1
Introduction

As the software development field has become more complex, agent-oriented software engineering (AOSE) [81][28] has emerged as a research area that has increased in importance over the past decade. An agent is a software entity situated in a particular environment that is capable of flexible and autonomous action with some level of initiative and reactivity [28][30][79]. Agents exhibit varying levels of the key attributes of learning, cooperation and mobility. Agents have partial control over their environments, perceive stimulus and receive inputs, and immediately respond by fulfilling their objectives. Autonomy refers to the capacity of agents to make decisions and pursue objectives without external input. This means that agents initiate actions and decisions without specific commands, based on present objectives and changes in their environmental parameters. To pursue their objectives, agents collaborate via communication protocols to accomplish complicated tasks, access information or services they do not have, or manage the dependencies that take place from being situated in a common environment.

In most cases, a given agent environment contains a number of agents whose actions mutually affect one another [38]. This interdependence arises because the dissimilar agents, with their own aims and objectives, must operate in a common environment that has finite resources and capabilities. Depending on the agents' environment, several different types of social interaction occur dynamically between the agents, including cooperation to achieve common goals, coordination of their actions, and negotiation to resolve conflicts. Some agents are even capable of learning from past experience and adapting their behavior in given situations. Agents process information and can act as assistant to the user rather than a tool by learning from interaction, proactively anticipating the user's needs and gradually making decisions on behalf of the user, for instance protecting them from excess information or excessive requests. A multi-agent system (MAS) is composed of individual agents, the interactions and social

relations among agents, and their mutual dependencies. Furthermore, a multi-agent application engineering process is characterized by the construction of applications based on reusable software artifacts from agent-oriented software [47].

Agents also helps on the development of complex and distributed systems, based on agent abstractions [79][30]. The main idea is to adopt human-inspired abstractions to model a complex, typically distributed, domain [80]. Using agent-oriented abstractions, complex problems are decomposed into autonomous, pro-active and reactive agents with social ability to manage their complexity [29]. An agent-based conceptual framework makes partitioning the problem space of agent-oriented decompositions more effective for modeling complex systems [28].

AOSE research contains the promise to build scalable, robust and high quality software systems. Benefits that could be garnered from adopting an agent-oriented development approach include the support of analyses, the design and implementation of software applications and the development of a range of complex and large-scale distributed software systems of great complexity [29]. Agent-centered frameworks for software development have already been developed and applied in a wide variety of application domains, e.g., electronic commerce [39], telecommunications network management [73], air traffic control [37][80], data mining [66], smart databases [55], digital libraries [46], information retrieval and management [80], education [13], automated personal digital assistants [80], and scheduling diary management [55]. More generally, internet applications that involve the development of personalized, cooperative, proactive tools for information gathering and management tend to be based on agent frameworks. Since agents maintain a description of their own processing state and the state of the world around them, agents are also ideally suited to automation applications, such as process and workflow automation, and robotics.

Currently, several solutions for engineering agent-based software systems are available. They aim at addressing complex problems intrinsic to agent-based systems, such as distributed communication, thread management, and coordination. Common examples of solutions in this context are modeling languages, e.g., AUML [3]; standardized protocol languages, such as those specified by FIPA [16]; development platforms, for example JADE [6]; and conceptual frameworks such as TAO [71]. In addition, several architectures have

been developed that propose particular methodologies for building agent-based systems, dealing with many different types of problem domain, e.g., agents provided with cognitive abilities are often specified in terms of high-level concepts, such as those part of the belief-desire-intention (BDI) paradigm [65].

Despite the fact that AOSE is a well-recognized area and there is a set of methodologies, techniques and tools for engineering agent-based software systems, there are still advances to be made. Developing an agent-based system from the ground up is a difficult, expensive, and lengthy software-engineering activity due to the various kinds of expertise necessary. In developing a multi-agent system, it is necessary to define multiple agents that communicate and cooperate among themselves and to group decision making as well as competitive behavior. The benefits of applying software-engineering principles to guide solving problems, such as maintenance-oriented development environments and software reuse to agent development, have not yet completely migrated to the agent-development community. There is a major technical obstruction to the widespread adoption of multi-agent technology which has been discussed [40][11]; in order to build development environments for the construction of agent-oriented software systems, there is a need to "… create powerful agent construction toolkits, model-driven generators, and visual builders to quickly define and generate (large parts of) of individual agents and agent systems. We need libraries of agent parts, complete agents, and pre-connected agent societies." This highlights the importance of applications created from predetermined software agent-oriented systems, or *agent reuse*. Therefore, the reuse of existing agents is an important element of the of software reuse.

The foremost purpose of software engineering is to produce methods, techniques and tools to develop software systems with high levels of quality and productivity. Software reuse is one of the central approaches proposed to address these software engineering goals due its manifold advantages. Software reuse techniques have contributed to significant improvements to reduce both cost and time invested in developing projects, as well as increased adaptation to different requirements and needs of software engineers and architects. Other advantages of the reuse include [45]: the resulting software is more reliable, consistent and standardized; the increased flexibility in the structure of the software produced facilitates its maintenance and evolution; and the improvement software system

interoperability. Over the last years several reuse techniques have been proposed and refined by the software engineering community including component-based development, object-oriented application frameworks and libraries, software architectures and patterns [56][57].

Software reuse is a point that must be considered in the AOSE as well, bringing its benefits the multi-agent system development. Functionally specific agents can be reused in diverse agent teams to solve dissimilar problems. By constructing new systems out of reusable agent-oriented artifacts, creating large, high-quality agent-oriented software applications will be more efficient than is currently possible. In this scenario, agents are an advanced form of reusable software artifacts capable of exhibit interesting features like autonomous reasoning and goal-direct behavior. Today, although software artifact reuse is already established in the literature on software engineering, the work addressing agent reuse is meager and does not tackle the problem of identifying, organizing and storing agent-oriented artifacts for reuse. Therefore, the process of retrieving existing reusable agent-oriented artifacts from different application domains, or *agent retrieval*, is limited by the absence of appropriate mechanisms and standards. In this context, the entire agent retrieval process, which includes identification, storage and maintenance, turns into a crucial impediment to be overcome in AOSE.

A serious problem involved with reuse is the location of appropriate software reusable artifacts, made difficult because potential reusable artifacts are created by others, making it challenge identify the appropriate artifact in less time than it could be developed. The objective of searching artifacts is to leverage the information captured in these artifacts and find those with similar functionality or some other attributes to the desired specifications. Thus, instead of attempting a focused search, a larger set of artifacts is typically retrieved; from which the most appropriate are selected. These requirements are not easily achievable. In this case, software repositories appear as a solution, which must have an information retrieval system that, according to user needs, identifies and locates appropriate reusable artifacts. The effort that is involved in locating appropriate artifacts is considerable due to the diverse range of artifacts that might be included, as artifacts are distributed in several sources. In addition, there is little documentation about what the artifacts are and how they should be used. In a

repository where the spectrum of artifacts can grow exponentially, the task of describing the artifacts becomes significant. Without guidance towards what might actually lead to an optimal repository population, the probability of a successful search is reduced.

In this context, from a reuse viewpoint, there is another challenge to be considered: to model artifacts parsimoniously and intelligibly in a way that can be matched against software engineers' implementation needs. For a software agent to be effectively reused, its specifications must be flexible enough and easy to adapt to the many variations that exist in an application domain, for example using a meta-model. The benefits of a meta-model, for any given domain, include: (i) domain concepts are easier to apply for newcomers (concepts would be present in the single meta-model instead of looking for them in a spread out collection of existing ones); (ii) increased portability of models across supportive modeling tools (they would refer to the same meta-model); (iii) better communication among researchers (they could use the same frame of reference, i.e., the unified meta-model); and (iv) the research could focus on improving and/or realizing the unified meta-model instead of being spread across a number of existing meta-models.

According to published literature [78], the gap between the problem formulation in user's mind and the artifact description in the repository is the main obstacle to retrieve an artifact that corresponds to the software engineer's need. Writing a good query is not an easy task for several reasons. First, users may not be familiar with the vocabulary used to describe particular concepts in the search space (i.e., the collection of software artifacts). Other factors inherently affect some retrieval techniques in other fields as well. A common example that contributes to the ambiguity of a query is synonyms and homonyms. In addition, these retrieval techniques do not use predefined vocabularies or grammars, which makes them fast and robust, but also limits their retrieval performance in such cases. Thus, descriptions must describe the artifact architecture, functionalities within some domain, relationship between other artifacts and supplementary properties. Consequently, artifacts description becomes complex not only making it hard to describe new artifacts and their common and variable features in a specific domain, but also to identify appropriate artifacts for the user.

In order to productively retrieve artifacts, a software engineer needs to have a constant awareness of existing reusable artifacts, by browsing through similar artifacts in the repository, or querying structural content of artifacts. The types of information related to software artifacts include: structural information (operations, concepts, control and dataflow), dynamic information (behavioral aspects of the program), and lexical information (problem domain and developer intentions). The schema of a repository itself often does not consider semantic relationships among components and thus omits important retrieval information. The information in existing repositories needs to be structured in a way that takes into account the meaning (semantics) of the artifacts. In this scenario, different practices should be experimented with to consider semantic relationships during the retrieval process. We propose a retrieval model to represent different information, structures, behaviors, relationships of software agents, and provide search and recommendation methods based on semantics to support the agent reuse.

1.1. Contributions

To undertake the problem of retrieving agent-oriented artifacts, we propose a methodology and a repository prototype based on semantic web technologies that exploit reuse in agent-oriented development among different agent architectures, platforms and programming languages in diverse application domains. Our prototype is implemented as a web-based repository which stores and shares reusable agents already created.

Firstly, it includes a flexible and scalable meta-model for representing the heterogeneous agent-based artifacts and their common and variable features, which are formally modeled by means of ontology and feature model. The meta-model was derived from a literature review of various already proposed agent specifications. It contains requirements that are more complex since each agent platform uses its own data structures and its own interaction interfaces. Moreover, the conversion between specifications cannot be done automatically, as there is no mapping process that will do this in a transparent and efficient manner. To make

this manageable, we explore how parts of an agent can be modularized and be sufficiently generic in order to be reused.

Secondly, we define two taxonomies in order to establish agent description protocols and to reduce the time invested during the retrieval by mapping specific attributes of stored agent-oriented artifacts, (i) context categories represented by domain-specific taxonomy to classify agents according their application domain, and (ii) an application domain taxonomy to structure intrinsic characteristics of software agents like autonomy and mobility.

Thirdly, we create a recommendation system that allows end-users to discover the existence of reusable interrelated agents, and to learn new information or agents as needed improving user productivity and promoting agent reuse.

Fourthly, we make a subscription service to announce updates to the agents that are associated to specific categories, allowing users to know about updated information regarding stored agents. The recommendation system and the subscription service keep users aware of such agent's existence or the need of such agents.

Finally, we implement enhanced search and browsing methods constituting the semantic retrieval model that supports the reuse of software agent-based artifacts on different domains. The semantic retrieval process includes the following phases: semantic indexing, query processing, searching and raking.

We argue that the combination of modern information retrieval practices and semantic web technologies provides a rich and effective retrieval system, helping to overcome the existing limitations on agent-oriented software engineering. We evaluate the constructed repository and verify that the proposed reuse method is an improvement in terms of the relevance of retrieved agent-oriented artifacts.

1.2. Outline

The remainder of this thesis is structured as follows. Chapter 2 introduces the basic concepts mentioned in the approach. Chapter 3 examines the related work and explains the need for reusing agent-oriented artifacts. Chapter 4 deals

with our methodology and repository prototype adopted. Chapter 5 evaluates the research, gives some remarks identifying the major issues addressed to support the agent reuse, and analyzes the limitations of our approach. Chapter 6 shows the tool that supports our study. Finally, Chapter 7 concludes the work presenting research directions in the area for future work and outlines a brief discussion of open problems, challenges, issues that must be addressed if agents are to achieve their potential as a software engineering paradigm.