Introducing Join-Computing

GIACOMO CABRI
Dipartimento di Ingegneria dell’Informazione
Università di Modena e Reggio Emilia
via Vignolese 905, 41100 Modena
ITALY

Abstract: Engineering the interaction between components is a key issue in the development of complex software systems. While technologies for developing components are spread, more difficult is the composition of such components in heterogeneous, unpredictable and dynamic scenarios. In this paper we introduce the Join-Computing as a general model in which components are able to “join” in order to exploit each other features. We discuss also the technological issues related to the Join-Computing, which can be exploited to concretize the model.

Key–Words: Software components, Software composition, Agents, Services, Interactions.

1 Introduction

Developing complex software systems is a challenge that is becoming harder and harder, and calls for appropriate strategies to face the related issues. Pervasive computing [19], Web applications, autonomic computing, and ambient intelligence are the more representative application fields that exhibit challenging features of heterogeneity, unpredictability and high dynamism.

In the field of software engineering, different paradigms have been proposed to address the engineering of complex software systems. Service Oriented Computing and software agents are two significant ones, which have features that help in facing the above-mentioned issues, but exhibit also some limitations that do not make them the definitive solution.

In particular, we identify one major issue in the lack of definition for the capability of these kinds of composing software components. Some approaches have been proposed, but we point out the lack of a more general view.

To this purpose, in this paper we introduce the Join-Computing as a style for base components to “join” each other to exploits their features. The paper is organized as follows. First, we introduce readers discussing the background of our approach (Section 2); then, we introduce the Join-Computing sketching its characteristics (Section 3), discussing the technological issues and how they can be faced (Section 4); finally, we report some conclusions and future work (Section 5).

2 Background

Developing complex software systems requires appropriate models and paradigms to face their features of heterogeneity, unpredictability and high dynamism.

Service-Oriented Computing (SOC) is gaining ground thanks to its simplicity and robustness, and allows also service assembling to carry out higher-level tasks [16]. EU has paid specific attention to this field, in particular with the FP6 project SECSE (http://secse.eng.it), which aims at defining methods, tools and techniques for specification, discovery, design, and management of services. Services provide effective mechanisms to accomplish tasks providing a high degree of separation of concerns, but they are likely to lack flexibility, since they are mainly passive entities. Moreover, a single service could not accomplish an entire task, so effective composition is needed.

Agent-Oriented Paradigm (AOP) relies on autonomous software components that exhibit a high degree of flexibility and adaptability in dynamic contexts [21]. Agents represent a useful paradigm to develop complex software systems [10]. The sociality of agents enables them to form societies where they play roles and are subject to organizational rules [22]. The EU network of excellence called Agentlink (http://www.agentlink.org), concerning agent-based computing, has been supported for almost ten years proofing the importance of this research field.

Agents provide the required flexibility and adaptability to face dynamic and complex situations, but they result difficult to be effectively exploited in par-
ticular because of the lack of standardization and intuitive tools.

From this short overview about these state-of-the-art computing models, we can state that a general limitation concerns the fact that they are enabling approaches in defining components (services or agents, respectively), but they lack in defining appropriate ways about how these components can interact each other. In the former case, composition and orchestration of services is usually carried out in a manual way and relates only to syntactic aspects. In the latter case, interactions derive from sociality of agents, which nevertheless often remain an abstract feature that hardly provides a useful impact on the system development.

In the literature we can find different proposals that address a static composition of software components [2, 5, 6, 11, 14]. Mainly, they provide a skeleton or a framework in which the components can be integrated. Of course, their main limitation is that they provide a static composition.

Other approaches propose dynamic composition [4, 8, 9, 13, 17, 18]. Most of them are based on component standards (such as JavaBeans) or focus on services. Even if they are dynamic than the previous ones, they address specific issues, while the situation calls for a more comprehensive approach.

Some steps in the direction of overcoming the above mentioned limitations have been done by Autonomic computing [12], a recent research field that concerns software components that are able to self-manage in order to work correctly or to achieve a task.

As a final note, we can see that another general limitation of the existing approaches is that they usually address only some parts of the complete picture; usually only one or two phases of software engineering are considered and supported, leading to fragmented solutions.

We argue that the next frontier of the software research will focus not on single components, but on sets of components that must interact with each other and exploit each other features. The "self-healing" feature of autonomic computing components must be extended to couples or sets of components that are required to aggregate to perform higher-level tasks.

3 Join-Computing

We propose a new approach to engineering complex software systems based on components. We call this approach “Join-Computing” because it relies on software components that not only work together, but also they are enabled to aggregate by actively join each other and to exploit each other features. We envision a sort of software “ecosystem”, where the actors are actively involved in providing high-level functionalities and carrying out tasks by interacting and exploiting each other.

Our research aims at an approach that enables different components to fruitful interact despite their brand or technology (tackling heterogeneity); the approach should enable components to adapt to other components’ behaviour that is not expected (tackling unpredictability); all this must happen at runtime, in a dynamic way (tackling dynamism).

The interactions between components can be envisioned as follows (see Figure 1).

1. **Joining point**. At this stage, the components dynamically discover each other features and how to exploit them;

2. **Interaction**. This is the period of time in which the components interact to actually exploit each other in order to carry out a task or a part of it;

3. **Release point**. After the exploitation, the components release the relationship that bounded each other.

![Figure 1: Join-Computing: 1) joining point; 2) interaction; 3) release point.](image)

In our opinion, the following advantages are provided by our approach.

- **Generality**. The proposed approach is quite general, and not related to specific situations, so it is effectively adaptable and flexible.

- **Locality**. Interactions could be modelled taking into consideration the context where they occur, with its features, rules, and so on.
Separation of concerns. Adopting such an approach (and of course relying on an underlying support), allows developers to separate composition issued from logic issues, leading also to a more modular approach. At the same time, the concrete implementation must meet the following requirements.

- Peculiarity. While following a general approach, it must preserve the peculiar features of the involved components.
- Reusability. The implementation must enable an easy and wide reuse not only of software code but also of solutions.
- Practical usability. Besides possible formalisms for modelling systems, we argue that the engineering systems must be supported and simplified in a concrete way.

An important aspect is that our approach will concern different levels that we have identified as follows:

- Methodologies; this is the starting point to provide a comprehensive support to developers; a methodology explains the different steps that developers must enact to address the development in the different software engineering phases;
- Tools; methodologies must be associated to tools that provide the concrete support and help developers in punctual decisions;
- Infrastructures; to complete the picture, operative infrastructures should provide runtime support to applications, accordingly to the

methodologies and tools exploited during the development; as better explained later, we think that infrastructure represent a key point for our approach.

4 Technological Issues

In this paper we have introduced the Join-Computing as a style of software composition.

What we call Join-Computing is a “style” with which software components can be composed and can work together. Of course, it must be supported by appropriate technologies, which make it concrete and usable.

To this purpose, we identify two main kinds of technological support, the former provided by components themselves, the latter by infrastructures. In the following we will explain them.

4.1 Components

Of course, we cannot assume that whatever components can easily join each other. But our aim is not to involve only ad hoc components.

For instance, we can exploit the proactivity feature of agents to let them discover the other components (and in particular services) available, and the sociality feature can turn out to be useful in order to negotiate the interaction issues. But, as we mentioned previously, these features are not always easy to exploit in this way.

The better solution would be to add components the capability of joining with other components, in terms of interface. For existing components, we can provide wrappers that carry out this task.

We remark that, even if it could seem easier, supporting Join-Computing directly in components can be not straightforward, leading to specific solutions difficult to maintain. As better detailed in the next subsection, in our opinion infrastructure-level solutions can be more suitable.

4.2 Infrastructures

Even if the more intuitive support is provided by the components themselves, we must not disregard the chance of involving infrastructures and embodying in them part of the needed support.

We can assist to this trend in different software fields, where some functionalities are delegated to the infrastructures. For instance, in the software agents field, interactions can be enabled [3] and ruled by an underlying infrastructure [7]. Often an infrastructure implements the concept of environment, which can be considered as a first-class entity in the development of applications [20].

There are several advantages of considering infrastructures as support for composing software components: first, the management and maintenance is made easier since it is centralized; then, an underlying infrastructure can have a more global view and can also suggest appropriate partners for the joining; moreover, a trusted infrastructure can access components’ and environments’ resources in a safer way than possibly unknown components.

In our case, an underlying infrastructure can provide the following “meta-services”:

1. Joining point. Publication of the features of the components, possibly in a standard format (for instance, let us mention WSDL); enabling the search for features; enabling a component resolution system;
2. Interaction. enabling the interaction between components, for instance providing channels, dataspaces, events or something similar;

3. Release point. Enabling a graceful disconnection, which takes care of resources exploited, accounting, and so on.

An approach that can be proposed is the one based on knowledge networks [15]: inspired by biology, it provides mechanisms of information crunching and re-assembly that enable to semantically find services even based on different interfaces or standards. Originally thought for services, and in particular for aggregating services upon clients’ requests, it can be fruitfully exploited also for joining components. This can be useful to face the above mentioned heterogeneity, unpredictability and high dynamism.

5 Conclusion

In this paper we have introduced the Join-Computing, as an approach to engineering complex software systems, where the components can “join” each other in order to exploit their capabilities. Our research is at its early stage, but we have sketched its path along with some technological hints.

We remark that providing appropriate and effective models, tools and technologies not only help in engineering complex software systems, but also in better understanding them. In fact, often the emergent behaviour is more important than the behaviour of single components; however, driving or simply predicting the emergent behaviour is not a trivial task, and requires appropriate models.

Further, in the last years we assist to a scenario where technologies are rapidly evolving often in a separate way from computational models; this has been likely to produce a “gap” that we aim at filling in our approach. Well-funded technologies are not only better understandable, but also more effective and more controllable.

With regard to future work, we are going to study the existing methodologies to evaluate whether and in which degree they can be used to support the development of joining components; to this purpose we can exploit the fragment approach [1], i.e., extracting useful fragments from existing methodologies in order to meet the specific requirements.

A parallel study concerns infrastructures, since our opinion is that embodying joining mechanism in the underlying infrastructures leads to key advantages. We do not disregard the idea of reusing existing infrastructures, extending them to accomplish our needs.

Acknowledgements: Work supported by the Italian MiUR in the frame of the PRIN project MEnSA - Agent oriented methodologies: engineering of interactions and relationship with the infrastructures.

References:


