennett, K. H., & Rajlich, V. T. (2000). Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering* (pp. 73–87). New York, NY, USA: ACM. http://doi.org/10.1145/336512.336534

Berners-Lee, T., Hendler, J., & Lassila, O. (2001). The semantic web. *Scientific American*, *284*(5), 28–37.

Beyer, D., & Hassan, A. E. (2006). However, version control systems (VCS) contain valuable historical information about a project, and mining the VCS repository may reveal interesting events in the development and maintenance of long-lived projects. *Reverse Engineering, 2006. WCRE '06. 13th Working Conference on*. http://doi.org/10.1109/WCRE.2006.14

Bin-bin, C., Hai, Z., Xiaoping, Z., & Hesheng, L. (2012). Bats' acoustic detection system and echolocation bionics. In *Radar Conference (RADAR), 2012 IEEE* (pp. 984–988). http://doi.org/10.1109/RADAR.2012.6212280

Borst, W. N. (1997). *Construction of engineering ontologies for knowledge sharing and reuse*. Universiteit Twente.

Bourque, P., & Fairley, R. E. (Eds.). (2014). *Guide to the Software Engineering Body of Knowledge Version 3.0. A Project of the IEEE Computer Society, 2014. SWEBOK* (3rd ed.). IEEE Computer Society Press.

Brown, D. (2006). Assumptions in Design and Design Rationale. In *DCC'06 Workshop on Design Rationale: Problems and Progress*. Eindhoven, The Netherlands. Retrieved from http://www.users.miamioh.edu/burgeje/DRWorkshopDCC06.html

Buchholz, W. (2000). Comments, Queries, and Debates. *IEEE Annals of the History of Computing*, *22*(4), 69–71. http://doi.org/http://doi.ieeecomputersociety.org/10.1109/MAHC.2000.887996

Buckley, J., Mens, T., Zenger, M., Rashid, A., & Kniesel, G. (2005). Towards a Taxonomy of Software Change: Research Articles. *J. Softw. Maint. Evol.*, *17*(5), 309–332. http://doi.org/10.1002/smr.v17:5

Canfora, G., Dalcher, D., Raffo, D., Basili, V. R., Fernández-Ramil, J., Rajlich, V., … Perry, D. E. (2011). In memory of Manny Lehman, "Father of Software Evolution." *Journal of Software Maintenance and Evolution: Research and Practice*, *23*(3), 137–144. Retrieved from http://dx.doi.org/10.1002/smr.537

Chapin, N., Hale, J. E., Kham, K. M., Ramil, J. F., & Tan, W.-G. (2001). Types of software

evolution and software maintenance. *Journal of Software Maintenance*, *13*(1), 3–30. Retrieved from http://dl.acm.org/citation.cfm?id=371697.371701

Chrissis, M. B., Konrad, M., & Shrum, S. (2011). *CMMI for Development: Guidelines for Process Integration and Product Improvement*. Pearson Education. Retrieved from http://books.google.it/books?id=dNyv0h91BJIC

Colburn, T. R. (1999). Software, Abstraction, and Ontology. *The Monist*, *82*(1), 3–19. Retrieved from http://www.jstor.org/stable/27903620

D'Ambros, M., & Lanza, M. (2009). Visual software evolution reconstruction. *Journal of Software Maintenance and Evolution: Research and Practice*, *21*(3), 217–232.

Dart, S. (1991). Concepts in Configuration Management Systems. In *Proceedings of the 3rd International Workshop on Software Configuration Management* (pp. 1–18). New York, NY, USA: ACM. http://doi.org/10.1145/111062.111063

Duncan, W. (2011). Using ontological dependence to distinguish between hardware and software. *AISB 2011: Computing and Philosophy*, 89–97. Retrieved from http://www.scopus.com/inward/record.url?eid=2-s2.0-84863896604&partnerID=40&md5=98eb6004c59ab63162fd35e9078fe0ba

Eden, A. H., & Turner, R. (2007). Problems in the ontology of computer programs. *Appl. Ontol.*, *2*(1), 13–36. Retrieved from http://dl.acm.org/citation.cfm?id=1412396.1412397

Ennis, R. (1982). Identifying implicit assumptions. *Synthese*, *51*(1), 61–86. http://doi.org/10.1007/BF00413849

Estublier, J., Leblang, D. B., van der Hoek, A., Conradi, R., Clemm, G., Tichy, W. F., & Weber, D. W. (2005). Impact of software engineering research on the practice of software configuration management. *ACM Trans. Softw. Eng. Methodol.*, *14*(4), 383–430.

Faulkner, P., & Runde, J. (2011). The social, the material, and the ontology of non-material technological objects. In *European Group for Organizational Studies (EGOS) Colloquium, Gothenburg.*

Ferrario, R., & Oltramari, A. (2005). Towards a Computational Ontology of Mind. In *Aerospace Conference, 2005 IEEE* (pp. 1–9). http://doi.org/10.1109/AERO.2005.1559636

Fischer, M., Pinzger, M., & Gall, H. (2003). Populating a Release History Database from version control and bug tracking systems. *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*. http://doi.org/10.1109/ICSM.2003.1235403

Fortier, J.-Y., & Kassel, G. (2004). Managing knowledge at the information level: an ontological approach. In *Proceedings of the ECAI'2004 Workshop on Knowledge Management and Organizational Memories* (pp. 39–45).

Franken, D., Karakus, A., & Michel, J. G. M. (2010). *John R. Searle: Thinking About the Real World*. De Gruyter. Retrieved from https://books.google.it/books?id=aGWP9HL68_8C

Gangemi, A., Borgo, S., Catenacci, C., & Lehmann, J. (2004). Task Taxonomies for Knowledge Content D07. Metokis Project. Retrieved from http://metokis.salzburgresearch.at/files/deliverables/metokis_d07_task_taxonomies_final.pdf

Gašević, D., Kaviani, N., & Milanović, M. (2009). Ontologies and Software Engineering. In S. Staab & D. Rudi Studer (Eds.), *Handbook on Ontologies* (pp. 593–615). Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/978-3-540-92673-3_27

Georgakopoulos, D., Hornick, M., & Sheth, A. (1995). An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, *3*(2), 119–153. http://doi.org/10.1007/BF01277643

Ghezzi, G., Wursch, M., Giger, E., & Gall, H. C. (2012). An architectural blueprint for a pluggable version control system for software (evolution) analysis. In *Developing Tools as Plug-ins (TOPI), 2012 2nd Workshop on* (pp. 13–18). http://doi.org/10.1109/TOPI.2012.6229803

G îrba, T., & Ducasse, S. (2006). Modeling history to analyze software evolution. *Journal of*

Software Maintenance and Evolution: Research and Practice, *18*(3), 207–236. http://doi.org/10.1002/smr.325
Godfrey, M. W., & German, D. M. (2008). The past, present, and future of software evolution. *Frontiers of Software Maintenance, 2008. FoSM 2008.* http://doi.org/10.1109/FOSM.2008.4659256
Gruber, T. R. (1993). A Translation Approach to Portable Ontology Specifications. *Knowl. Acquis.*, *5*(2), 199–220. http://doi.org/10.1006/knac.1993.1008
Gruber, T. R. (1995). Toward principles for the design of ontologies used for knowledge sharing. *Int. J. Hum.-Comput. Stud.*, *43*(5-6), 907–928. http://doi.org/10.1006/ijhc.1995.1081
Guarino, N. (1995). Formal ontology, conceptual analysis and knowledge representation. *International Journal of Human-Computer Studies*, *43*(5-6), 625–640. http://doi.org/10.1006/ijhc.1995.1066
Guarino, N. (1997). Semantic matching: Formal ontological distinctions for information organization, extraction, and integration. In M. Pazienza (Ed.), (Vol. 1299, pp. 139–170). Springer Berlin / Heidelberg. http://doi.org/10.1007/3-540-63438-X_8
Guarino, N. (1998). Formal Ontology and Information Systems. Retrieved from citeulike-article-id:4148125
Guarino, N. (2009). The Ontological Level: Revisiting 30 Years of Knowledge Representation. In A. T. Borgida, V. K. Chaudhri, P. Giorgini, & E. S. Yu (Eds.), *Conceptual Modeling: Foundations and Applications* (pp. 52–67). Berlin, Heidelberg: Springer-Verlag. http://doi.org/10.1007/978-3-642-02463-4_4
Guarino, N. (2014). Artefact Kinds: Ontology and the Human-Made World. In M. Franssen, P. Kroes, A. C. T. Reydon, & E. P. Vermaas (Eds.), (pp. 191–206). Cham: Springer International Publishing. http://doi.org/10.1007/978-3-319-00801-1_11
Guarino, N., & Giaretta, P. (1995). Ontologies and knowledge bases: Towards a terminological clarification. In *Towards very Large Knowledge bases: Knowledge Building and Knowledge sharing* (pp. 25–32). IOS Press.
Guarino, N., Oberle, D., & Staab, S. (2009). What Is an Ontology? In S. Staab & R. Studer (Eds.), *Handbook on Ontologies (International Handbooks on Information Systems)* (2nd ed., pp. 1–17). Berlin, Heidelberg: Springer Berlin Heidelberg. http://doi.org/10.1007/978-3-540-92673-3
Guarino, N., & Welty, C. A. (2009). An overview of OntoClean. In *Handbook on ontologies* (pp. 201–220). Springer.
Gunter, C. A., Jackson, M., & Zave, P. (2000). A reference model for requirements and specifications. *Software, IEEE*, *17*, 37–43. http://doi.org/10.1109/52.896248
Hailperin, M., Kaiser, B., & Knight, K. (1999). *Concrete Abstractions: An Introduction to Computer Science Using Scheme* (1st ed.). Boston, MA, USA: PWS Publishing Co.
IEEE Standard for Software Maintenance. (1998). *IEEE Std 1219-1998*. http://doi.org/10.1109/IEEESTD.1998.88278
IEEE Standard Glossary of Software Engineering Terminology. (1990). *IEEE Std 610.12-1990*, 1. http://doi.org/10.1109/IEEESTD.1990.101064
Irmak, N. (2013). Software is an Abstract Artifact. *Grazer Philosophische Studien*, *86*(1), 55–72. Retrieved from http://www.ingentaconnect.com/content/rodopi/gps/2013/00000086/00000001/art00004
Jackson, M. (2000). The Real World. In J. Davies, B. Roscoe, & J. Woodcock (Eds.), *Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare* (pp. 157–173). Palgrave.
Jackson, M. (2007). Specialising in Software Engineering. In *Software Engineering Conference, 2007. APSEC 2007. 14th Asia-Pacific* (pp. 3–10). http://doi.org/10.1109/ASPEC.2007.9
Jackson, M., & Zave, P. (1995). Deriving specifications from requirements: an example. In *Proceedings of the 17th international conference on Software engineering* (pp. 15–24).

New York, NY, USA: ACM. http://doi.org/10.1145/225014.225016

Jarzabek, S. (2007). *Effective Software Maintenance and Evolution: A Reuse-Based Approach*. Taylor & Francis. Retrieved from http://books.google.it/books?id=F6eOs1JMBdgC

Jureta, I. J., Mylopoulos, J., & Faulkner, S. (2009). A core ontology for requirements. *Applied Ontology*, *4*(3), 169–244. http://doi.org/10.3233/AO-2009-0069

Kassel, G. (2010). A formal ontology of artefacts. *Appl. Ontol.*, *5*(3-4), 223–246. Retrieved from http://dl.acm.org/citation.cfm?id=1889917.1889918

Kitchenham, B. A., Travassos, G. H., von Mayrhauser, A., Niessink, F., Schneidewind, N. F., Singer, J., … Yang, H. (1999). Towards an Ontology of software maintenance. *Journal of Software Maintenance*, *11*(6), 365–389. http://doi.org/10.1002/(SICI)1096-908X(199911/12)11:6<365::AID-SMR200>3.0.CO;2-W

Kleine, M., Hirschfeld, R., & Bracha, G. (2012). *An Abstraction for Version Control Systems*. Univ.-Verlag. Retrieved from http://books.google.it/books?id=_4-Qdmlf1xIC

Kontogiannis, K. (2010). Maintenance: Techniques. In *Encyclopedia of Software Engineering* (pp. 482–512).

Lamsweerde, A. Van. (2009). From Worlds to Machines. In *A Tribute to Michael Jackson*. Lulu Press.

Lando, P., Lapujade, A., Kassel, G., & Fürst, F. (2007). Towards a General Ontology of Computer Programs. In *ICSOFT (PL/DPS/KE/MUSE)* (pp. 163–170).

Lando, P., Lapujade, A., Kassel, G., & Fürst, F. (2009). An Ontological Investigation in the Field of Computer Programs. In J. Filipe, B. Shishkov, M. Helfert, & L. Maciaszek (Eds.), *Software and Data Technologies SE - 28* (Vol. 22, pp. 371–383). Springer Berlin Heidelberg. http://doi.org/10.1007/978-3-540-88655-6_28

Lehman, M., & Fernandez-Ramil, J. (2006). Software Evolution. In N. H. Madhavji, J. Fernandez-Ramil, & D. Perry (Eds.), *Software Evolution and Feedback: Theory and Practice* (1st ed., pp. 7–40). Wiley.

Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*. http://doi.org/10.1109/PROC.1980.11805

Lehman, M. M. (1996). Laws of Software Evolution Revisited. In *Proceedings of the 5th European Workshop on Software Process Technology* (pp. 108–124). London, UK, UK: Springer-Verlag. Retrieved from http://dl.acm.org/citation.cfm?id=646195.681473

Lewis, G., Mahatham, T., & Wrage, L. (2004). *Assumptions Management in Software Development*. Pittsburgh, PA. Retrieved from http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=6941

Lientz, B. P., & Swanson, E. B. (1980). *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley Pub (Sd). Retrieved from citeulike-article-id:3944634

Lions, J.-L. (1996). Flight 501 failure. *Report by the Inquiry Board*.

Lungu, M. (2008). Towards reverse engineering software ecosystems. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on* (pp. 428–431). http://doi.org/10.1109/ICSM.2008.4658096

Maass, W., Goyal, S., & Behrendt, W. (2004). Knowledge Content Objects and a Knowledge Content Carrier Infrastructure for ambient knowledge and media aware content systems. *Knowledge-Based Media Analysis*, 449–456.

Mamun, M. A. Al, & Hansson, J. (2011). Review and Challenges of Assumptions in Software Development. In *Second Analytic Virtual Integration of Cyber-Physical Systems Workshop (AVICPS)*.

Martin, J. (2010). *Introduction to Languages and the Theory of Computation*. McGraw-Hill Companies,Incorporated. Retrieved from http://books.google.it/books?id=arluQAAACAAJ

Masolo, C., Borgo, S., Gangemi, A., Guarino, N., & Oltramari, A. (2003a). *WonderWeb Deliverable D18. Communities*. Trento. Retrieved from

http://wonderweb.semanticweb.org/deliverables/documents/D18.pdf
Masolo, C., Borgo, S., Gangemi, A., Guarino, N., & Oltramari, A. (2003b). *WonderWeb-D17: Library of Foundational Ontologies*. Padova.
McCarthy, J., & Laboratory, S. U. C. S. D. A. I. (1963). *Situations, actions, and causal laws*. Comtex Scientific. Retrieved from http://books.google.it/books?id=iF8iGwAACAAJ
Mens, T., & Grosjean, P. (2015). The Ecology of Software Ecosystems. *Computer*, *48*(10), 85–87. http://doi.org/10.1109/MC.2015.298
Mens, T., Gueheneuc, Y.-G., Fernandez-Ramil, J., & D'Hondt, M. (2010). Guest Editors' Introduction: Software Evolution. *Software, IEEE*. http://doi.org/10.1109/MS.2010.100
Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., & Jazayeri, M. (2005). Challenges in Software Evolution. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution* (pp. 13–22). Washington, DC, USA: IEEE Computer Society. http://doi.org/10.1109/IWPSE.2005.7
Messerschmitt, D. G., & Szyperski, C. (2003). *Software Ecosystem: Understanding an Indispensable Technology and Industry*. Cambridge, MA, USA: MIT Press.
Mezrich, B. (2010). *The Accidental Billionaires: The Founding of Facebook : a Tale of Sex, Money, Genius and Betrayal*. Anchor Books. Retrieved from http://books.google.it/books?id=1cw8AJTBbZMC
Moor, J. H. (1978). Three myths of computer science. *The British Journal for the Philosophy of Science*, *29*, 213–222. http://doi.org/10.1093/bjps/29.3.213
Mylopoulos, J. (1992). Conceptual Modelling and Telos. In P. Loucopoulos & R. Zicari (Eds.), *Conceptual modeling, database, and CASE* (pp. 49–68). Wiley.
Nardi, J. C., De Almeida Falbo, R., Almeida, J. P. A., Guizzardi, G., Ferreira Pires, L., van Sinderen, M. J., & Guarino, N. (2013). Towards a Commitment-Based Reference Ontology for Services. In *Enterprise Distributed Object Computing Conference (EDOC), 2013 17th IEEE International* (pp. 175–184). http://doi.org/10.1109/EDOC.2013.28
Nkwake, A. (2013). What are Assumptions? In *Working with Assumptions in International Development Program Evaluation SE - 6* (pp. 81–91). Springer New York. http://doi.org/10.1007/978-1-4614-4797-9_6
Nuseibeh, B., & Easterbrook, S. (2000). Requirements engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering* (pp. 35–46). New York, NY, USA: ACM. http://doi.org/10.1145/336512.336523
Oberle, D. (2006). *Semantic Management of Middleware* (Vol. 1). New York: Springer. http://doi.org/10.1007/0-387-27631-9
Oberle, D., Grimm, S., & Staab, S. (2009). An Ontology for Software. In S. Staab & D. Rudi Studer (Eds.), (pp. 383–402). Springer Berlin Heidelberg. http://doi.org/10.1007/978-3-540-92673-3_17
Ogden, C. K., Richards, I. A., Malinowski, B., Constable, J., & Crookshank, F. G. (2001). *The Meaning of Meaning: A Study of the Influence of Language Upon Thought and of the Science of Symbolism*. Routledge. Retrieved from http://books.google.it/books?id=pl0qAQAAIAAJ
Omori, T., & Maruyama, K. (2008). A Change-aware Development Environment by Recording Editing Operations of Source Code. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories* (pp. 31–34). New York, NY, USA: ACM. http://doi.org/10.1145/1370750.1370758
Osterweil, L. J. (2008). What is software? *Autom. Softw. Eng.*, *15*(3-4), 261–273.
Parnas, D. L. (1994). Software aging. In *Proceedings of the 16th international conference on Software engineering* (pp. 279–287). Los Alamitos, CA, USA: IEEE Computer Society Press. Retrieved from http://dl.acm.org/citation.cfm?id=257734.257788
Parnas, D. L. (2011). Precise Documentation: The Key to Better Software. In S. Nanz (Ed.), (pp. 125–148). Springer Berlin Heidelberg. http://doi.org/10.1007/978-3-642-15187-3_8

Pfleeger, S. L., & Atlee, J. M. (2009). *Software Engineering: Theory and Practice*. Prentice Hall. Retrieved from http://books.google.it/books?id=7zbSZ54JG1wC

Raskin, R. G., & Pan, M. J. (2005). Knowledge representation in the semantic web for Earth and environmental terminology (SWEET). *Computers & Geosciences*, *31*(9), 1119–1125. http://doi.org/10.1016/j.cageo.2004.12.004

Robbes, R., & Lanza, M. (2008). SpyWare: A Change-aware Development Toolset. In *Proceedings of the 30th International Conference on Software Engineering* (pp. 847–850). New York, NY, USA: ACM. http://doi.org/10.1145/1368088.1368219

RUIZ, F., VIZCA ÍNO, A., PIATTINI, M., & GARC ÍA, F. (2004). An ontology for the management of software maintenance projects. *International Journal of Software Engineering and Knowledge Engineering*, *14*(03), 323–349. http://doi.org/10.1142/S0218194004001646

Ryan-Collins, J., Greenham, T., Werner, R., & Jackson, A. (2014). *Where Does Money Come From?: A Guide to the UK Monetary and Banking System*. New Economics Foundation. Retrieved from https://books.google.it/books?id=j0usoAEACAAJ

Schultz, M., Conshohocken, W., Hill, C., Vegas, L., Angeles, L., Diego, S., & Francisco, S. (2002). *Defenses in a Product Liability Claim*.

Silva Souza, V. E. (2012). *Requirements-based Software System Adaptation*. University of Trento. Retrieved from http://www.inf.ufes.br/~vitorsouza/en/academia/phd-thesis/

Smith, B. (2004). Beyond concepts: Ontology as reality representation. In A. Varzi & L. Vieu (Eds.), *Proceedings of FOIS 2004* (pp. 73–84). Turin: IOS Press.

Smith, B., Malyuta, T., Rudnicki, R., Mandrick, W., Salmen, D., Morosoff, P., … Parent, K. (2013). IAO-Intel: An Ontology of Information Artifacts in the Intelligence Domain. In K. B. Laskey, I. Emmons, & P. C. G. da Costa (Eds.), *STIDS* (Vol. 1097, pp. 33–40). CEUR-WS.org.

Studer, R., Benjamins, V. R., & Fensel, D. (1998). Knowledge engineering: principles and methods. *Data & Knowledge Engineering*, *25*(1), 161–197.

Suber, P. (1988). What is software? *The Journal of Speculative Philosophy*, *2*(2), 89–119.

Swanson, E. B. (1976). The dimensions of maintenance. In *Proceedings of the 2nd international conference on Software engineering* (pp. 492–497). Los Alamitos, CA, USA: IEEE Computer Society Press. Retrieved from http://dl.acm.org/citation.cfm?id=800253.807723

Tappolet, J., Kiefer, C., & Bernstein, A. (2010). Semantic web enabled software analysis. *Web Semant.*, *8*(2-3), 225–240. http://doi.org/10.1016/j.websem.2010.04.009

Tukey, J. W. (1958). The Teaching of Concrete Mathematics. *The American Mathematical Monthly*, *65*(1), 1–9. http://doi.org/10.2307/2310294

Tun, T., Lutz, R., Nakayama, B., Yu, Y., Mathur, D., & Nuseibeh, B. (2015). The role of environmental assumptions in failures of DNA nano systems. In *Proceedings of Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS 2015) co-located with ICSE 2015*. Florence, Italy.

Turner, R. (2013). The Philosophy of Computer Science. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Fall 2013).

Twerski, A. D., Weinstein, A. S., Donaher, W. A., & Piehler, H. R. (1975). Use and Abuse of Warnings in Products Liability-Design Defect Litigation Comes of Age. *Cornell L. Rev.*, *61*, 495.

van Lamsweerde, A., Darimont, R., & Massonet, P. (1995). Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt. In *Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on* (pp. 194–203). http://doi.org/10.1109/ISRE.1995.512561

Wallich, P. (1997). Cracking the U.S. Code. *Scientific American*, *276*(42). http://doi.org/10.1038/scientificamerican0497-42

Wang, X., Guarino, N., Guizzardi, G., & Mylopoulos, J. (2014a). Software as a Social Artifact:

A Management and Evolution Perspective. In E. Yu, G. Dobbie, M. Jarke, & S. Purao (Eds.), *33rd International Conference on Conceptual Modeling* (Vol. 8824, pp. 321–334). Atlanta: Springer International Publishing. http://doi.org/10.1007/978-3-319-12206-9_27

Wang, X., Guarino, N., Guizzardi, G., & Mylopoulos, J. (2014b). Towards an Ontology of Software: a Requirements Engineering Perspective. In O. K. Pawel Garbacz (Ed.), *8th International Conference on Formal Ontology in Information Systems* (pp. 317–329). Rio de Janeiro: IOS Press. http://doi.org/10.3233/978-1-61499-438-1-317

Wloka, J., Ryder, B. G., & Tip, F. (2009). JUnitMX - A Change-aware Unit Testing Tool. In *Proceedings of the 31st International Conference on Software Engineering* (pp. 567–570). Washington, DC, USA: IEEE Computer Society. http://doi.org/10.1109/ICSE.2009.5070557

Yu, E. S. K. (1997). Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. In *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering* (p. 226–). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dl.acm.org/citation.cfm?id=827255.827807

Zave, P., & Jackson, M. (1997). Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.*, *6*(1), 1–30. http://doi.org/10.1145/237432.237434